Good code

29-Oct-2020

Objectives

Software engineers work in teams to deliver large projects to customers who are not software engineers. This means that their code is often read by their teammates, no single person knows everything about the codebase, and the code must be resilient to mistakes made by customers during operation. Thus, an important skill in software engineering is to write code that is understandable by other developers and resilient to errors. As a software engineer, you will need to know and use common “patterns” that make it easier to understand and discuss modules and programs.

In this module, you will learn:

- How to write readable code
- How to architect clear modules
- How to architect clean programs
- How to handle “bad” inputs and environments

Producing write-once/read-often code

Code is not written only for the compiler, but rather to be understood by humans, and programmers spend the majority of their time reading code rather than writing it, since maintenance is a big part of the life of a software product. Therefore, readability is a key quality of a piece of code. The less time is required to understand someone else’s code, the less time is taken to start implementing features using that code and programming becomes more productive. One way to try and unify the methods used to write code is through coding conventions. These include language-specific conventions, general variable naming rules, and proper comment-writing.

> *“Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.”*
>
> Clean Code, by Robert C. Martin

Code conventions

Coding conventions are a set of guidelines that programmers should follow when writing code in any language. They usually cover coding style, such as indentation, comments, declarations, documentation, and naming conventions, but also cover programming and architectural best practices. For example, in Java, lowerCamelCase is used for variables and methods as a convention, and UpperCamelCase is used for class names. Conventions may be defined by multiple parties such as companies or simply the creator of the language. Programming teams should always agree on a set of conventions, known as a style guide, at the start of any project. As an example, the Google Style Guides cover the style guidelines for many of the languages used at Google, such as Java, Python or C++. These guidelines are also used in many projects outside of Google, so be aware of them when writing code in a specific language.

Every common programming language has globally accepted conventions that any programmer can follow. A recent trend of modern languages such as Go or Rust, is to enforce conventions at compile time by warning the user about inconsistent naming and practices. This produces very consistent code across projects which is very desirable.

Most languages also have tools to help you produce clean and consistent code. We distinguish two categories of such tools:

- Linters, such as ESLint for JavaScript and Flake8 for Python. They perform static code analysis to detect programming errors, bugs, stylistic errors and code with potentially unintended results. Most editors and IDEs have the ability to run linters in the background as you type.
- Formatters, such as Prettier for JavaScript and Black for Python. They automatically parse your code and reprint it according to a set of opinionated formatting rules, which saves time and facilitates code reviews.

A collection of linters and formatters for a variety of languages can be found here.

It is almost never the case that a piece of code is read by a single person and maintenance is a big part of the life of a software product. Coding conventions help with readability and reduce the effort required to understand and maintain the code. They also make sure that the code is clean and well-packaged in the case where it is shipped as a product. In summary, code conventions allow programmers to focus more on the functionality and spend less time arguing about conventions.

"*Maintenance typically consumes 40 to 80 percent (average, 60 percent) of software costs. Therefore, it is probably the most important life cycle phase of software.*"

Fact 41, Facts and Fallacies of Software Engineering, by Robert L. Glass

Effective naming

Effective naming of variables, functions and classes is an important aspect of writing good code, as it reduces the effort needed to read, understand and reuse code.

The key aspect of effective naming is **clarity** and **readability**.

For example, consider the two (Python) functions below. While the first one is syntactically correct, its purpose is not at all evident. Contrast this with the second function, which is much clearer, as it clearly conveys the intent and meaning of the code without the need for extra comments.

```python
def func(a, b, c, d):
    if a > b:
        foo = a * c
        bar = (a - b) * d
        p = foo + bar
    else:
        p = a * c
    return p

def compute_weekly_pay(hours_worked, regular_hours, hourly_rate, overtime_rate):
    if hours_worked > regular_hours:
        regular_pay = regular_hours * hourly_rate
        overtime_pay = (hours_worked - regular_hours) * overtime_rate
        pay = regular_pay + overtime_pay
    else:
        pay = hours_worked * hourly_rate
    return pay
```

There are no exact rules regarding how to name variables and functions, and naming conventions can vary depending on the context. Nonetheless, here are a few general guidelines that can help you write cleaner code (with examples in Python):

General guidelines:

- Use easily pronounceable names. Avoid using hard-to-pronounce abbreviations in an attempt to shorten a name. For example, `cchd_pg_hdr` is much harder to memorize than `cached_page_header`, despite being shorter.

- Enforce consistency across names. Choose one word per concept and stick to it, avoid using synonyms interchangeably in your code.

Booleans:

- Prefix variables with **"is"**, **"can"** or **"has"** so that the type of the variable can easily be inferred.

```python
# bad
open = True
fly = False
gluten = False

# good
is_open = True
can_fly = False
has_gluten = False
```

Lists / arrays:

- If they are designed to contain multiple values, **pluralize** the variable name.

```python
# bad
country = ["Spain", "Brazil", "New Zealand"]

# good
countries = ["Spain", "Brazil", "New Zealand"]

# great - "names" implies strings
country_names = ["Spain", "Brazil", "New Zealand"]
```

Functions / methods:

- Functions should be named using a **verb,** and a **noun.** If a function performs some action on a resource, reflect it in its name.

```python
# bad
def total_of_items(items):
def shape(points):

# good
def compute_total(items):
def draw_shape(points):
```

Class variables:

- Keep useless context out. There is no need to put the class or module of the variable in its name.

```python
# bad
class Student:
   def __init__(self, name, age):
       self.student_name = name
       self.student_age = age

# good
class Student:
   def __init__(self, name, age):
       self.name = name
       self.age = age
```

The guidelines discussed so far (and more) can be found in these two **short** articles:

- Writing good variable names
- The art of naming variables

Indicating word boundaries:

There are different methods of delimiting words in identifiers. Most notably:

- `flatcase`
- `camelCase / lowerCamelCase`
- `UpperCamelCase / PascalCase`
- `snake_case`
- `CONSTANT_CASE / SCREAMING_SNAKE_CASE`
- `kebab-case`

These naming conventions usually depend on the language you're using. For example, Java recommends `lowerCamelCase` for variables and methods, and `UpperCamelCase` for classes and interfaces, while Python recommends `snake_case` for variables and functions / methods, and `UpperCamelCase` for classes.

Effective comments

Comments are another tool to further help other developers understand code, and they do this in several ways, which we will discuss in this section. The most well-known use of comments is to **describe code**. There are a few general guidelines that should be followed for effective code description using comments:

- Comments should not repeat what the code does. They should only clarify the programmer's intent.
- Comments cannot fix bad code! If code needs comments to be understood, it is preferable to refactor it.
- Comments should be kept short and concise.

We can summarize these guidelines as the following: if you aren't confused by what a piece of code is doing, but rather why it's doing it at that moment, then a comment should be added.

As we said earlier, comments have other uses beside code description. They can be used to **include metadata** about a program file, like licensing information, name of the creator, file version, change log, etc... or be used to provide information that can be exploited by external tools such as Symfony in PHP. One can also use comments to leave **TODO notes or warnings for other programmers**. However, a more interesting use of comments is for **documentation**.

The purpose of documentation is to help external users of your code understand what it is and how they can use it. This overlaps with the general purpose of comments, which is why tools have been developed that allow comments to be used for documentation. The most important pieces of code to write documentation for are the public APIs of classes. They should contain a description of what each API method will do, what its arguments are and what effect they have on the method output, and what value is returned.

There exist many formats and tools for writing documentation. These tools scan all source files and generate summaries of annotated code in formats such as PDF, HTML, etc… They can also warn the user about undocumented code. The most well-known tools are Doxygen and Javadoc (for Java).

Effective commit messages

Commit messages are just as important to understanding code as are good comments. Other developers look at changelogs to understand how a piece of software has evolved, and they look there for explanations of why a particular change was made. And so does your future self. Please re-read the "Five rules for commit messages" section in the DevOps lecture notes.

How to approach module design?

We have seen in the previous weeks that modularity is the #1 tool to fight against complexity, and that it can take many forms (functions, objects, files, libraries, containers, …) at many different levels (code, process, OS, physical machine, ...). However, building a good modular system is not easy: where to define the boundaries? How to deal with dependencies between modules? What should be hidden vs what should be exposed? What existing tools should be used and what should be written from scratch? In this section, we will cover a few techniques to produce modular code, in particular discussing **module interfaces** and a number of **design patterns**.

Modularity enables quick prototyping

One major benefit of modularity is reusability, and once many modules have been produced to perform various tasks, it is very easy for any programmer to stitch modules together to produce working code very rapidly. This is one of the reasons why Python is so successful as a programming language for prototyping. There exist a lot of libraries and frameworks that enable programmers to achieve very complex tasks with only a few lines of code.

Moreover, the open-source community provides an incredibly large amount of libraries that can be used by anyone and technologies formerly available to few people like image processing, machine learning, computer graphics processing are now widely usable. Now is maybe the best time ever to start to use these tools and learn about various subjects by building projects, during your own time or in the context of Hackathons which are always a great learning experience.

As an example, here is a project from Lauzhack 2018 which aimed at providing an augmented reality interface for industrial machines. This project made use of a lot of techniques like image recognition, pattern matching and 3d tracking, that all needed to run on an Android device. This was all made possible using Google's ARCore library which allowed the whole project to be implemented in only a few files.

Module interfaces

Modules should have well-defined interfaces that require low context and use well-known terminology. Let's talk about these three points in order.

First, as we saw in the Modularity and Abstraction lecture, modules should have a **well-defined interface** that **abstracts away their implementation details**. For instance, there should not be additional parameters to tweak internal details unless this is required for some reason; even if it is, providing defaults for these parameters is a good idea.

Second, module interfaces should **require low context**, meaning a developer should not need to know about too many other interfaces in order to understand a single one. This means modules should not have too many externally-visible dependencies; while it is fine to use some dependencies for the module's implementation, try to avoid making these dependencies trickle down to the interface. For instance, a geo-location module that happens to use the Android operating system's geolocation service should define its own types for inputs and outputs, rather than reusing Android's, which would require developers to know about Android types.

Third, modules should use **well-known terminology**. This applies both in terms of programming terminology and in terms of context. For instance, the term "long" is usually understood in programming languages to mean "long integers", i.e., integers with a wider range than the usual 32 bits. Using "long" to mean something else, such as referring to a size, would confuse programmers. Similarly, modules should use common names in the context they are working with, such as "a patient" in the context of a hospital instead of "a customer" or "a user". Sometimes these goals can conflict, such as "class" in the context of a university; prefer the user-facing terminology, which will make it easier to discuss with customers, or avoid the problem by using synonyms such as "course" if they make sense in context. One common kind of well-known terminology is design patterns, which we will see next.

Design patterns

Engineers often meet similar problems in similar contexts.

For instance, while there are many rivers in the world, the problem of crossing a river is common, the idea of building a bridge on a river is common, and the solutions will look similar.

It is possible to build a bridge on one's own, without any knowledge of other bridges and without talking to other engineers. However, the resulting bridge may not work very well, since building a bridge is a complex task that is not suited for a single person.

Instead, engineers build a shared knowledge base of bridge blueprints, or **patterns**, that they can reuse whenever they need to build a bridge. This does not mean all bridges will look alike, but they will share overall designs that make them safe.

Software works the same way: there are many similar contexts, such as sharing data, editing documents, or buying things. They have similar problems, such as retrying failed requests, caching data, or validating user input.

To avoid writing solutions from scratch, which will look like “spaghetti code” with no modularity in sight, a software engineer can use the shared knowledge base: design patterns. These patterns help the developer orient their thinking in a modular way while facilitating the solution of common problems.

Some design patterns become so well-known that they are integrated into programming languages. For instance, the Iterator is a design pattern. While it may seem like a fundamental concept that does not deserve a name, it only appeared in Java in version 5! Before that, each developer had to write their own implementation of the pattern, adding complexity to every application and preventing libraries that used different implementations of Iterators from interoperating. Imagine having to re-write a method to convert an iterable to a list for every library you use, because they all implement their own iterable type!

An important fact to remember is that there is no silver bullet: using design patterns does not magically make for good code. **With added flexibility comes added complexity**. Old Java frameworks such as Spring and Apache tend to use so many patterns at once that their objects have comical names, such as Spring’s `SimpleBeanFactoryAwareAspectInstanceFactory`, or Apache’s `RequestProcessorFactoryFactory.RequestSpecificProcessorFactoryFactory`.

Refresher: what you (should) already know

These are patterns covered by the 1st year introductory class in computer science at EPFL.

You should skip this part if you remember these patterns.

Builder

Consider the following constructor:

```
Rectangle(int width, int height,
            int borderThickness,
            Color borderColor,
            boolean isBorderDotted,
            Color backgroundColor,
            boolean hasShadow,
            int shadowLeftOffset,
            int shadowTopOffset, ...)
```

It has multiple problems. There are too many arguments to remember. Some arguments are dependent on each other, e.g. it does not make sense to specify a shadow offset if there is no shadow.

Most arguments are of a primitive type; thus, an invocation of this constructor is hard to read:

```
new Rectangle(
  100, 200,
  10, Colors.BLACK,
  false, Colors.RED,
  true, 0, 10,
  ...
);
```

Instead, we can use the **builder pattern** to express construction in a nicer way:

```
new RectangleBuilder(100, 200)
   .withBorder(10, Colors.BLACK)
   .withBackground(Colors.RED)
```

```
    .withOffsetShadow(0, 10)
    .build();
```

This is much clearer: each group of arguments is now its own method, which programmers can choose to call or not. The implementation of the builder then looks like this:

```
class RectangleBuilder {
  RectangleBuilder(...) { ... }
  RectangleBuilder withBorder(...) { ... }
  RectangleBuilder withDottedBorder(...) { ... }
  RectangleBuilder withBackground(...) { ... }
  RectangleBuilder withOffsetShadow(...) { ... }
  Rectangle build() { ... }
}
```

There are two key parts of a builder. First, the methods to configure the object return the builder itself, so that developers can chain them as in the example above without needing to declare a variable and spell out its name every time. Second, the method build() returns the built object.

Another benefit of builders is that if a programmer wants to build objects that are similar except for a small number of differences, they can configure the builder with the common parts, then configure the differences and build each object. They only need the calls for the common parts once.

Decorator

Requirements are often orthogonal: fetch data, and cache it, and retry if it fails, and clean the data, and other things. All these “ands” are linking orthogonal requirements together: for instance, the code that cleans data does not need to know about the code that retries data fetching.

Furthermore, there may be multiple modules that all fetch data, and all need to have retrying and caching logic. Copy-pasting this code into each module would be a maintainability nightmare.

Instead, we can use the **decorator pattern** to augment a class.

The core idea is that a decorator “wraps” an interface implementation and exposes the **same** interface. Thus, from an outside point of view, the decorator is not visible. This also means decorators can compose: a decorator can wrap another decorator, without being aware of it.

In this figure, the green wave is the interface that each module exposes. The “cache” decorator wraps the “fetch” module and exposes the exact same interface to the outside world. A developer can then choose which decorators to use and in which order, without changing the code that uses this interface.

Let us now see how to write a decorator.

```
interface SocialNetwork {
      List<User> getUsers();
}
```

Given this interface representing a social network with users, and an implementation:

```
class GitHub implements SocialNetwork {
  ...
}
```

We can write a decorator that wraps a network and adds retry logic:

```
class RetryingSocialNetwork implements SocialNetwork {
  SocialNetwork wrapped;

  RetryingSocialNetwork(SocialNetwork wrapped) {
      this.wrapped = wrapped;
  }

  List<User> getUsers() {
      try { return wrapped.getUsers(); }
      catch (...) { ... }
  }
}
```

This decorator can then wrap our original implementation.

Composite

Applications often needs to handle groups of items, such as files and folders. Folders can contain files, and they can also contain other folders. One way to represent this is with the following code:

```
class File { ... }

class Folder {
  List<File> files;
  List<Folder> folders;
  ...
}
```

With these classes, any code dealing with folders must explicitly deal with both files and folders. This is not too bad when there are only two kinds of items, but the problem gets exponentially worse as more kinds of items are added, such as libraries that can contain folders, files and other libraries.

Instead, we can use the **composite pattern** to represent all items in a uniform fashion:

```
interface Item {
  List<Item> children();
}
class File implements Item { ... }
class Folder implements Item { ... }
class Library implements Item { ... }
```

From an outside perspective, all items look the same. Internally, each item can behave in its own way: for instance, a “search” operation on a file will search the contents of the file, while a folder will delegate the search to its children.

Developers can then add new items without having to change the existing ones; for instance, a folder no longer explicitly lists what can go inside it, thus a new “archived folder” type could go in a folder as long as it exposes the “item” interface.

Adapter

When building complex systems from parts such as third-party libraries, it is unlikely that all parts use the same models to represent various concepts. This is a problem for integrating parts because they cannot interact if they do not speak the same language.

This problem also exists in real life: one cannot use a Swiss three-pronged electrical plug on a European power outlet. The solution is an electrical adapter.

In software, this is the **adapter pattern**: an object that wraps one interface and exposes another.

For instance, consider the following service that gets the temperature of places:

```
interface Place {
  double getLongitude();
  double getLatitude();
}

interface WeatherService {
  int getTemperature(Place place);
}
```

This service represents places as a longitude/latitude pair. But another part of the system may represent cities in the following way:

```
interface City {
  String getName();
}
```

There are no coordinates in this interface, only a city name. To get the temperature of a city, we must write an adapter:

```
class CityPlace implements Place {
  CityPlace(City city) { ... }

  double getLongitude() { ... }
  double getLatitude() { ... }
}
```

This adapter will internally use location service to get coordinates from a city name. Using the adapter, we can now get the temperature of a city by wrapping it in an adapter.

Observer

Modules often need to react to some external event, such as from another part of the system. One way to do this is polling: continuously asking whether anything happened. This is fine if the answer is “yes” most of the time, such as a Wi-Fi router in a busy location asking its network card if any packets have arrived.

However, polling is inefficient if the answer is “no” most of the time, such as whether the user has clicked on a button. Instead, we should ask the button to let us know whenever something happens. We then do not need to interact with the button any more until something happens.

This is the observer pattern, and we can implement it by keeping track of who has asked for notifications, such as in this Button that remembers a list of click handlers:

```
class Button {
  private List<Runnable> handlers;

  void add(Runnable handler) { ... }

  void remove(Runnable handler) { ... }
}
```

Note that we can chain observers. For instance, our Button knows when to call the click handlers because it listens to changes from the operating system, which itself listens to the hardware to know when the user clicked the mouse.

More design patterns

Let us now see a few patterns that the 1st year computer science class did not cover.

Singleton

In a computer system, most objects have many instances, such as connections to the Internet or opened files. However, some objects must only have a single instance. Consider a user interface: many applications only have a single main window, and do not want users to create more than one since they manage multi-tasking within the application, such as with tabs. The same is true for application configuration: once the user has changed their settings, the entire application must use the same, updated settings.

One way to solve this problem would be to synchronize all instances of some objects, so that they always have the same value. However, this is tedious and bug-prone.

Instead, we can use the **singleton pattern** to ensure there is only a single instance of an object.

For instance, this configuration class can only be accessed through a static method, which can internally make sure that it always returns the same instance:

```
class Config {
  private Config() { /* ... */ }

  private static final Config INSTANCE = new Config();
  static getInstance() { return INSTANCE; }

  // or, "lazy" version:
```

```
  private static Config INSTANCE;
  static getInstance() {
    if (INSTANCE == null) { INSTANCE = new Config(); }
    return INSTANCE;
  }
}
```

In Java, one can use enums to achieve the same purpose at compile-time:

```
enum Config {
  INSTANCE {
      /* contents of the class here, e.g. fields, methods, ... */
  }
}

// Then use like this, assuming there is a “getSomething” method:
Config.INSTANCE.getSomething();
```

Singletons are useful in the high-level application code that glues modules together. However, you should avoid using them in low-level modules, since they are the equivalent of a global variable. If each module has an implicit dependency on some singleton, writing tests becomes painful because of the global state. Wikipedia has more details, including code snippets for many programming languages.

Factory

Polymorphism allows us to have different implementations of an interface to perform different actions, such as reading from XML or JSON files.

However, one cannot always know which implementation to use at compile time. For instance, if users can choose which file to open, the file may be XML or JSON and the code cannot know this at compile time. Thus, we must have a way to select the implementation at runtime.

Ideally, we would like to return the correct implementation directly from the constructor of the base class. Unfortunately, this is not possible in languages like Java, where a constructor can only initialize an object and not choose which subtype to return.

Instead, we can use the **factory pattern** by creating a method that chooses which subtype to return based on its argument:

```
interface RecordReader { /* methods to read records */ }
class XmlRecordReader implements RecordReader { /* XML implementation */ }
class RecordReaderFactory {
  static RecordReader get(String path) {
    if (path.endsWith(".xml")) {
      return new XmlRecordReader(path);
    }
    ... JSON, CSV, ...
  }
}

// Usage:
var reader = RecordReaderFactory.get("file.xml");
```

This allows us to cleanly separate the modules that handle different file formats from each other, and to provide a single entry point for the rest of the system. Other modules can use this factory without knowing which implementations exist.

Factory methods have other benefits compared to constructors, such as having a specific name (one could imagine creating a record reader either from a path or from text directly, but both of these are Strings so they need different names), and being able to return cached objects if needed. See this article for more information.

Proxy

Some resources can be present either locally or remotely, such as files on the local machine and on cloud storage. Apps could handle these cases separately, but this increases code complexity, and implies that existing apps will not be compatible with any new remote cases.

Instead, we can use the **proxy pattern** to treat remote resources as if they were local. The application believes it is communicating with a local resource, but in fact this resource is a proxy that communicates with a remote resource under the covers.

Consider this snippet:

```
class Folder { /* ... */ }

interface FileSystem {
  Folder getRootFolder();
  /* ... other methods ... */
}
```

The file system interface could be implemented in a class named `LocalFileSystem` for a given drive, like so:

```
class LocalFileSystem implements FileSystem {
  private final String root;

  public LocalFileSystem(String root) { this.root = root; }

  public Folder getRootFolder() { return this.root; }

  /* ... other methods ... */
}
```

But it could also be implemented by a class that passes along the request to some remote endpoint:

```
interface WebClient { /* methods to talk to a remote server */ }

class RemoteFileSystem implements FileSystem {
  private final WebClient client;

  public RemoteFileSystem(WebClient client) { this.client = client; }

  public Folder getRootFolder() {
    // ... create a web request corresponding to the "get root folder" operation ...
    // ... get a response ...
    // ... translate the response to a Folder, this method's return type ...
  }
}
```

Importantly, you should only use this pattern if the interface of the resource allows it. For instance, while it is theoretically possible to offload arithmetic operations to a cloud computation engine, it would be very odd for an app to crash while executing “1 + 1” due to a failing Internet connection. On the other hand, accessing files is a good match for this because handling files is already an asynchronous operation and already has failure cases such as “the file is not available right now”. Wikipedia has more details, including code snippets for many programming languages.

Visitor

Consider the following class hierarchy:

```
interface Shape { }

class Square implements Shape { ... }

class Triangle implements Shape { ... }

class Circle implements Shape { ... }
```

Imagine you want to add a feature that draws these shapes. You could add a method to the interface and implement it in every class. However, you may want multiple drawing methods for specialized tasks, such as drawing as PNG, as JPEG, as SVG… and shapes should not be aware of the intricate implementation details of image file formats.

Ideally, we would want to define methods like these:

```
void draw(Square square) { ... }

void draw(Triangle triangle) { ... }

void draw(Circle circle) { ... }
```

And then call `draw(getShape())`, where `getShape()` returns some Shape whose exact type is known only at runtime.

Unfortunately, languages like Java do not support this: the runtime types of method arguments are not used to choose which method to call, only the runtime type of the instance on which the method is called. What Java has is known as single dispatch; what we would like is multiple dispatch, as found in C# and Common Lisp.

We could solve this using type tests, like so:

```
if (shape instanceof Square) {
  ...
}
```

However, this is verbose and the compiler will not warn us if we forget to handle one of the subclasses.

(As an aside, note that languages such as Scala or C# have a “pattern matching” statement that does what we want: type tests without verbosity and with an exhaustiveness check by the compiler; this is coming in a future version of Java as well)

Instead, we can use the **visitor pattern**, which allows us to emulate double dispatch. Let us begin by declaring a visitor interface, with the methods we wanted to write in the first place:

```
interface ShapeVisitor {
  void visit(Square square);
  void visit(Triangle triangle);
  void visit(Circle circle);
}
```

We can then add a single method to the interface allowing visitors to visit it:

```
interface Shape {
  void visit(ShapeVisitor visitor);
}
```

We will be able to use this method with any visitor, thus we only need one method on the interface to add any number of methods to the hierarchy. The implementation in each class is pure boilerplate:

```
class Square implements Shape {
  void visit(ShapeVisitor visitor) {
      visitor.visit(this);
  }
}
```

Calling the `visit` method on a `Square` call will end up calling the `visit(Square)` method on the visitor, since "`this`" is statically typed to be a `Square` in this context. Other implementations of the interface should have the exact same code, whose effect will change due to the static type of "`this`".

Our drawing problem from earlier can thus be solved by writing a visitor:

```
class DrawingVisitor implements ShapeVisitor {
  public void visit(Square square) { /* ... */ }
  public void visit(Triangle triangle)  { /* ... */ }
  public void visit(Circle circle)  { /* ... */ }
}

// usage, assuming a "getSomeShape" method returning a Shape:
getSomeShape().visit(new DrawingVisitor());
```

We can then create any number of visitors for various kinds of drawings or other operations, without polluting the existing class hierarchy with unrelated concerns. This article contains further information.

Design Patterns Summary

Name	**Description**
Adapter	Wraps one object to expose a different but related interface to access the object.
Builder	Build an object step-by-step, with optional and mandatory steps.
Composite	Simplify an tree-like interface by making nodes expose the same interface as leaves.

Decorator	Adds functionality to an object, exposing the same interface.
Factory	Get an instance of an interface, without knowing which exact type it will be.
Observer	Watch an object for changes, and react to them.
Proxy	Delegate to a remote object, without local objects having to know about the delegation.
Singleton	Ensures there is only one instance of an object in an entire program, like a global variable.
Visitor	Add functionality to a class hierarchy without adding extra methods for each feature.

How to approach program design?

In the previous section, we discussed how to design building blocks of code which can be picked and tweaked to solve specific scenarios similar to common problems with comparable constraints. However, how should one organize an entire codebase, choose its rules and ensure that they stay consistent?

Inheritance vs. composition and containment

You have already studied inheritance, composition, and containment at length in your earlier courses. Inheritance allows you to reuse code and interfaces by inheriting code instead of duplicating it. Alas, programmers have a tendency to abuse inheritance, so always ask yourself whether containment (i.e., defining an object to contain another object instead of inheriting from it) is not a better choice. The key decision factor is whether object A "is a" B or "has a" B: If you can say that class A is a B, then inheritance is more likely to be appropriate. If you can say that A has a B, then it" likely that composition and containment is more appropriate. This is more precisely captured in the Liskov Substitution Principle.

If you would like to recap these ideas on inheritance and composition, check out the recap videos posted on Moodle. You will also get a chance to refresh your memory on multiple inheritance, the Diamond Problem, mixins, and general advice on inheritance hierarchies.

User interface design patterns

Many applications must deal with non-technical end users, thus an intuitive general user interface (GUI) is often required, which also drives the behaviour of part or the entirety of the software system (e.g. websites, mobile apps, desktop programs, consumer electronics, ...). However, mixing code that handles user displays and input events and business logic is a bad idea:

- How to refactor code when UI technologies change or when a redesign is needed?
- How to organize code such that designers and developers can work on it separately?
- How to unit test program logic independently from UI?
- How to share program logic across several platforms with different UI systems?

The core idea to solve these issues lies in the separation of concerns: at the most basic level, we want to physically split the code into different files based on its role. We will call the user interface “View” and the business logic and data representation “Model”. However, we need some additional glue to link the Model to the View: we will explore 3 variants called Controller, Presenter, and ViewModel.

Disclaimer: the terminology is quite loose and actual implementations can vary. You will find similar definitions that are sometimes interchanged, sometimes not expressed at the same level of abstraction. There is no strict or true definition as these patterns have evolved quite a lot over time, however the core concepts apply to all of them. In this course, we will stick to the definitions below, but you may encounter differences in existing frameworks or in the industry.

A model is the domain-specific data representation of an entity, and its associated logic. A model can be a set of fields (for instance, a set of integer fields), or a more complex object such as a collection, or a tree. Importantly, a model does not know anything about a view or a controller, and it only implements the data structure to hold its values, as well as business logic that manipulates it. For instance, a Java model is often a simple POJO.

A view on the other hand implements the display to the user. Often, it is desirable that the view is "as dumb as possible", i.e. it should contain no logic nor data, but only describe which type of UI elements are used (text, image, button, ...), and how they are arranged and styled. In many systems, the view has its own declarative language(s) (HTML / CSS for the web, XML in Android, dedicated View classes). Sometimes, an additional abstraction for views is provided in the same language as the model and the glue to facilitate interconnections, such as Android's View classes hierarchy. Views can often be composed through the Composite pattern, as they form a tree hierarchy (for instance a button and a label can be contained inside a form).

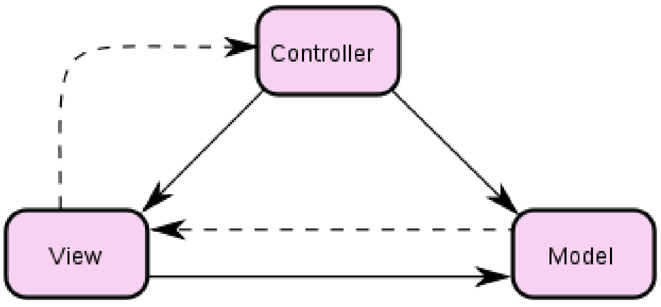
Depending on the framework and the level of abstraction enforced by the developers of a project, each of the model, view and glue (controller, presenter, viewmodel) may have their own dedicated interfaces that will describe the behaviour of their respective classes (for instance in Android, Activities are sometimes considered as part of the view and will implement an interface contract that is defined by the developers, while another project may define Activity as the Presenter itself). In some other frameworks, these roles are implicit (for instance in PHP Symfony, the view is simply HTML).

MVC

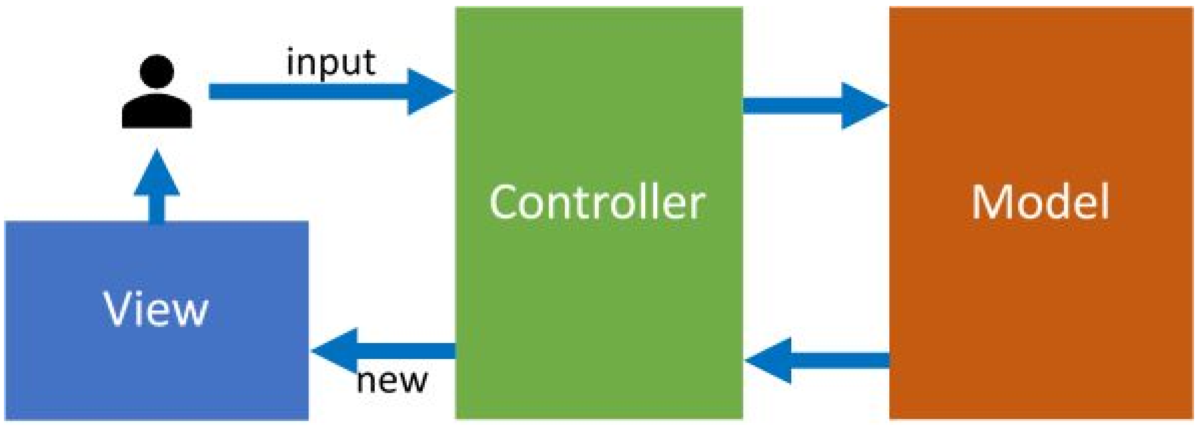
The first pattern to decouple View and Model is MVC, which stands for **Model-View-Controller**.

In the original Smalltalk MVC, the **Controller** is a class that dictates the lifecycle of the View and updates the Model data. The Controller decides how to mutate the Model and optionally the View after each user input.

Model, View and Controller form a triangle as follows: the Controller holds an instance of the View and the Model (solid lines), while the Model notifies the View of its data changes, and the View notifies the Controller of user inputs (dashed lines). The View must also keep a reference of the model (solid line), as it must know the shape and types of the Model and transform them to some corresponding UI displays. In Smalltalk, all models and views inherit from the Object class, which implements the Observable pattern: views observe models and delegate decisions to the Controller. Note that data updates in the model require no changes from the Controller in the View if the layout does not change, i.e. values of the Model are automatically updated in the view! The Controller will only mutate the View if, for instance, a different View must be selected.



However, following the evolution of web systems, in which the View lives in a client browser while Model and Controller sit remotely on a back-end server, modern usage of the MVC pattern usually refer to the implementation in which the **Controller** is an entity which receives user input, and creates a View to display based on the Model:



From the users' point of view, they talk to the Controller by interacting with the View (e.g. clicking a link) and receive back another View. Internally, the Controller talks to the Model. This is enough to decouple View and Model. However, it does not support incremental updates: the Controller must send a new View each time. In this flavor of MVC, a Controller may handle multiple Views and Models.

MVC is common for websites, in which it is normal to send a new page at every request. However, for desktop applications, replacing the entire view is a waste. Furthermore, even on the Web there is a trend towards applications that do not need to reload the entire page for each user input.

MVP

The **MVP** pattern, which stands for **Model-View-Presenter**, is a natural consequence of the way frameworks like Android work: user inputs must go through the View.

In MVP, the Presenter is a mediator between the View and the Model. The View calls the Presenter whenever it receives input, and the Presenter calls the View whenever it needs to update a part of the View. Similarly, the Presenter calls the Model when it needs to execute core logic. The View and the Model never talk to each other. Usually, there is a 1-to-1 mapping of Presenter to View. The difference between MVC and MVP lies in the fact that MVP allows the Presenter to update the View imperatively, without the need to rebuild a complete View object. For instance, a web page may use a Javascript presenter to detect user inputs, fetch data from the server directly, and mutate the HTML in-place, rather than load an entire webpage at each request.

Consider the following example:

When the user clicks the button, the View tells the Presenter, which calls the Model. The Model then returns the proper greeting, which the Presenter tells the View to display.

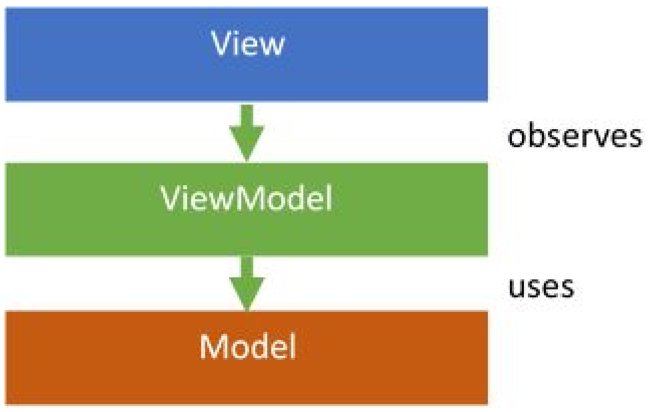
MVP allows for incremental updates, unlike MVC. However, if the Presenter does not raise the level of abstraction by implementing a higher-level interface, it remains intrinsically tied to the UI framework; for instance, handling button clicks is not the same in Android and on Windows.

In practice, MVP is the default way to write applications in Android, and a common way to write UI applications in general.

MVVM

The core idea of the next pattern is that the View can observe data changes, instead of explicitly being called to update something specific. Indeed, in the MVP model, each change from the model must be explicitly forwarded to the view and vice-versa: however, modern applications are very complex with thousands of UI elements and models and any forgotten call will cause a desynchronization of state and view, which proved to be major source of bugs. To solve this issue, MVVM systems automate the bindings between them.

This leads to **Model-View-ViewModel**, or MVVM for short: rather than imperatively mutating the state and the view at each user input, data flows automatically from the model to the view. Thus, View, ViewModel and Model have a “depends-on” relationship as follows:



The ViewModel is an intermediate representation which contains data and commands, as well as an implementation of the Observer pattern to allow Views to be notified of changes. Importantly, it does not describe how the data should be displayed, but only what to display. To reuse the example above, while in MVP the Presenter would tell the View “display this text in that specific text block”, in MVVM the ViewModel would update its “current text” property and let the View decide where to show it.

The View is a function of the ViewModel, i.e. the underlying MVVM system will compute and derive the view automatically from the current state of the ViewModel. While it can incrementally update itself as an optimization (e.g., Svelte), it can also recompute itself at any time. This is useful in scenarios where the View is thrown away when not in focus, such as mobile apps. Data flow may be unidirectional (React.js, Flutter, ...), i.e. the model state must be explicitly modified, and changes are automatically pushed down to the view, or bidirectional (Vue.js), i.e. the view also binds backwards to the ViewModel such that user interactions on the View affect the model state directly and automatically.

MVVM enforces a clean layering: the View observes the ViewModel, but the ViewModel is not aware of the View. Similarly, the ViewModel uses the Model, but the Model is not aware of the ViewModel.

In practice, MVVM is used in many modern frameworks: Windows with the “Universal Windows Platform”, Android (Jetpack Compose), React.js and Vue.js on the Web…

System design

Last week, we discussed ways to write clean code, build programs with reusable module design techniques such as design patterns, and how to organize your modules across your entire codebase using architectural patterns. In this lecture, you will learn about system design principles. They structure and dictate the entire ecosystem around (and including) your software and define the behavior and the implementation of the entire environment.

Layering

We have seen in the modularity and abstraction lecture that one of the symptoms of complexity was a high number of module interconnections. We would like to reduce direct dependencies between modules, however without relaxing the strict isolation provided by module boundaries or losing expressive power. One way to achieve this is to assign roles to modules, and arrange them into a hierarchy with additional rules.

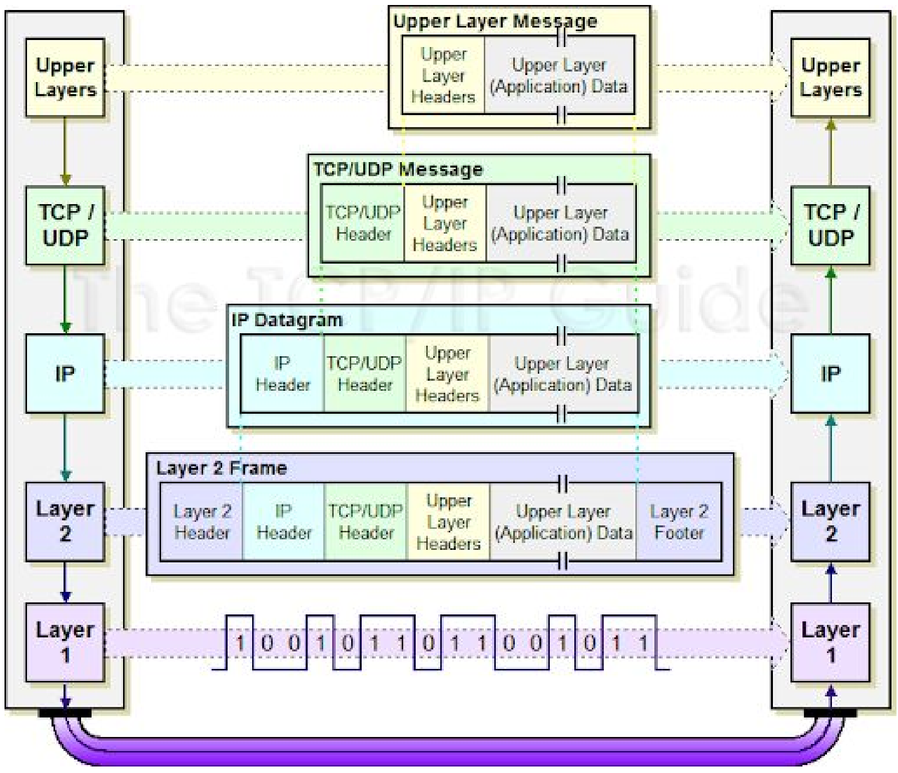
Layering is a great example of module organization, used extensively in network and computer system design among others. A layered system organizes modules into a stack of layers, such that each layer is composed of one or several related modules. Each layer has exactly one interface above and one below, except the top and bottom layers.

A 3-layer stack system with 3 components per layer

The bottom layer implements the base set of constructs: they usually are low-level primitives that provide the raw capabilities of the system. Each additional layer then implements additional, richer features on top of the existing stack. The topmost layer thus provides end-user functionalities and relies on all layers below (note that lower layers must guarantee that they perform as expected by their interface, otherwise the entire stack will fail).

Several instances of a stack architecture may be spawned (possibly in a distributed fashion). On the same stack instance, a layer may only communicate with layers above and below through an **API**. A layer component can only communicate with a different stack instance at the same layer level, through a **protocol**. When a data representation must be manipulated by several layers of a stack, each layer will only write to its corresponding **header** using **predefined semantics**. Headers encapsulate the data body (which usually belongs to the topmost end-user layer), in the inverse order of their corresponding layer.

One well-known example of a layered system is the TCP/IP networking stack: each connected machine runs an instance of the networking stack. The bottom layer is the physical layer, which enables bits transfers over some medium (copper, wireless, fiber, ...). The link layer provides connectivity between 2 devices on the same local network by implementing the grouping of bits into frames, as well as communication protocols such as Ethernet. The Internet layer provides addressing (IP), host-to-host communication (across connected local networks), variable length data sequences as well as routing protocols. The transport layer provides control and guarantees over data rate and quality: for instance TCP provides reliability, in-order delivery, congestion control, error detection while UDP provides performance without additional guarantees. The application layer implements many protocols for end-users such as web content transfer (HTTP), email retrieval (IMAP) or secure communications (TLS/SSL, SSH).



Example: 2 instances of the TCP/IP network stack running on 2 machines connected through a physical link; a single application data body is sent and encapsulated by a header at each layer

Each layer will also provide an abstraction layer: each layer does not see the inner workings of other layers, which allows separation of concerns. This provides several advantages:

- **Interoperability:** all systems that adhere to the stack protocols will work together as long as their interfaces do not change. This includes interoperability through time (e.g., 56kbps modem vs fiber optic) and platform independence (e.g. ARM smartphone through WiFi vs x86 desktop computer through LAN)
- **Independence:** each part of the system can be developed independently, each at a different pace, by a dedicated group of organizations or people (e.g. network switch manufacturers vs OpenSSL developers)
- **Specialization**: multiple implementations of upper layers can co-exist and each variant can be specialized to achieve some specific requirement (e.g. TCP reliability vs UDP performance, HTTP / FTP / SSH / ... application protocols)

Another use case of layering is web application development: usually a web service is not designed as a single monolithic software, but rather as several components living on separate software systems. This example depicts a 3-tier architecture: the goal is to detach the presentation, application processing and data management functionalities physically, such that different people with distinct jobs can work on each part of the system using different tools. The physical support of each tier is also usually different, for instance, the web server software that handles incoming client requests lives on a different (virtual) machine than the database servers.

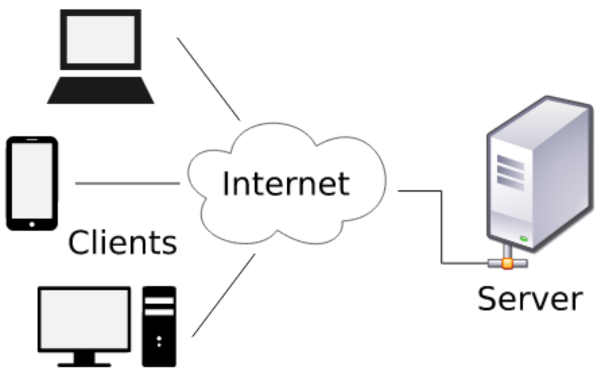
Let us take the example of the web services provided by a university to its students and staff. At the topmost tier, which lives in the clients' browsers, users can interact with the system (students getting their grades, staff validating their monthly payrolls, ...). The user interface and prestation layers are developed by UI/UX designers and developers using tools such as ReactJS, (S)CSS, Photoshop. The application and domain model layers handle requests coming from the user interface layer through a web server, as well as data representation and business logic. They are maintained by software engineers and school administrators with example tools Java Spring, Excel and domain-specific code. The persistence and data layers provide data storage and hardware management by database engineers, IT workers with example tools MySQL DB, physical servers or third-party services such as Amazon S3 Cloud Storage. A full-stack engineer is thus a developer which can handle all aspects in each layer of the n-tier architecture.

A layering violation occurs when a module breaks the communication rules, or when a module makes assumptions about the operation of another module that belongs to a different layer. Sometimes, such a violation is necessary to overcome some other limitation: for instance Network Address Translation (NAT) is a hack around TCP/IP layering. To overcome the shortage of IPv4 addresses, a router may assign non-globally unique addresses inside its subnet and expose a single common IPv4 address for all machines inside to the external world. To remember which replies belong to which request, the router must keep a translation table in which it uses and transforms transport-level port numbers to identify the original sender. Thus, host-to-host IP communication is **impossible** with NAT without cheating and peeking at the UDP/TCP port number, which is a transport layer detail. This entails additional problems, as transport ports usually carry their own semantics.

When designing a layered system, the core question is how to pick the layers? There is no universal answer to this, but it boils down to domain-specific constraints and a balance between cognitive load (i.e. the amount of features that a layer must handle) and performance overhead (caused by the implementation of each layer, and by layers crossing). Ideally, natural boundaries exist and define the layers architecture, or experienced developers / domain experts can decide from past know-how. In general, one should pick the finest layering that makes sense and that provides the required performance.

RESTful systems

Another powerful system architecture is the client / server model which splits the system in 2 independent parts: the servers are providers of resources (such as files, raw data, computations, ...) or services (printers, game servers, media streaming, ...). The servers wait for clients to perform requests to gain access to the resources / services; many clients may connect to each single server. Clients and servers communicate over a network.



RESTful systems are a subset of client / server web services with additional properties and restrictions. They are largely used to facilitate automated communication between a broad spectrum of computing devices. In RESTful services, clients request textual representations (typically JSON or XML) of a server resources, and can retrieve and manipulate them (possibly modify them as well) using predefined stateless operations. REST stands for REpresentational State Transfer.

Server resources can be documents, files, but also any entity or action that can be identified with a name (for instance, the GitHub REST API allows you to create a repository). They are identified by an URI (often an URL) which is not necessarily the concrete location of the resource on the server, but rather an abstract, hierarchical naming scheme. HTTP(S) verbs are typically used to perform actions: GET (obtain resource), POST (create), PUT (update / replace), PATCH (partial update / replace), DELETE (remove). A client can thus specialize a request using a URL (to access a particular resource directly), a verb (to describe what action to perform on the resource), and method parameters (for client-provided data input, for instance a search keyword).

Example: retrieving Wikipedia pages which contain a location close to GPS coordinates 37.7891838, -122.4033522

API URL (GET request):

- The resource is https://en.wikipedia.org/w/api.php which is the script that handles any API request on this server (the exact URL mapping semantics are left to the developer: for instance, WikiMedia could have decided to implement geosearch at a specific URL such as https://en.wikipedia.org/w/api/geosearch).
- The verb is GET (default when entering an URL in a web browser) to retrieve results
- The parameters are action = query (obtain a search result), list = geosearch (define the type of search), gscoord = 37.7891838|-122.4033522 (coordinates), gsradius = 10000 (search radius) and gslimit=100 (max number of results). The format = json parameter is specified to receive JSON instead of HTML.

https://en.wikipedia.org/w/api.php?action=query&list=geosearch&gscoord=37.7891838%7C-122.4033522&gsradius=10000&gslimit=100&format=json

Server response:

```
{
    "batchcomplete": "",
    "query": {
        "geosearch": [
            {
                "pageid": 18618509,
                "ns": 0,
                "title": "Wikimedia Foundation",
                "lat": 37.78918,
                "lon": -122.40335,
                "dist": 0.5,
                "primary": ""
            },
            {
                "pageid": 42936625,
                "ns": 0,
                "title": "Foxcroft Building",
                "lat": 37.78916666666667,
                "lon": -122.40333333333334,
                "dist": 2.5,
                "primary": ""
            },
            {
                "pageid": 9293658,
                "ns": 0,
                "title": "One Montgomery Tower",
                "lat": 37.7891,
                "lon": -122.4033,
                "dist": 10.4,
                "primary": ""
            },
            {
                "pageid": 2608926,
                "ns": 0,
                "title": "San Francisco Mechanics' Institute",
                "lat": 37.788844,
                "lon": -122.403042,
                "dist": 46.6,
                "primary": ""
            },
            // … truncated
        ]
    }
}
```

Using structured text instead of other representations (binary, custom format, …) makes it simple for computer systems to parse, but also for humans to read, while providing a uniform interface that is compatible with virtually any device. Debugging is also straightforward, as the communication is isolated, visible and explicit.

Importantly, not all HTTP systems or API systems are RESTful. A RESTful system must comply with the following constraints:

- **Client-server architecture**

  As defined previously, which enables portability (many different clients can connect to the REST API server), scalability (servers can be extremely lean, as they do not need to handle client-side concerns), security (as client and server data / logic are separated) and independence of the 2 parts of the system (allowing them to be developed at a different rate, with different technologies).

- **Statelessness**

  No client context must be stored on the server in-between requests: each request is self-contained, atomic, and has all the necessary information for the server to service it. We call client state (or context) “session”, which must be kept on the client side. Temporary session state is allowed on the server, as long as it is transferred to another service (for instance, authentication tokens can be temporarily saved in a database). For instance, a client may attempt to fetch a restricted resource: a RESTful system would require a token to be sent by the client to service the request successfully, however a RESTful system cannot simply remember (in memory) that a given client has visited the login resource beforehand. In other words, the response sent by the server cannot depend on previous interactions with the client, i.e. the server should not remember what it has seen. Similarly, two clients sending the exact same request will receive the same response. This property is one the most important of RESTful systems, and sometimes forgotten on some APIs that are wrongfully labelled as REST.

- **Cacheability**

  Responses from the server must implicitly or explicitly declare themselves as either cacheable or non-cacheable. A response marked as cacheable can be reused by the client (or retransmitted by intermediaries such as proxies) for later identical requests. This improves scalability and performance, as some requests can be immediately serviced without connecting to the server again.

- **Layered system**

  From the client point of view, a requester cannot tell whether they are interacting with the server directly, or with an intermediate layer such as a proxy, load balancer or firewall. The server can also perform subroutines on other servers before gathering the results and forming a response. Thus, the server can be composed of a hierarchy of layered components (which can be physically distinct), however the interface exposed to the client is identical at any level of the stack.

- **Code on demand (optional)**

  Servers can send executable code to clients to augment their features: for instance, web browsers may obtain script resources from the server such as Javascript, Java Applets or WASM binaries, which they can execute in a sandboxed environment. For instance, they can allow a client to animate their obtained views (HTML with Javascript), or access low-level computations with content provided by the server (such as GPU rendering for web games using WASM).

- **Uniform interface**

  The interface consists of the server-side resource identifiers (URIs), the textual format sent to the client (MIME type e.g. JSON, XML, YAML, ...) as well as their structure (tree hierarchy of URIs, shape of key-values responses). The interface should be as generic and explicit as possible, while maintaining a consistency across all possible values. To this purpose, 4 additional constraints are imposed:

  - **Resource identification in requests**

    Each resource is accessible through a unique identifier (for instance a specific URL). The actual location of the resource on the server is decoupled: for instance a file resource may have a completely different path on the server filesystem. Resources may not be materialized when the request is performed (e.g. computations, actions or real-time server state). The textual representation is also separate from the actual underlying representation: for instance a server may store data in binary inside a database, however the returned values will be HTML views, or structured key-values in JSON or XML.

  - **Resource manipulation through representations**

    The textual representation of a resource, including its metadata (such as creation date, encoding, ...) holds all the necessary information for a client to perform subsequent manipulations on the resource (for instance modifying it through a PUT request, or removing it with a DELETE).

- **Self-descriptive messages**

  Each message is self-contained and contains all necessary information to process it (e.g. MIME type metadata, correct syntax)

- **Hypermedia as the engine of application state (HATEOAS)**

  Starting from a single URI, a client can discover all other resources on the server through hyperlinks. The body of responses may contain URIs to additional resources, and all URIs thus appear at least on one response body. The service thus can use itself to discover itself.

Example: HATEOAS on an hypothetical SwEng API

Assume there exists an API endpoint to list all staff members of SwEng, i.e. a request to `sweng.epfl.ch/staff/list` returns the following answer:

```
      {
         "staff": [
             {
                 "name": "George Candea",
                 "uri": "sweng.epfl.ch/staff/person/1"
             },
             {
                 "name": "Solal Pirelli",
                 "uri": "sweng.epfl.ch/staff/person/2"
             },
             {
                 "name": "Samuel Chassot",
                 "uri": "sweng.epfl.ch/staff/person/3"
             },
             {
                 "name": "Alexandre Chau",
                 "uri": "sweng.epfl.ch/staff/person/4"
             },
             // ... other staff members
         ]
}
```

A client can parse the result, and access the details of each individual staff member through the provided “uri” field of each element in the list. It can then send additional requests to “sweng.epfl.ch/staff/person/1”, “sweng.epfl.ch/staff/person/2”, … to obtain the complete representation of all staff members. Each person may also contain additional URIs. The client used the URIs provided by the system to access additional resources, i.e. it used HATEOAS.

Reactive programming

Up until now, most code you have seen and studied executes in the order in which it is written (at least from your point of view as a programmer, see out-of-order processor execution). Put simply, reactive programming encompasses any declarative paradigm in which code is not processed in-order of lines, but rather as defined by “reacting” to events, i.e. parts of the code (usually functions) are ran only when an event (which semantics can be arbitrarily defined by a language, framework or programmer) occurs. One example that you already know is stream programming: when processing a stream using map for instance, the iteration on the collection is invisible to the programmer, however the argument function is applied to each element individually.

Reactive programming may not be intuitive to programmers that are used to imperative constructs. Here's a toy example in Node Javascript, try to answer to the following questions:

- In which order do the `console.log` statements at lines 10, 15, 20 and 25 print ?
- How many times are the `console.log` statements printing something ?
- In which code locations can you have a guarantee about the value of the counter that is passed around?

```
1. let emitter = new EventEmitter()
2. let counter = 0
3.
4. setInterval(() => {
5.    emitter.emit("t", counter)
6.    counter = counter + 1
7. }, 1000)
8.
9. emitter.on("b", (value) => {
10.   console.log("Print 1: " + value)
11.   emitter.emit("x", value)
12.})
13.
14.emitter.on("x", (value) => {
15.   console.log("Print 2: " + value)
16.   emitter.emit("k", value)
17.})
18.
19.emitter.on("t", (value) => {
20.   console.log("Print 3: " + value)
21.   emitter.emit("b", value)
22.})
23.
24.emitter.on("k", (value) => {
25.   console.log("Print 4: " + value)
26.})
```

Answers: (highlight the text between each pair of brackets)

- [ ]
- [ ]
- [ ]

As you can see with the above example, the code clearly does not run in the order of the lines in which it is written, but rather jumps from closure to closure following the "path" of the event produced by the emitter. One may wonder how such code would be useful: it all makes sense in the context of applications that rely heavily on network or live data. Indeed, given the unpredictability of network message (packet losses, delays, latencies) and the nature of real-time data sources (such as real-life sensors, stock market values, audio signals), a paradigm in which code execution does not depend on the availability of data sources is desirable.

A synchronous approach would suffer in performance: a program would have to actively wait for the data to become accessible. Imagine that an application is drawing a user interface and must also fetch some data on a server, the code may look like the following:

```
data = fetch("example.com/my/data")
view = render(template, data)
```

```
userInput = pollButtons()
```

The program would be blocked at the first line of code, until the request is sent, processed by the distant server, sent back to the client and loaded into the program memory (this could take indefinitely long given the data size and the network conditions!). During all that time, the view is frozen and the user is unable to interact with the program. This is clearly not good enough for human reactivity, and it will quickly frustrate the user. Another approach would consist in continuously verifying if the data is available (assuming the fetch operation can be detached in another thread for instance and is thus now non-blocking), and incrementally updating the rest of the system (the UI in this instance, assuming render can create displays even with missing or partial data):

```
data = fetch("example.com/my/data") // this is now non-blocking
done = false
do {
  done = data.ready()
  view = render(template, data) // even if data is not ready of partial
  userInput = pollButtons()
} while (!done)
```

This is a technique called “polling” and can be used to implement pull data propagation in reactive programming systems. However, this implementation can be quite wasteful on resources, such as CPU usage, thus performance would still suffer. Instead of actively waiting on data, we would like to design programs in such a way that the control flow is reversed: events should trigger computations, and propagate them to dependent operations. Ideally, the reactive system is able to perform other tasks while some long computation is performed in the background. The first code example should now make more sense to you.

In essence, reactive programming systems will build a computing graph in which nodes model computations and edges data flow, i.e. a path from a result to a dependent computation. A node may either use polling on preceding nodes to obtain new values (consumers *pull* producers), or notify neighboring nodes when its computation is finished (producers *push to* consumers). A third variant called *push-pull* exists, where nodes push a *notification change* to neighbors, which in turn will then pull the actual value from the sender (this is useful when data values have a significant memory size for instance).

The implementation of the graph and its execution vary by platform and usage; common designs include event loops with queues that store event requests and dispatch workloads across time slices, thread pools or coroutines. Functional programming is often used to declaratively program reactive systems, as computations can be implemented as callbacks (or derivatives such as promises / async-await).

Some reactive techniques will be covered more extensively later in the course, in the lecture about asynchrony.

Defensive programming

Please watch the video lecture in lieu of written notes: part 1 and part 2.