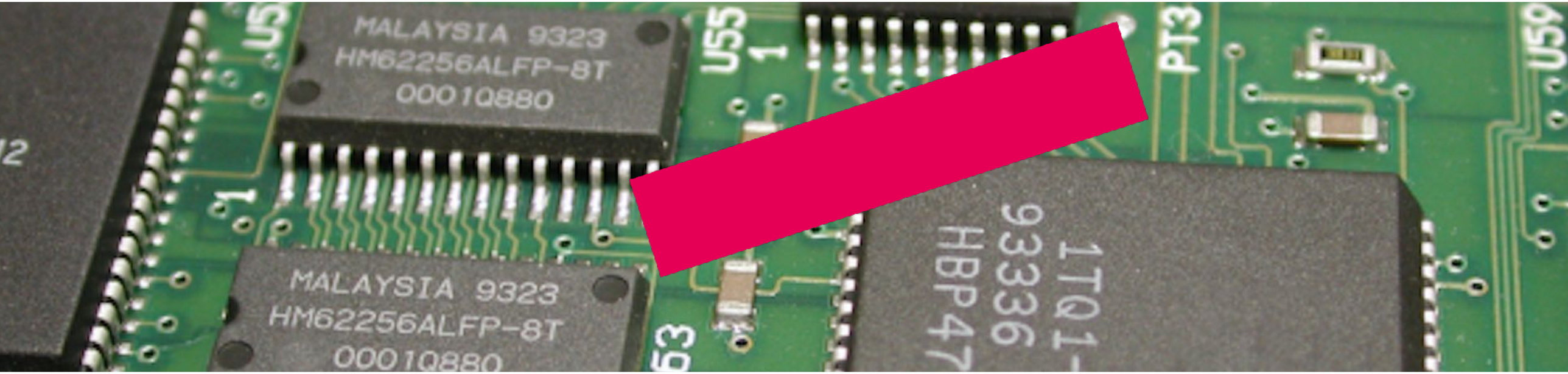
Prog 5 Embedded Systems Development - 5. Concurrency and more



Embedded Systems Engineering
School of Engineering and Automotive

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

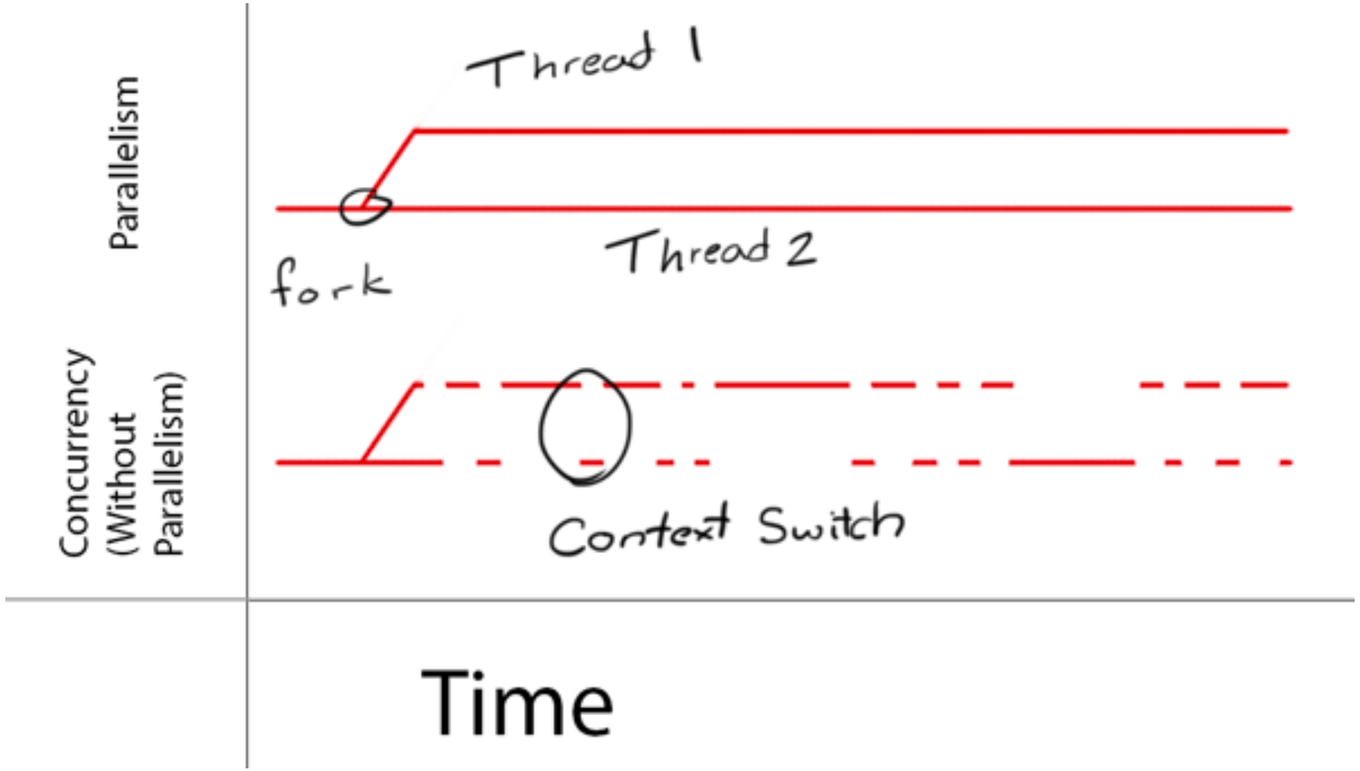
To begin with…

Attendance

Schedule (exact info see #OO and roster at insite.han.nl)

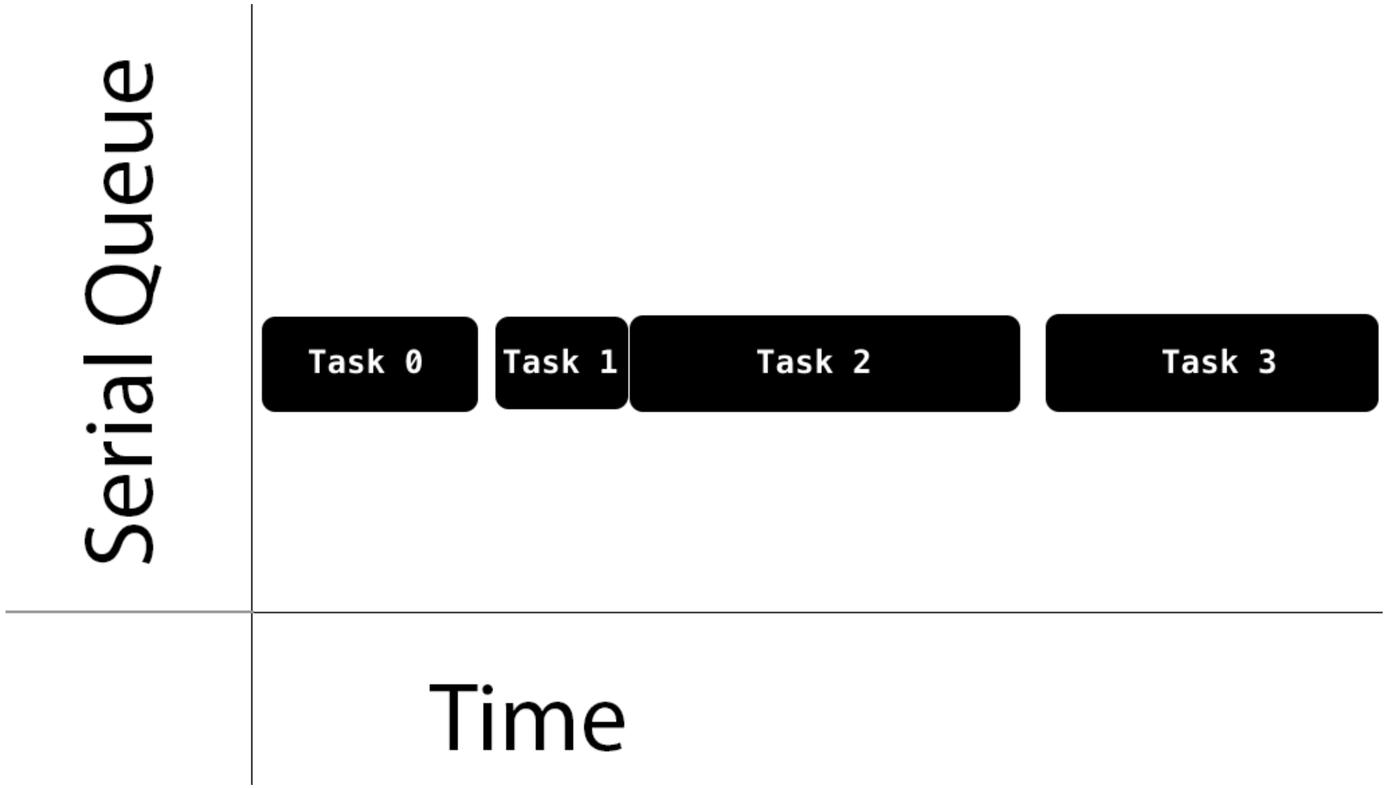
		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	Class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	Liskov Substitution Principle	lists, inline functions, default params	Activities
Step 4	Interface Segregation Principle	interfaces and abstract classes	Depencencies
Step 5	**Dependency Inversion Principle**	**threads, callback**	**Sequence diagrams**
Step 6	Coupling and cohesion	polymorphism	Composition, packages
Step 7	SOLID in Embedded Context	Wrap-up	Use cases

Concurrency



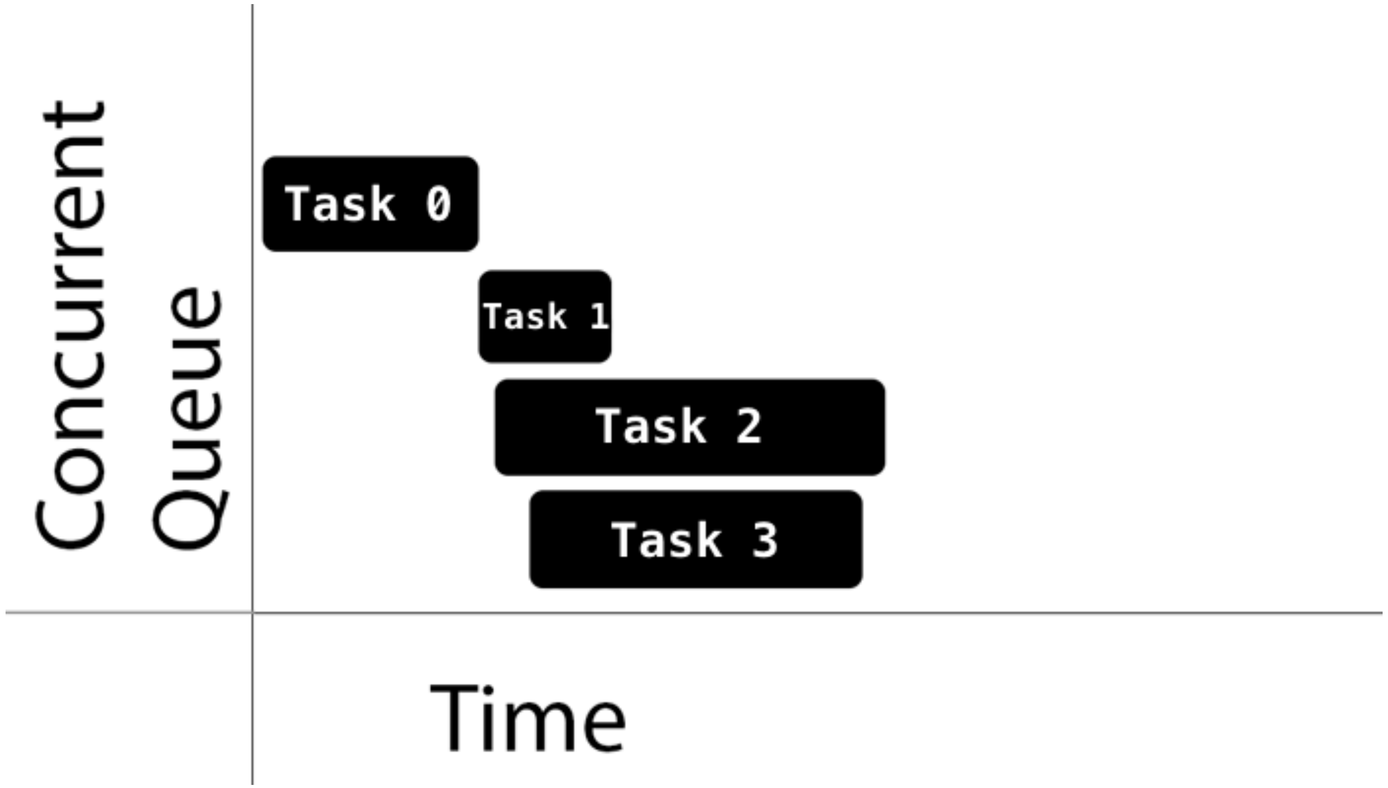
Source: Ray Wenderlich

Queues: Serial Queue



Source: Ray Wenderlich

Queues: Concurrent Queue



Source: Ray Wenderlich

C++ Language: Threads (C++11 and higher)

```
#include <thread>
```

C++ Language: Threads (C++11 and higher): Callable function

```
 #include <thread>


// function to be used in callable


void callable_func(int N)
 {
   for (int i = 0; i < N; i++) {
        cout << "Thread 1 :: callable => function pointer\n";
   }
 }
```

C++ Language: Threads (C++11 and higher): Callable Object

```
 #include <thread>


// A callable object
class thread_obj {
 public:
   void operator()(int n) {
       for (int i = 0; i < n; i++){
           cout << "Thread 2 :: callable => function object\n”;


   }
};
```

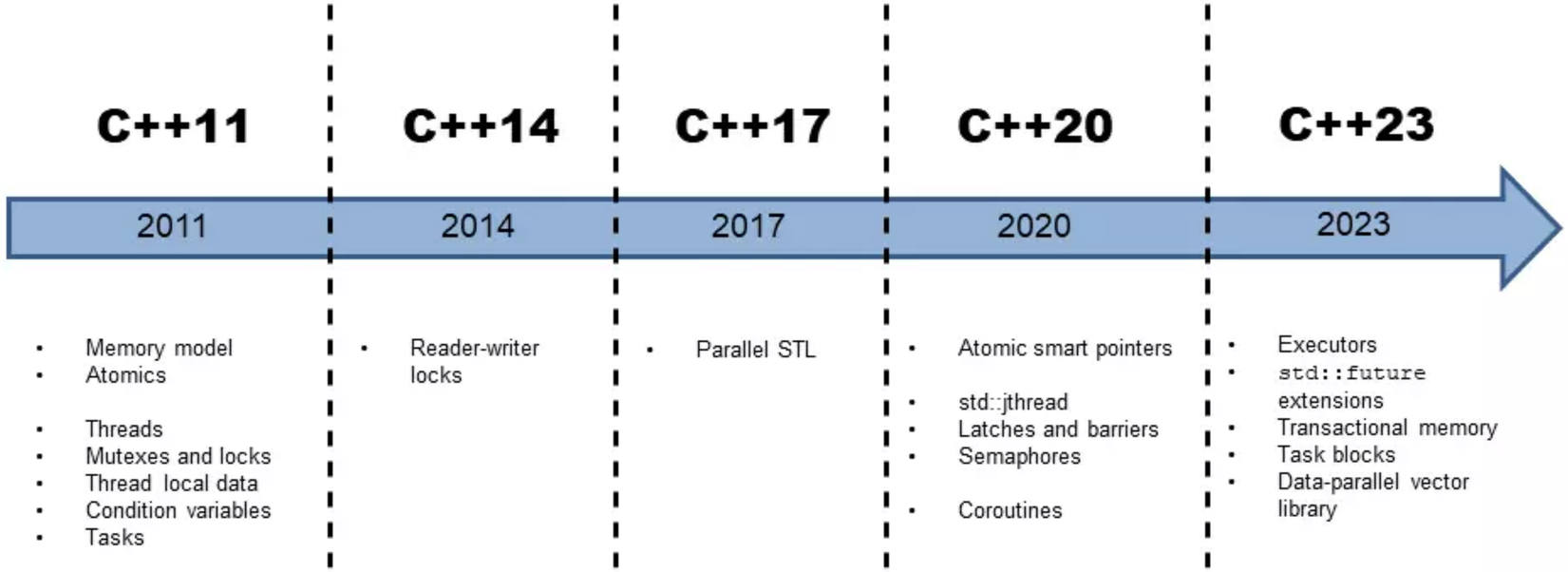
C++ Language: Threads (C++11 and higher): Lambda Expression

```
#include <thread>


// Define a Lambda Expression
    auto f = [](int n) {
        for (int i = 0; i < n; i++) {
                cout << "Thread 3 :: callable => lambda expression\n";
        }
    };
```

C++ Language: Threads (C++11 and higher): Lambda Expression

```cpp
#include <thread>

int main()
{
    // ... e.g. Lamda expression

    //launch thread using function pointer as callable
    thread th1(collable_func, 2);

    // launch thread using function object as callable
    thread th2(thread_obj(), 2);

    //launch thread using lambda expression as callable
    thread th3(f, 2);

    // Wait for thread t1 to finish
    th1.join();
    // Wait for thread t2 to finish
    th2.join();

    // Wait for thread t3 to finish
    th3.join();

    return 0;
}
```

C++ Language: 'Concurrency Evolution'



https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-concurrency-and-parallelism/

https://www.sitepoint.com/premium/books/concurrency-with-modern-c/read/1/

Typical Concurrency Problems / Solutions

Problems:

- Resource contention (dining philosophers) / Race Conditions (e.g. data race)
- Starvation and Deadlock (ultimate form)
- Flight booking system (locking)

https://youtu.be/NbwbQQB7xNQ

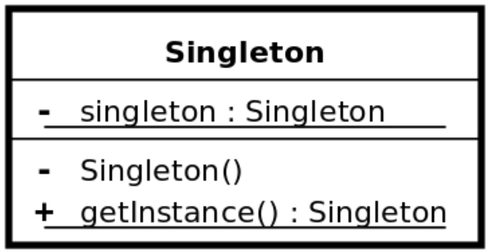
Solutions:

- Atomic operations
- Locks
- Mutual Exclusives (e.g. semaphores)
- Priority queues

C++ Language Pattern: Singleton

A *singleton pattern* is a software design pattern that restricts the instantiation of a class to one "single" instance.

This is useful when exactly one object is needed to coordinate actions across the system.



C++ Language: Callback methods Blocking

```cpp
#include <iostream>
#include <functional>

void processData(int data, std::function<void(int)> callback) {
    // Do some processing on the data
    int processedData = data * 2;

    // Invoke the callback function with the processed data
    callback(processedData);
}

int main() {
    // Define a lambda function to be used as a callback
    auto printResult = [](int result) {
        std::cout << "Processed data: " << result << std::endl;
    };

    // Pass the lambda as a callback to the processData function
    processData(10, printResult);

    return 0;
}
```

Tip: use breakpoints to check the actual flow!

C++ Language: Callback methods Non-Blocking

```cpp
#include <iostream>
#include <functional>
#include <future>  // For async

std::future<void> processDataAsync(int data, std::function<void(int)> callback) {
    // Use std::async to launch asynchronous task and return a future
    return std::async( policy: std::launch::async,  f: [data, callback]() -> void {
        // Simulate some asynchronous processing with a delay
        std::this_thread::sleep_for( d: std::chrono::seconds( r: 10));

        // Process the data
        int processedData = data * 2;

        // Call the callback function with processed data
        callback(processedData);
    });
}
```

Tip: use breakpoints to check the actual flow!

C++ Language: Callback methods Non-Blocking

```cpp
int main() {
    // Define a lambda function to be used as a callback
    auto printResult : void (int) const = [](int result) -> void {
        std::cout << "Processed data: " << result << std::endl;
    };

    // Call the asynchronous version of processData and get the future
    std::cout << "Starting asynchronous processing..." << std::endl;
    std::future<void> asyncTask = processDataAsync( data: 20, callback: printResult);

    // Do other things while waiting
    std::cout << "Main thread is free to do other work..." << std::endl;

    // Wait for the async task to complete (if necessary)
    asyncTask.wait();  // This waits for the asynchronous task to finish

    std::cout << "Asynchronous task finished." << std::endl;
    return 0;
}
```

Tip: use breakpoints to check the actual flow!

C++ Language: Callback methods Non-Blocking

```
int main() {
    // Define a lambda function to be used as a callback
    auto printResult : void (int) const = [](int result) -> void {
        std::cout << "Processed data: " << result << std::endl;
    };

    // Call the asynchronous version of processData and get the future
/Users/jakorten/Development/CallbacksCPP_Nonblock/cmake-build-debug/CallbacksCPP_Nonblock
Starting asynchronous processing...
Main thread is free to do other work...
Processed data: 40
Asynchronous task finished.

Process finished with exit code 0
    asyncTask.wait();  // This waits for the asynchronous task to finish

    std::cout << "Asynchronous task finished." << std::endl;
    return 0;
}
```

Output of our code…

Our code will work in C++11, C++14, C++17, C++20, and even C++23 without any modifications because it relies on core features (std::async, std::future, and lambdas) that were introduced in C++11 and have been stable across subsequent standards.

C++ Language: Async Callback

std::async() does following things:

- It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.

- Then passes the std::promise object to thread function and returns the associated std::future object.

- When our passed argument function exits then its value will be set in this promise object, so eventually return value will be available in std::future object.

See *async_callback.cpp* and *async_single.cpp* examples.

https://thispointer.com/c11-multithreading-part-9-stdasync-tutorial-example/

Revisiting SOLID

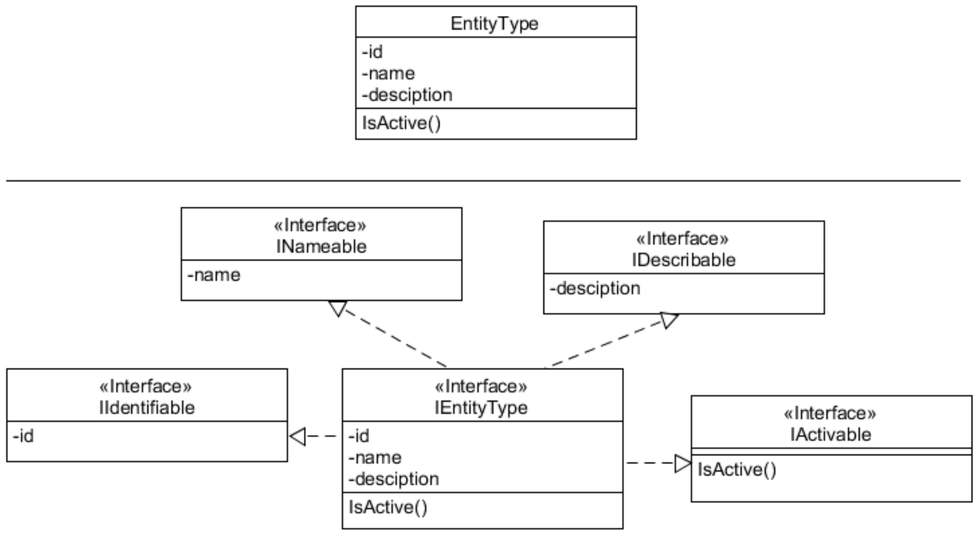
SOLID: Interface Segregation

“No client should be forced to depend on methods it does not use.”

- Uncle Bob

https://claudiorivera.net/2018/02/04/isp-interface-segregation-principle/

SOLID: Interface Segregation



https://claudiorivera.net/2018/02/04/isp-interface-segregation-principle/

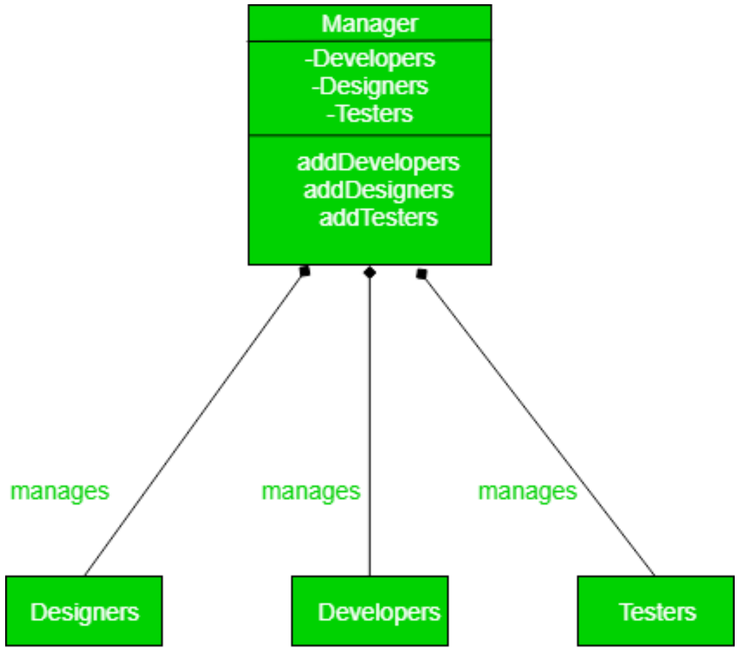
SOLID: Dependency Inversion Principle

“High-level modules should not depend on low-level modules.
Both should depend on abstractions.”
“Abstractions should not depend on details.
Details should depend on abstractions.”
—R. Martin

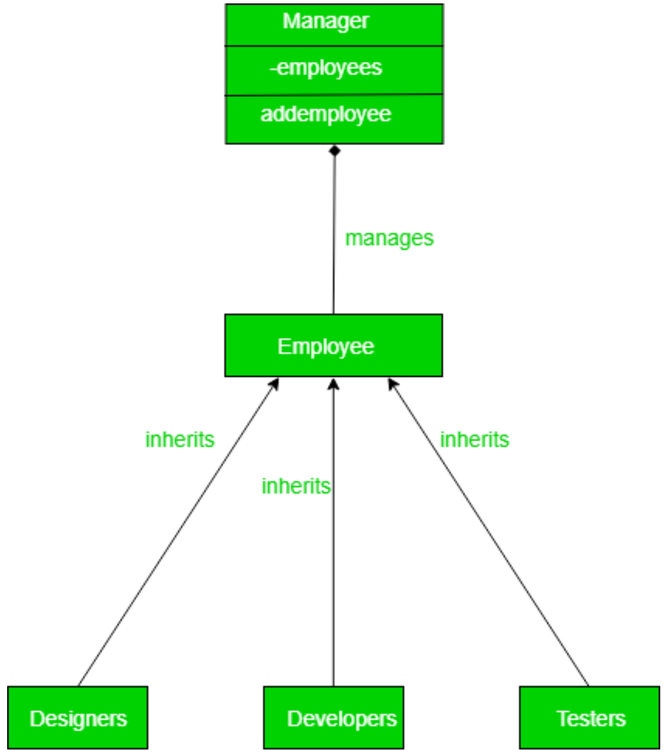
https://claudiorivera.net/2018/02/12/dip-dependency-inversion-principle/

SOLID: Dependency Inversion Principle

From:



To:



https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

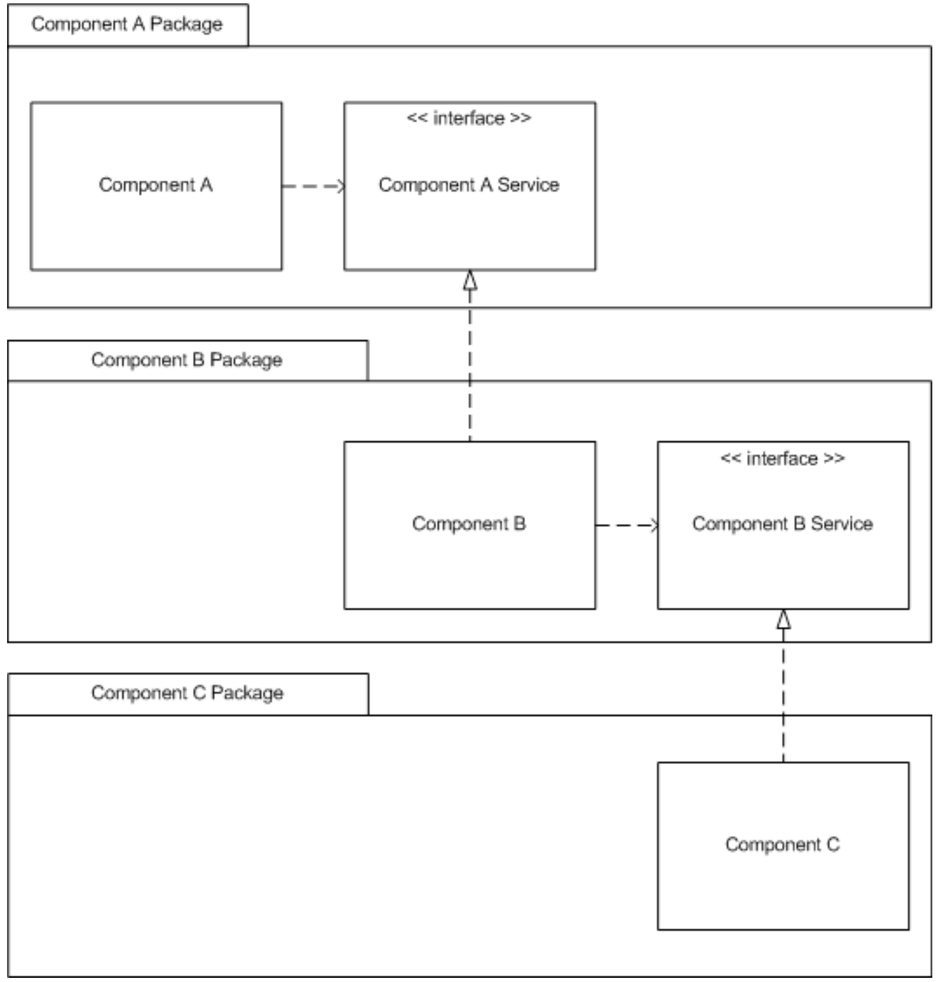
SOLID: Dependency Inversion Principle

Helps to sustain low-coupling between the components comprising an application.

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

https://dzone.com/articles/the-dependency-inversion-principle-for-beginners



SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Back to our challenge on how to make Coffee…

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

How to make Coffee?!

a. Put:
- Water
- A filter
- Grounded coffee

In a coffee maker (seen at the right).

b. Turn machine on…
c. Wait till it is brewed
d. Enjoy

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Design for change! How to make Coffee for different machines?

a. Put:
- Water
- A filter
- Grounded coffee

A coffee pad…

In a coffee maker (seen at the right).

b. Turn machine on…
-> press left or right button (one or two cups)
c. Wait till it is brewed
d. Enjoy

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Design for change! How to make Coffee for different machines?

a. Put:
- Water
- A filter
- Grounded coffee A coffee pad… Nespresso cup

In a coffee maker (seen at the right).

b. Turn machine on…
-> press left or right button (one or two cups)
Uh, which one???
c. Wait till it is brewed
d. Enjoy

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Design for change! How to make Coffee for different machines?

a. Put:
- Water
- A filter
- Grounded coffee A coffee pad… Nespresso cup Beans!

In a coffee maker (seen at the right).

b. Turn machine on…
-> press left or right button (one or two cups)
Uh, even more complicated???
c. Wait till it is brewed
d. Enjoy

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Design for change! How to make Coffee for different machines?

How would we manage this?
- At coffee level?
- At machine level?

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov / Interface Segregation / Dependency Inversion: Coffee

Design for change! How to make Coffee for different machines?

How would we manage this?
- At coffee level?
- At machine level?

Adding interface for the machine.

Possible also at coffee level.

https://stackify.com/solid-design-liskov-substitution-principle/

SOLID: Liskov Substitution Principle

Takeaway Points:

- Subtypes must be substitutable for their base types without altering the correctness of the program.

- Derived classes should extend the behavior of base classes without changing their originally intended function.

- Violating this principle can lead to problems with class hierarchies and inheritance, where derived classes are not true extensions of their base classes.

SOLID: Interface Segregation Principle

Takeaway Points:

- Clients should not be forced to depend on interfaces they do not use.
- It’s better to have several specific interfaces rather than a one-size-fits-all "general purpose" interface.
- By segregating interfaces, you can ensure that a class only needs to know about the methods that are of interest to it.

SOLID: Dependency Inversion Principle

Takeaway Points:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

- By depending on abstractions rather than concrete implementations, you can create more modular and adaptable code.

SOLID: Main reasons

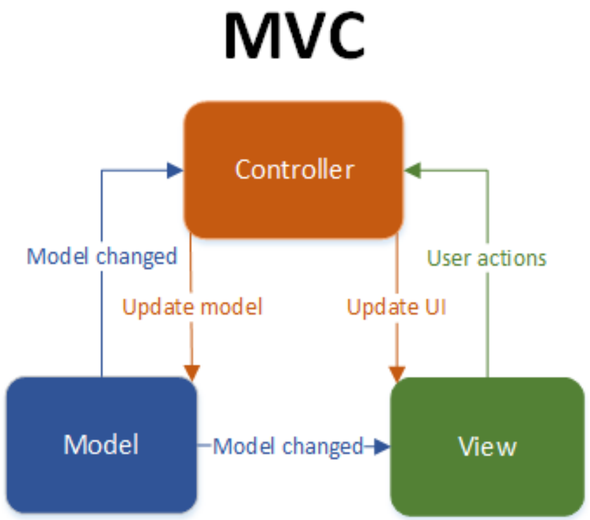
Takeaway Points:

- Following the SOLID principles:
- can greatly improve the quality of your object-oriented design and architecture.
- makes your code more robust, maintainable, and extensible
- leads to software that is easier to understand, modify, and enhance.

So: Learn to use it as much as possible!

We will talk later about: applying solid in an Embedded Context

Patterns: MVC



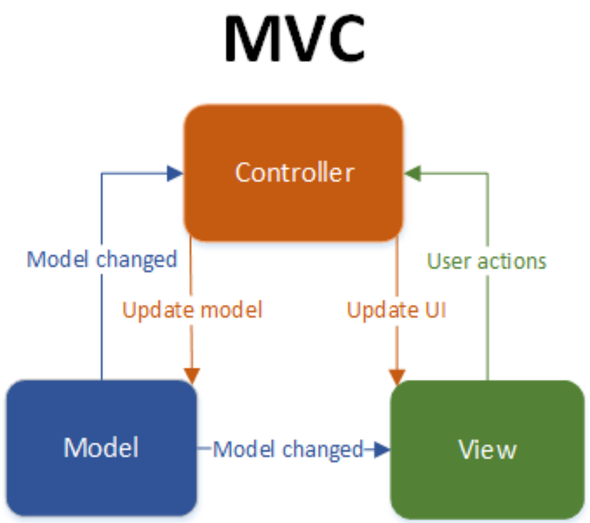
The *model* communicates with a source of data, providing an *interface* for the other components in the architecture.

The *view* obtains *model indexes* from the model; these are references to items of data. By supplying model indexes to the model, the view can retrieve items of data from the data source.

In standard views, a *delegate* renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes

https://doc.qt.io/qt-5/model-view-programming.html

Patterns: MVC



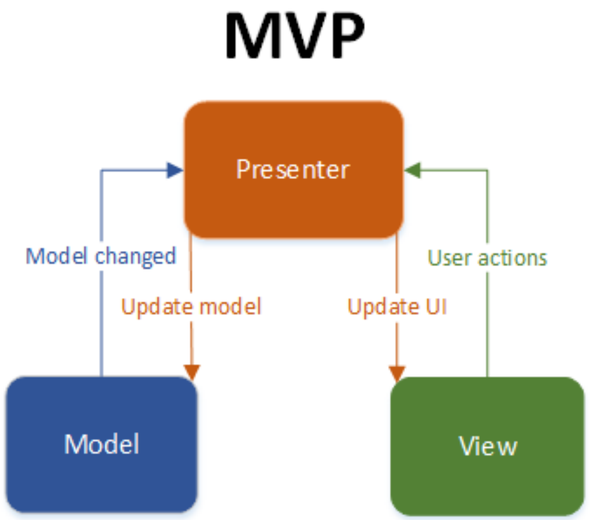
MVC consists of three kinds of objects:

- *Model* is the application object,
- *View* is its screen presentation,
- *Controller* defines the way the user interface reacts to user input.

Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

https://www.codeguru.com/cpp/cpp/implementing-an-mvc-model-with-the-qt-c-framework.html

Patterns: MVP

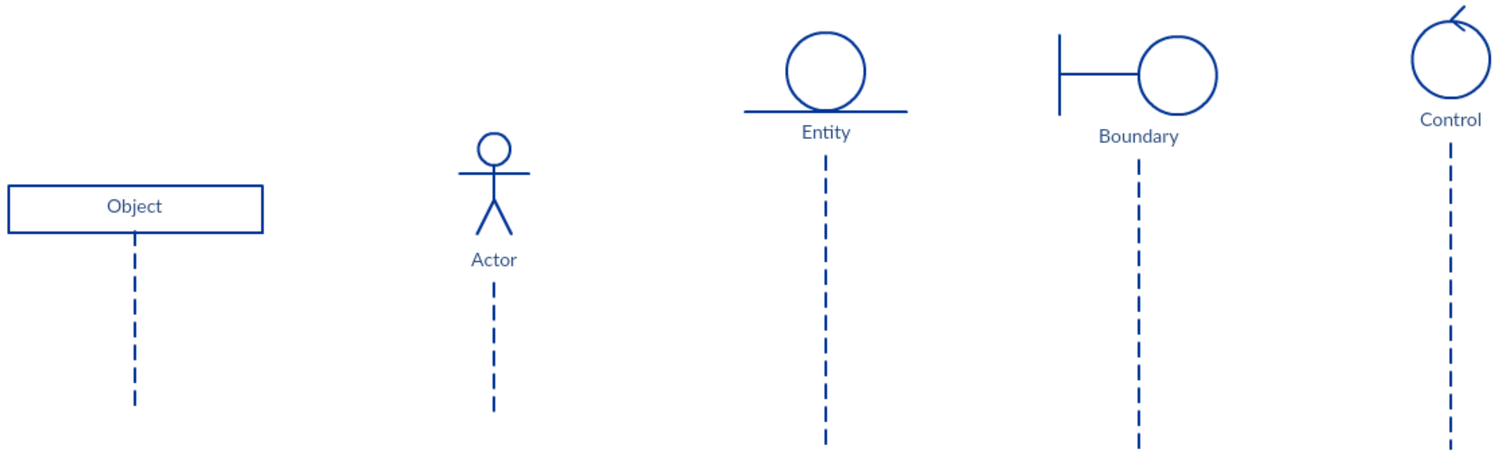


Popular alternative for MVC

Note: always consider alternative options when selecting Design Patterns. E.g. MVVM is also something alike.

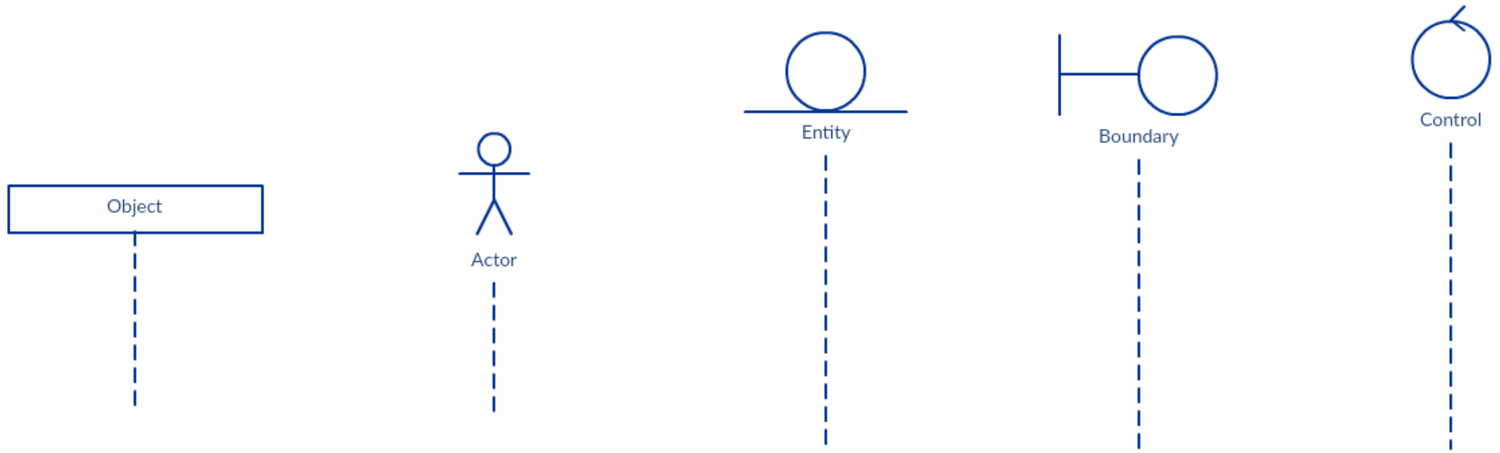
https://www.codeguru.com/cpp/cpp/implementing-an-mvc-model-with-the-qt-c-framework.html

UML: Sequence Diagram - Stereotypes



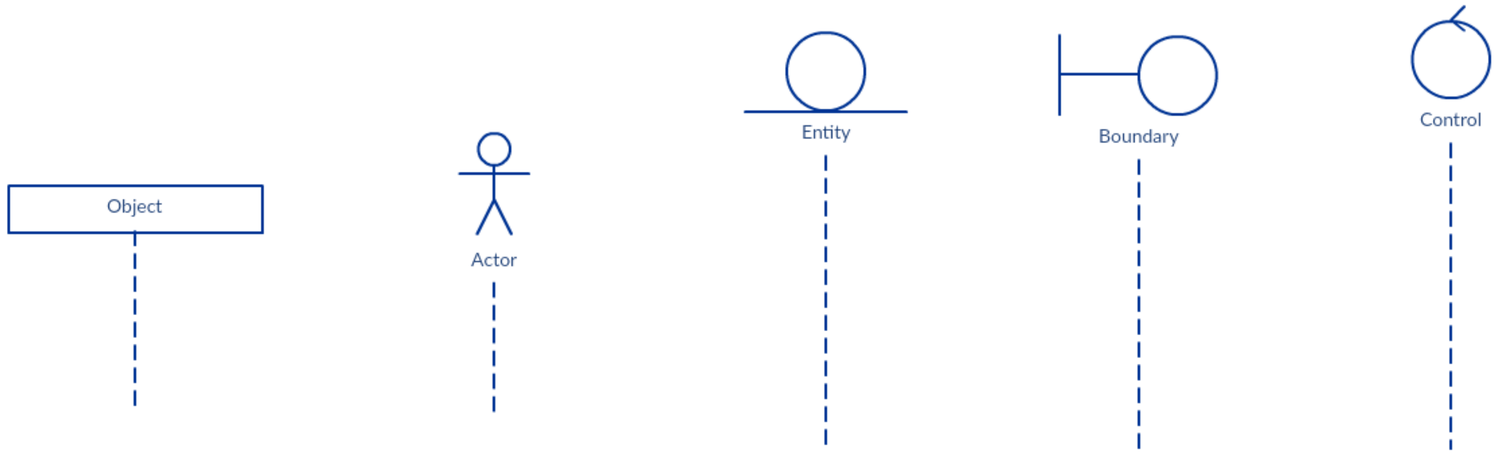
Typical **stereotypes** used in a Sequence Diagram.

UML: Sequence Diagram - Object



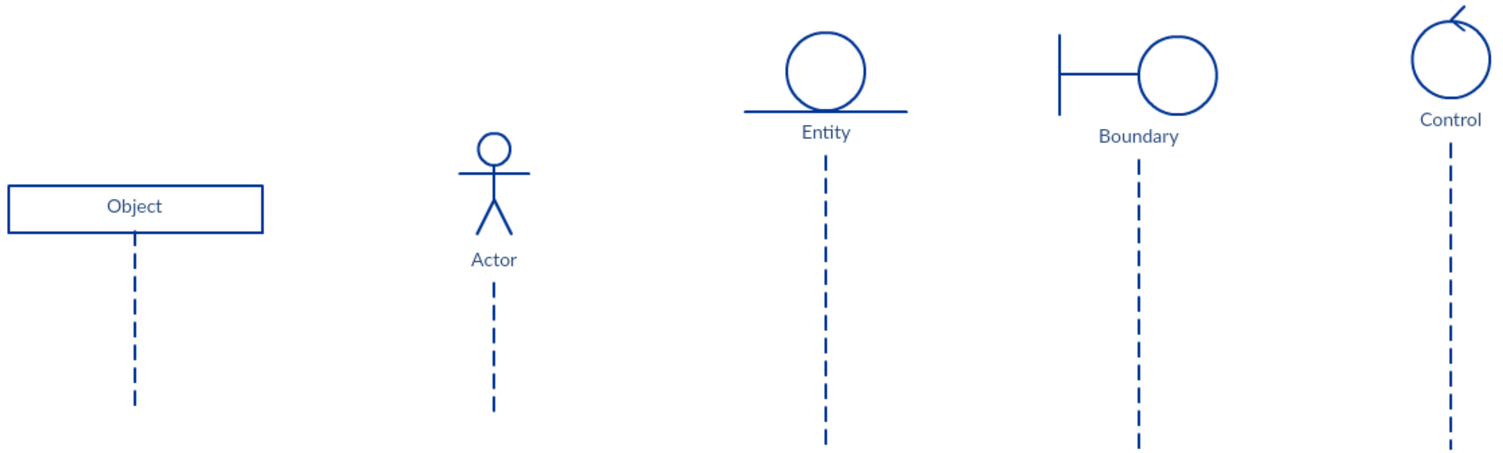
An **object** represents an *instance* of a class.

UML: Sequence Diagram - Actor



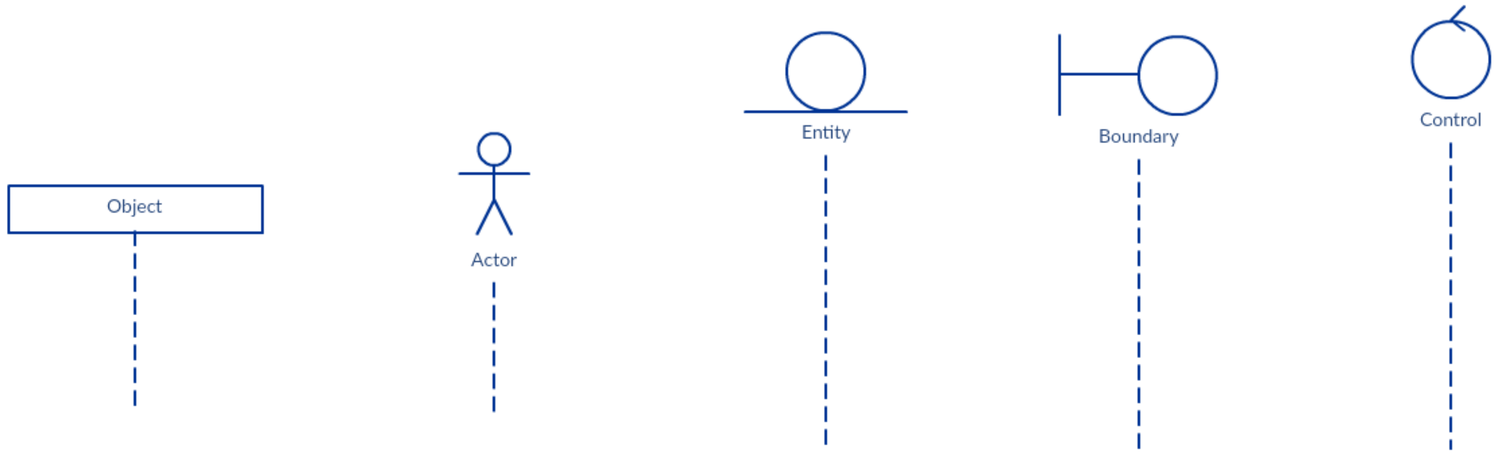
An **actor** represents a *user* in the system or an *other system* interacting with the current system

UML: Sequence Diagram - Entity



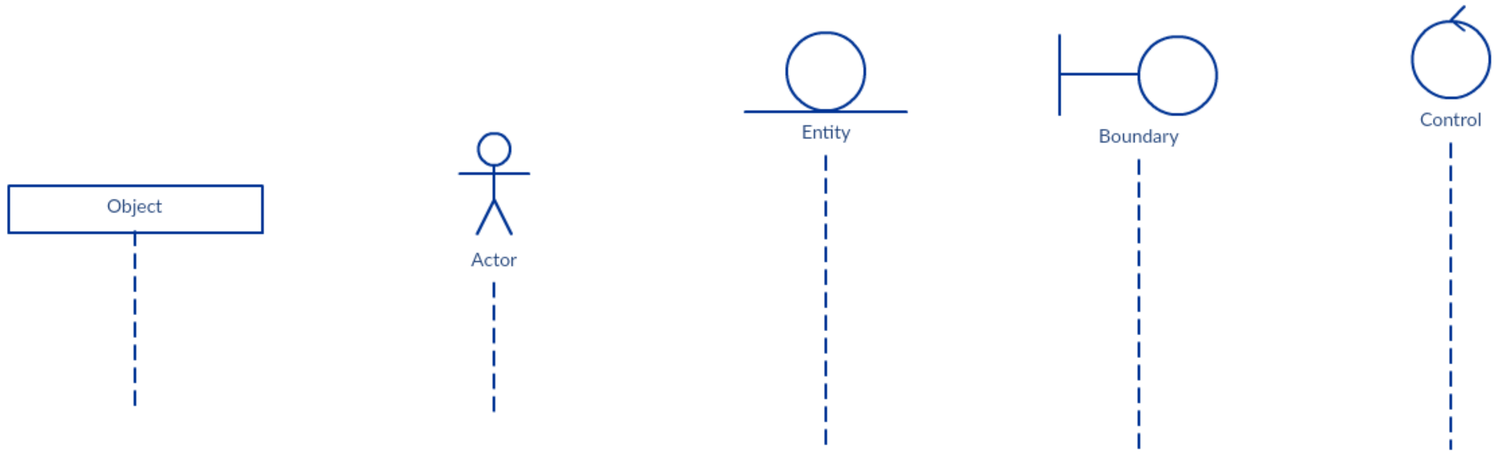
An **entity** is any singular, identifiable and separate object. It refers to individuals, organizations, systems, bits of data or even distinct system components that are considered significant in and of themselves.

UML: Sequence Diagram - Boundary



A **boundary** is a stereotyped Object that models some *system boundary*, typically a user interface screen. In an MVC (Model-View-Controller) pattern it represents the *View*.

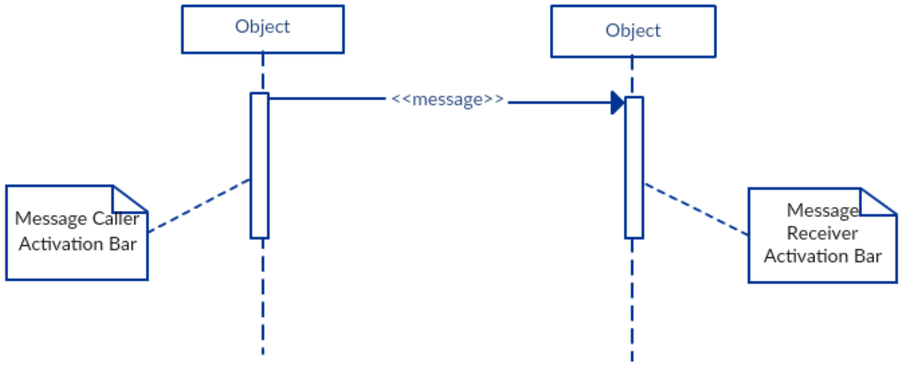
UML: Sequence Diagram - Control



A **Control** is a stereotyped Object that models a controlling entity or manager.

A Control organizes and schedules other activities and elements, typically in Analysis (including Robustness), Sequence and Communication diagrams. It is the *controller* of the MVC (Model-View-Controller) Pattern.

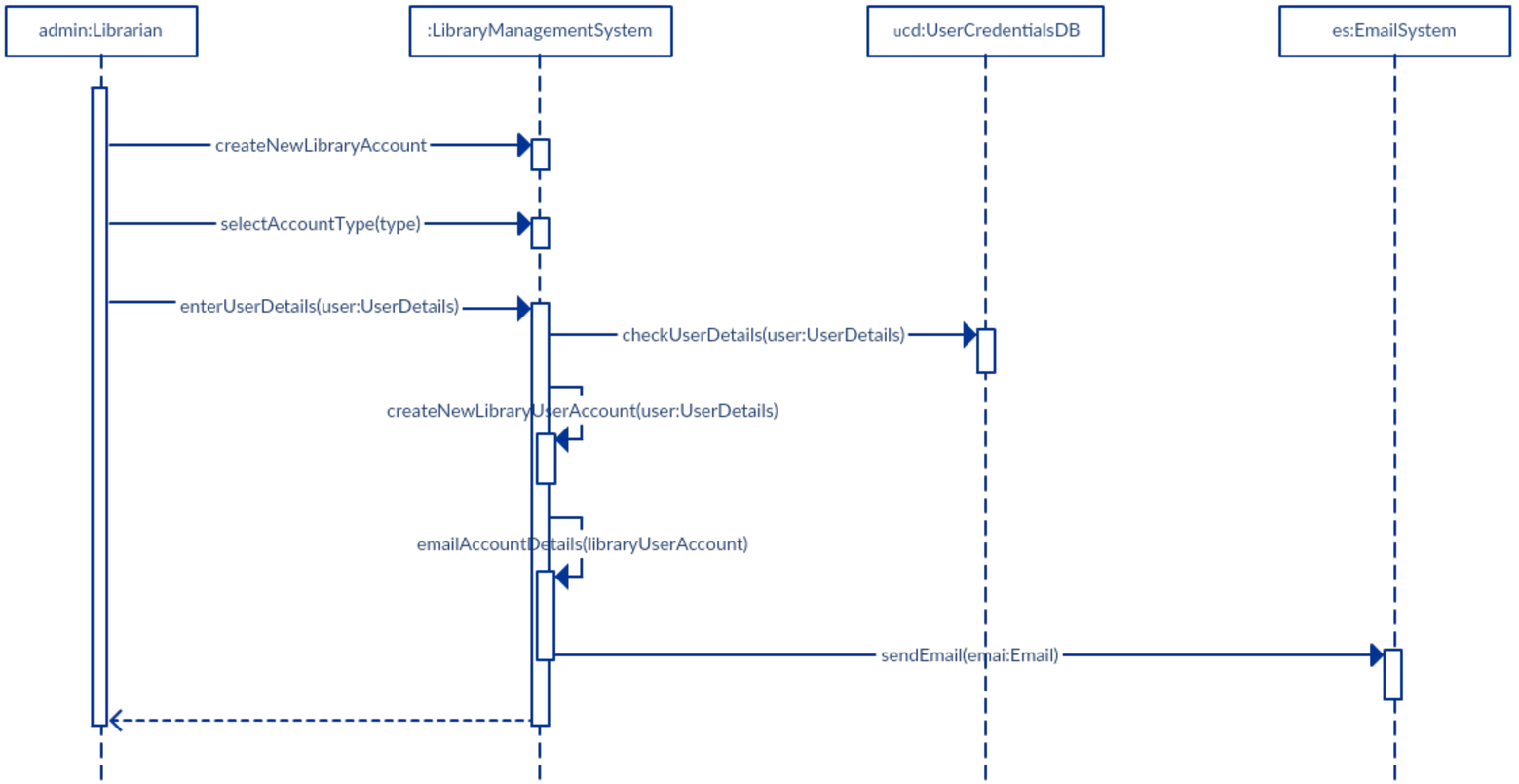
UML Sequence Diagram - Activation Bars



An **activation bar** is the box placed on the lifeline.

It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.

UML Sequence Diagram - Activation Bars



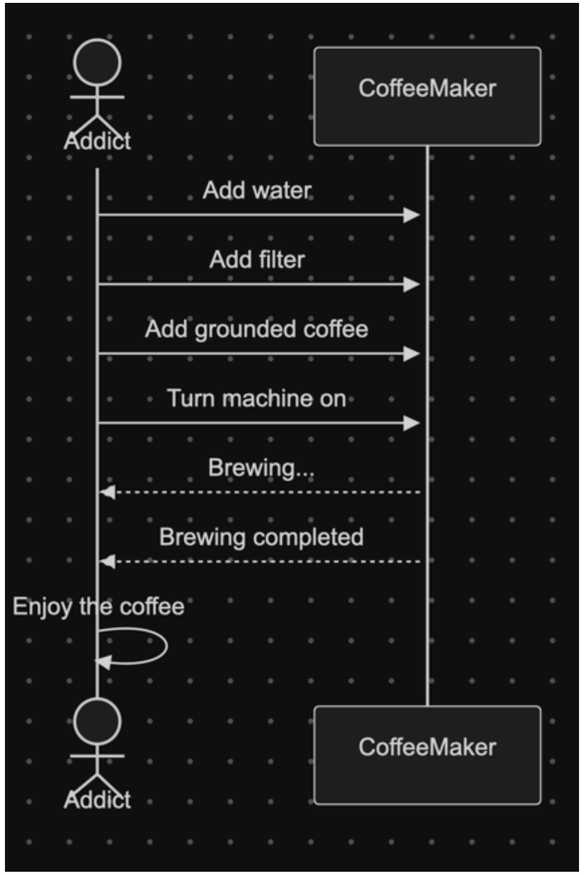
Brewing coffee sequence diagram

How to make Coffee?!

a. Put:
- Water
- A filter
- Grounded coffee

In a coffee maker (seen at the right).

b. Turn machine on…
c. Wait till it is brewed
d. Enjoy



Links on UML

Recommended resource: https://creately.com/blog/diagrams/sequence-diagram-tutorial/

https://www.tutorialspoint.com/uml/uml_interaction_diagram.htm

https://sparxsystems.com/enterprise_architect_user_guide/14.0/model_domains/otherelements2.html

Links on C++

https://www.bfilipek.com/2019/12/threading-loopers-cpp17.html

https://www.geeksforgeeks.org/multithreading-in-cpp/

https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp

https://www.softwaretestinghelp.com/multithreading-in-cpp/

C/C++ (< 11)

https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm

Any questions?