1. Active Object

2. Implementing Active Objects
Suppose we want to design a simple server
- Clients send processing requests to the server gateway
- The requests are processed according to their type
- While the server processes, clients should not be blocked

In other words, we want to implement an asynchronous method call:
- The client “sends” a request to the server and then continues its processing
- The server starts processing the request concurrently with the clients
- When the server completes the request, the result is stored so that it can be read later by the client
In a synchronous method call, there is only one thread of flow control: the client one. The client code can continue only when the function call returns.
Asynchronous method call

- in an asynchronous method call, both the client and the object have their own thread of flow control
- when a client calls a method on the object, it is actually sending a message, and after that it can continue its execution
- when it wants to get the result, it synchronises on the `Future` that contains the result
The Active Object design pattern decouples method execution from method invocation for an object

- Method invocation should occur in the client’s thread of control
- Method execution occurs in the Active Object’s thread of control
- Design should make this transparent: it should appear as the client is invoking and ordinary synchronous method
The Active Object design pattern decouples method execution from method invocation for an object

- Method invocation should occur in the client’s thread of control
- Method execution occurs in the Active Object’s thread of control
- Design should make this transparent: it should appear as the client is invoking and ordinary synchronous method
- To solve the latter problem, we must apply the “proxy” pattern
Pattern Components

- **A proxy**
  - Provides an interface to the client, with regular methods
- **A method request**
  - A hierarchy of classes that models client requests
  - We need one for each public method in the proxy
- **An activation list**
  - Contains the method requests object
- **A scheduler**
  - Decides with request must be processed next
- **A servant**
  - Processes the requests
- **A future**
  - Contains the response
The proxy has the right interface (the one that is seen by the client)
it is an ordinary object
The proxy has the right interface (the one that is seen by the client)

it is an ordinary object

however, it does not processes requests, but transforms each request (method call) into an appropriate “method request” object, which is then inserted into the “activation list”

it also prepares a *future*, that is an object that will contain the method return value (once the processing is complete)

the *future* is initially empty, and it is returned back to the client
Method requests

- A hierarchy of classes, each one models one request
- A method request encodes the method parameters, and contains a reference to a future (for the response)
- It may also contain other specific fields (e.g. priority, preconditions, etc.)
- The proxy creates method requests, and insert them into the activation list, according to some policy
The activation list is a protected object
- It will be shared between the client’s thread and the scheduler’s thread
- It can be implemented using the *Monitor pattern*
The activation list is a protected object
- It will be shared between the client’s thread and the scheduler’s thread
- It can be implemented using the *Monitor pattern*

the Scheduler is a thread that extracts requests from the activation list and executes them by calling the servant’s methods

The processing order can be customised
- For example, according to a FIFO order, or based on priority
- The scheduler also checks the *guards* on the request (i.e. conditions that must be true before a method can be executed)
- It is useful to define a specific separate class for defining the order, according to the *strategy pattern*
- In this way, the list can be easily customized with a user-defined scheduling policy
The Servant

- It has the same interface of the Proxy, and it implements the actual methods.
- It can also implement *guard methods* that are used by the method request object to understand if they can actually be processed.
A Client invokes a method on the Proxy

- the proxy creates a concrete method request,
- binds the method’s parameters to it
- if necessary, creates a future and binds it to the request
- inserts the request into the activation list
- if necessary, returns the reference to the future
Dynamics

- A Client invokes a method on the Proxy
  - the proxy creates a concrete method request,
  - binds the method’s parameters to it
  - if necessary, creates a future and binds it to the request
  - inserts the request into the activation list
  - if necessary, returns the reference to the future

- The scheduler monitors the activation list
  - it calls extract() in a loop, and it blocks if the queue is empty
  - when the extract returns a request object, it first checks the conditions, by calling the guard() method
    - this is resolved by calling the preconditions methods on the servant
  - if the response is true, the scheduler runs it by calling method call()
    - this is resolved into a method call to the servant object, the result is stored in the future
  - if the response is negative, the request is enqueued again in the list
  - when no other request is runnable, the scheduler suspends itself
It is convenient to start implementing and testing the Servant object

- In this way we can test the functionality separately
- There is no need to implement synchronisation in the Servant, because its methods will be automatically serialised by the Scheduler
- In other words, all its methods are called by a single thread, the Scheduler
- However, we have to export part of the state through appropriate getter methods, so that we can implement the “guards” in the request objects
- these are equivalent to the blocking conditions in a monitor
- also, they are “equivalent” to guards in the Rendez-vous interactions of Ada
Implementing the method requests

- Method requests are similar to the *Command pattern*
  - they encode commands with their parameters and results
  - they also implement *guards*
  - they contain a reference to the Servant

- A base class *MethodRequest* implements the basic methods for checking the guards and executing the request
  - they are declared as pure virtual functions, and will be implemented in the concrete classes
  - the concrete classes also are in charge of storing the parameters and the future

- Notice that requests are created by the Proxy, but will be destructed by the Scheduler
  - For safety, it may be the case to wrap them inside smart pointers (e.g. `shared_ptr<>`), to avoid memory leaks or memory corruption due to exceptions and border cases
Proxy

- It simply creates the method requests
- It may use a factory pattern (for example, factory method, or abstract factory) for the purpose
- if it can be called by several concurrent clients, it may need to be synchronised (especially if it uses a single factory)
  - a simple mutex is sufficient
  - to generalize, we can use a *strategized locking* pattern, so that the Proxy can be configured with a regular mutex, or with a null mutex (when used by only one client), thus reducing overhead
- after creating the request, it inserts it into the Activation List
The Activation List is a queue of requests

- It needs to be synchronised because there are at least two threads using it: the client and the scheduler’s thread
- usually implemented using the *monitor pattern*

When the list is full we can:

- Block the client (with or without timeout)
- return an error
- raise an exception (will be caught in the client’s thread)

The scheduler may need to go through the list to find the first *runnable* request

- Therefore, we should let it iterate through the list and invoke the `guard()` method on every request until it finds a “good” one
Future

- The future needs to implement a *rendez-vous* policy
  - The client that calls the `get_result()` method too soon will be blocked waiting for the result to become ready
  - it will be unblocked when the result has been computed by the servant
  - it’s the request object responsibility to fill the result and unblock the client
- in practice, a future is a single-buffer synchronized queue that is used once
  - however, we must be careful with its lifetime
  - the future is created by the proxy as an “empty” buffer and returned to the client
  - the client becomes the “owner” of the future (the one responsible for its deallocation)
  - however, the pointer to the future is also used by the request object to store the result
  - to deal with corner-cases (exceptions, errors, etc.) without memory corruption and memory leaks, we should use a smart pointer
Outline

1. Active Object

2. Implementing Active Objects
The Active Object pattern requires the implementation of many classes.

In particular, for every operation on the Servant (and on the Proxy), a Message Request class must be prepared.

Every such class must encode all the parameters of the operation and maintain a future, and call the appropriate Servant operation.

This is a lot of code, that must be written and tested.

It is also “boring code”, that could be made automatic.

It is a candidate for “code generation tools”.

Even code generators like Rapsody require the programmer to “draw” classes, operations and attributes.

Another way of generating code it through C++ templates.
If we want to automatize the production of request classes, we could take advantage of type lists

- We can use type lists to specify a certain number of parameters for the constructor of the request class
- We can also use templates for generating calls to the Servant object