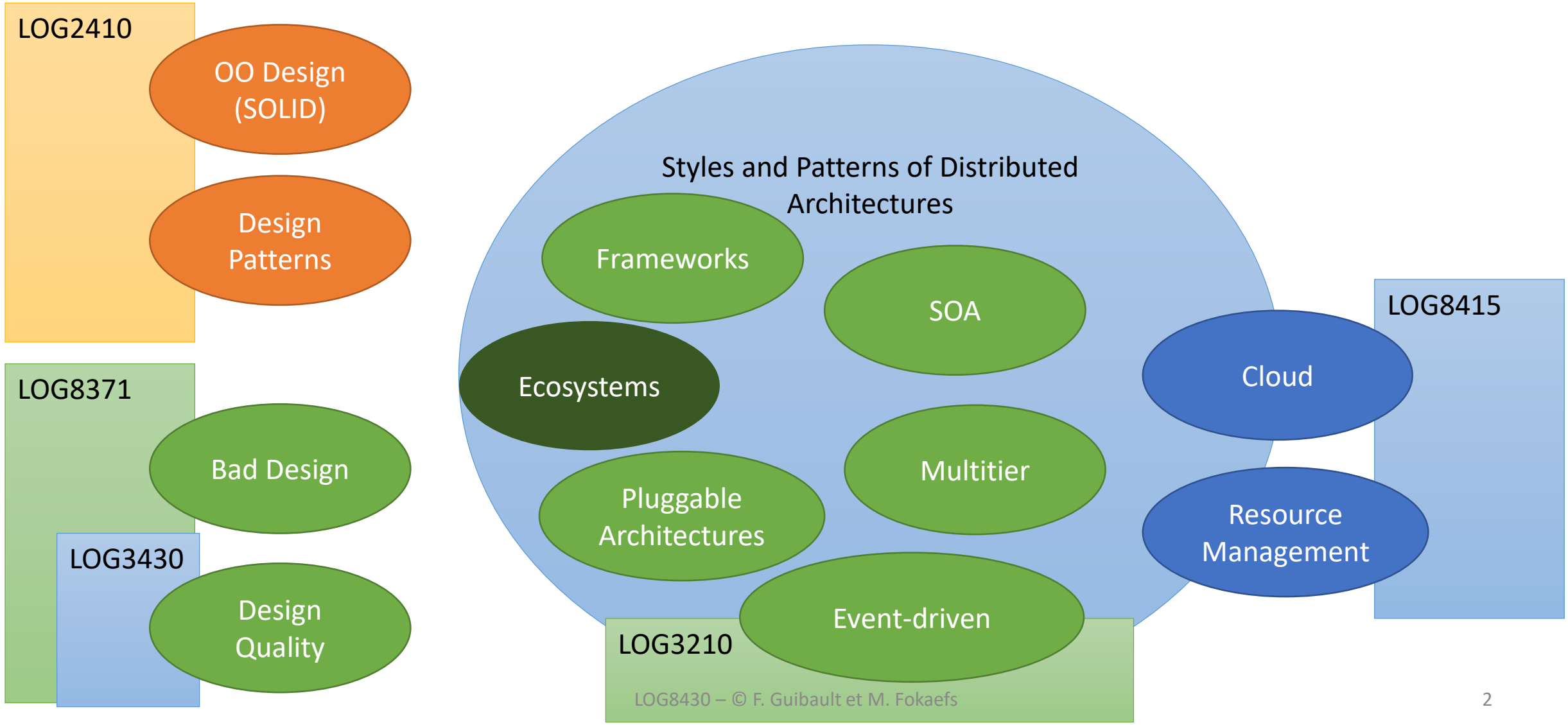
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG8430E: Event-driven Architectures and Concurrency Patterns

Course Map


LOG2410
OO Design (SOLID)
Design Patterns
LOG8371
Bad Design
LOG3430
Design Quality
Styles and Patterns of Distributed Architectures
Frameworks
SOA
Ecosystems
Cloud
Multitier
Pluggable Architectures
Resource Management
Event-driven
LOG3210
LOG8415

So far…

LOG2410

OO Design (SOLID)

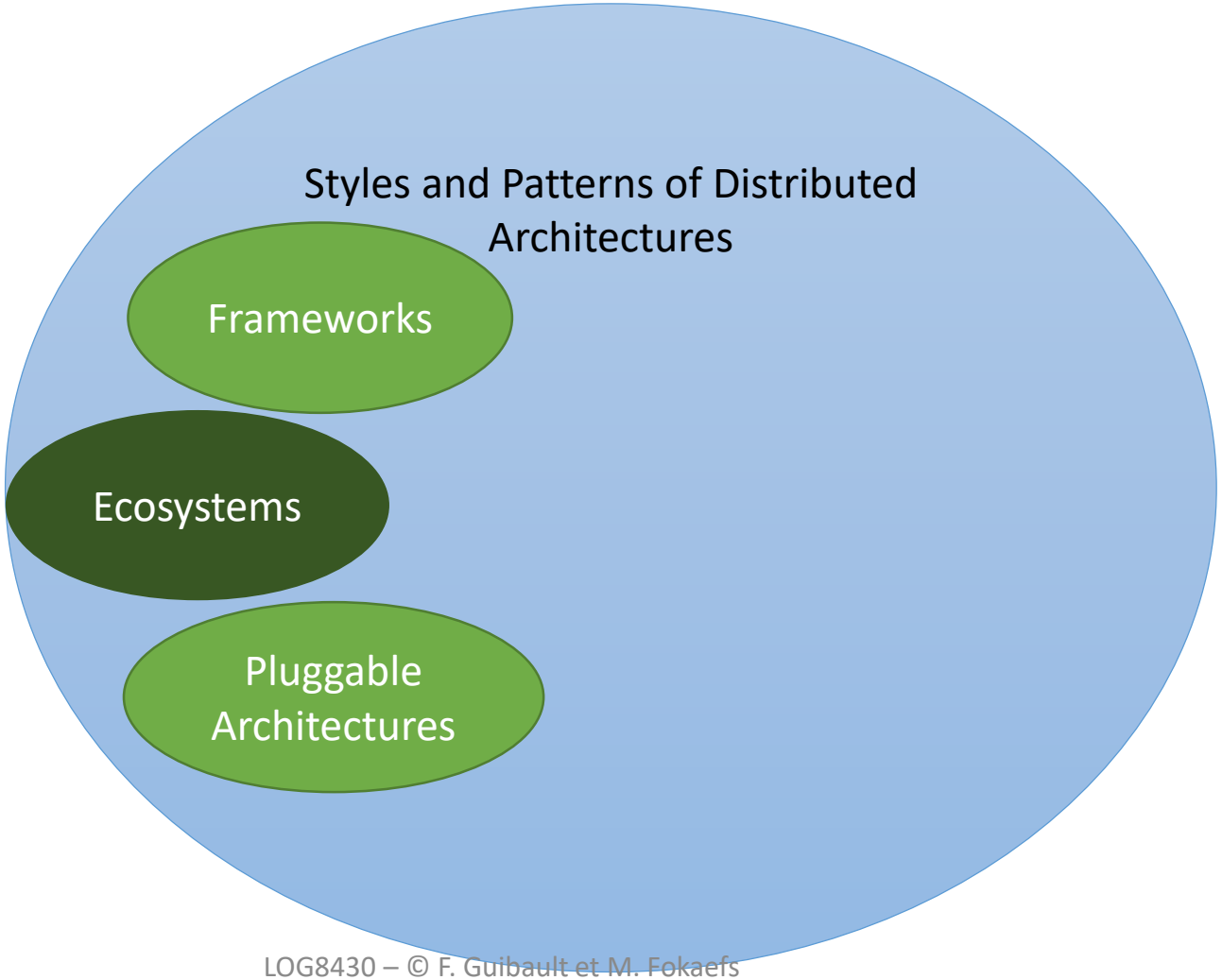
Design Patterns

LOG8371

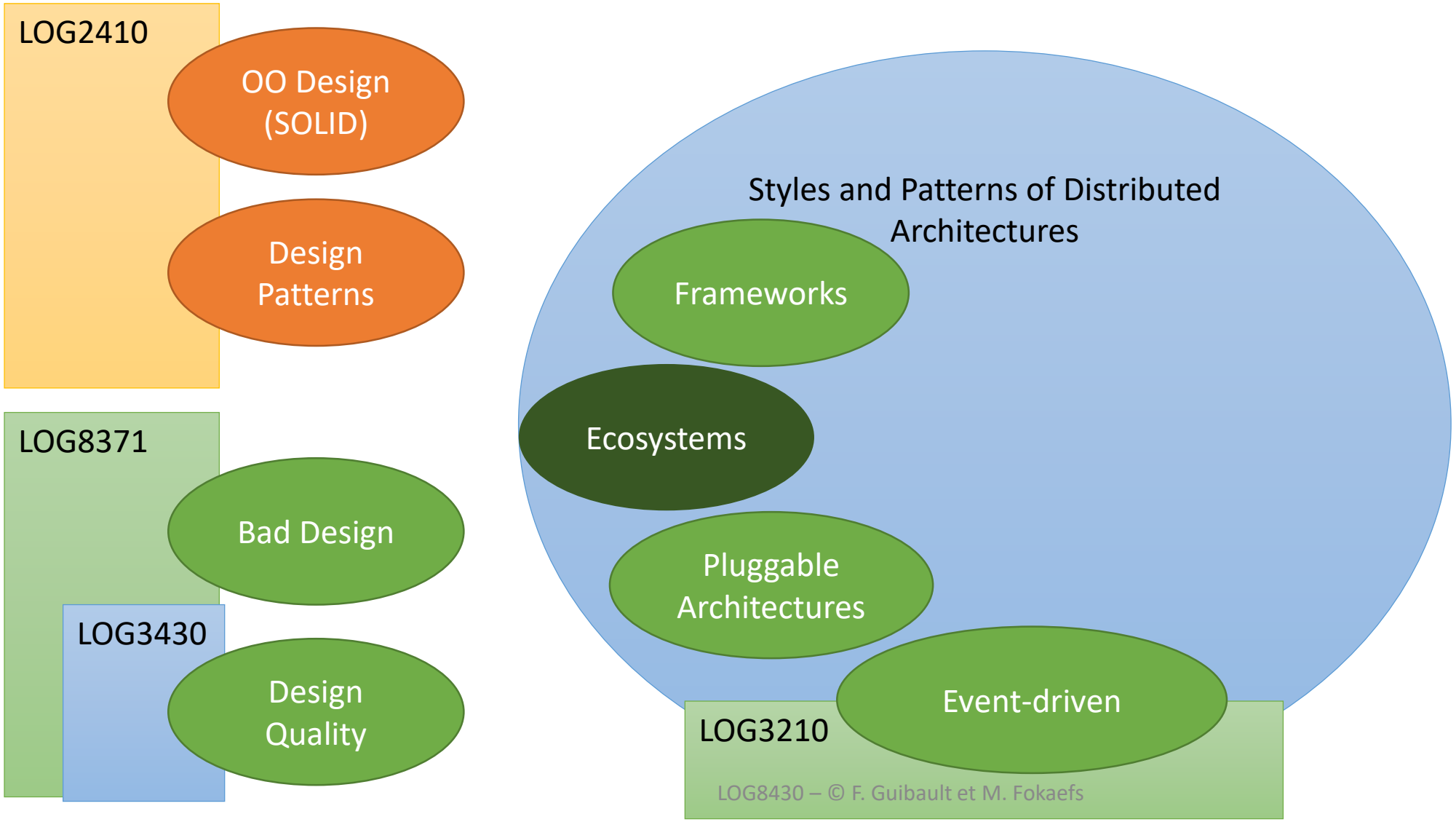
Bad Design

LOG3430

Design Quality

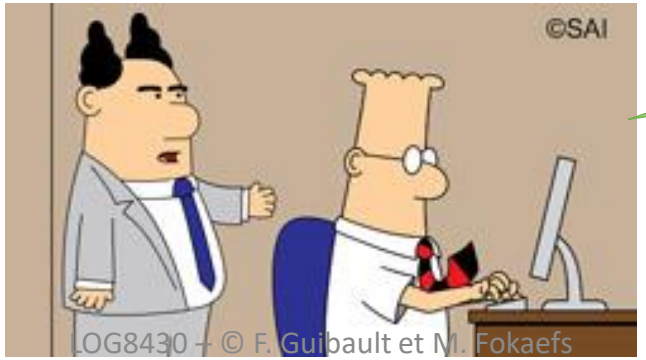


Today


LOG2410
OO Design (SOLID)
Design Patterns
LOG8371
Bad Design
LOG3430
Design Quality
Styles and Patterns of Distributed Architectures
Frameworks
Ecosystems
Pluggable Architectures
Event-driven
LOG3210

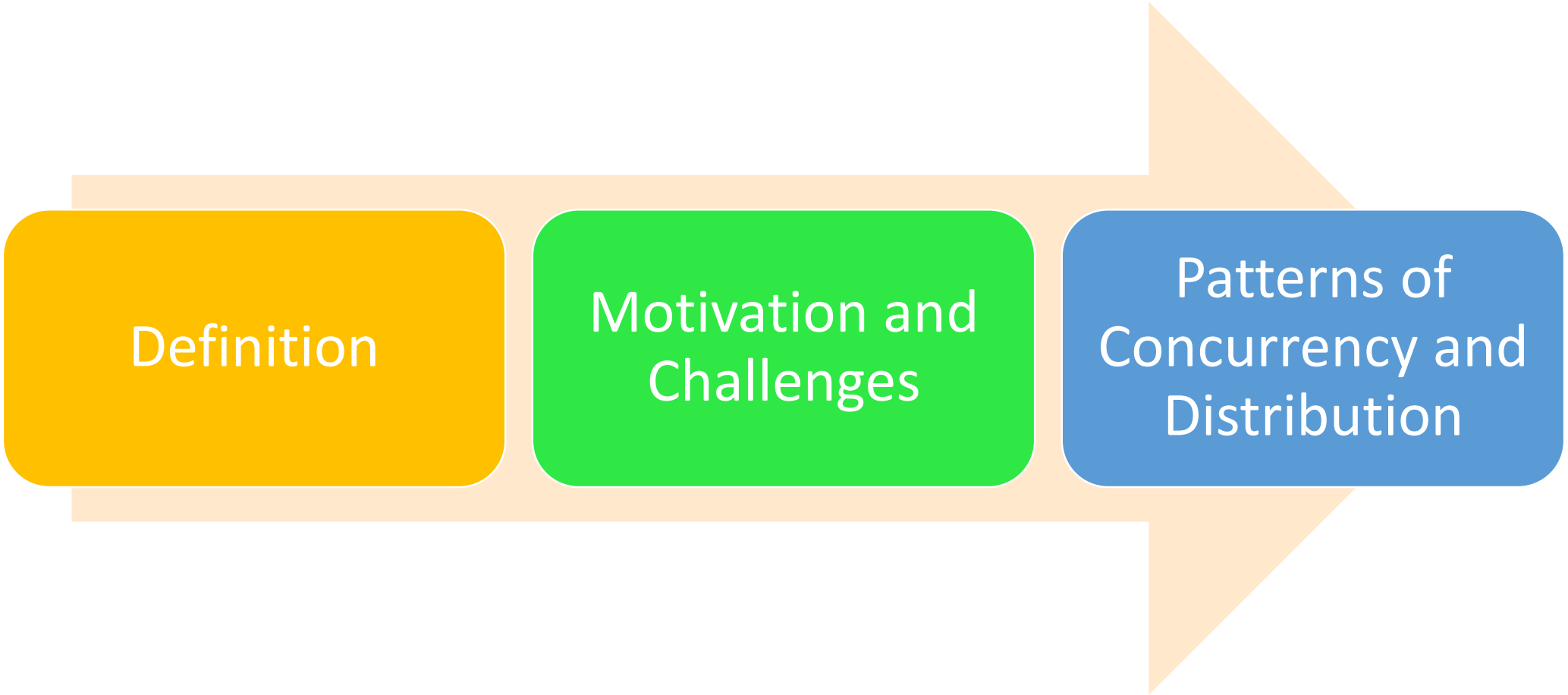
Scenario…

- Welcome to the company!
- Our software products follow an event-driven architecture.
- Your responsibilities include the development of new modules, the maintenance and the evolution of existing modules.
- The majority of the decisions on the architecture are already made, but do not hesitate to take new ones!



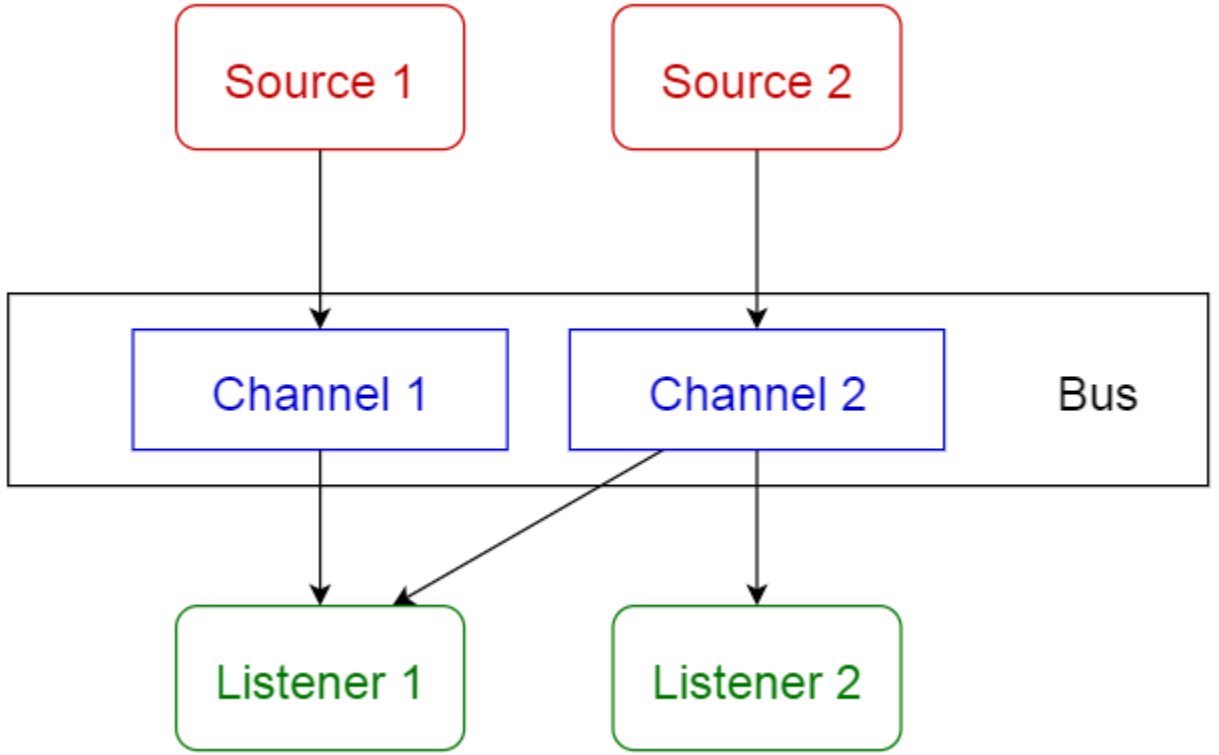
- But…
- What is an event-driven architecture?
- Why do we use it and what are the advantages?
- Is it easy to design and implement?
- Are there examples and patterns to implement such an architecture?

Event-driven architectures



Definition

- In these architectures, the events that occur affect the state or the behavior of the objects.
- **Event**: Any action that is relevant to the system , e.g., a click on a graphical interface, a change on data.
- **Message**: A notification that follows the event, produced by the system or by the producer of the event and destined for the listener of the events. (Sometimes the messages are confounded with the events.)
- **Event bus**: A channel or a buffer where the events are registered, temporarily stored and transmitted to listeners.



Properties

- Increased capacity for parallelization.
- The bus is common and it can be shared between multiple producers and listeners.
- The same event can be processed by multiple listeners.
- The listeners, as processors of events, can be replicated to parallelized identical tasks. This increases the scalability of the system.
- The events can be technology agnostic, and so we can define different interfaces for different languages. This is the base for service-oriented architectures.

Motivation

- Hardware is not very expensive, we can add computation and processing power without space or financial constraints.
- Modern computers do not have a single CPU, but they work with multicore CPU. This facilitates parallelisation a lot!
- Network technologies have also evolved. The speed and capacity of networks have significantly increased.
- Distributed and multithreaded applications are everywhere nowadays.
- Some advantages of distributed systems include:
  - Collaboration and connectivity: distributed data and resources are now possible.
  - Performance, scalability and fault tolerance are increased.
  - The economies of scale and the network externalities improve cost management.

Challenges

Four principal challenges for the adoption of an event-driven architecture.

1. Service access and configuration.
2. Synchronisation
3. Concurrency
4. Event management

1. Service access and configuration

- The distributed services can be accessed using APIs at different levels of abstraction:
  - Interprocedural communication based on shared memory.
  - Communication protocols such as TELNET, FTP, SSHT, etc.
  - Remote method calls using middleware, like COM+, CORBA, or JAVA-RMI.
- The static or dynamic evolution of services is a complex problem that requires tools to manage:
  - The evolution of interfaces and the relations between components,
  - The dynamic reconfiguration of resources to respond to load changes.
- The clients should not be affected during the evolution.

2. Synchronisation

- How can we correctly synchronise the access to shared resources?
  - Adequate acquisition and release of locks.
  - Flexibility in the choice of synchronisation mechanisms.
  - Minimization of locking overhead and prevention of self-locking in intra-component method calls.
  - Reduction of conflicts and overhead for the critical sections rarely executed.

3. Concurrency

- The simultaneous execution of multiple threads and processes requires the management of frequent difficulties related to concurrency such as:
  - The blocking and the state of concurrency.
  - Multithreaded and non-portable APIs.
  - Contradictions in the semantics of threads between operating systems.
- The fundamental problems of designing concurrent systems include:
  - The choice of an efficient architecture that minimizes the overhead.
  - The combination of synchronous and asynchronous tasks.
  - The selection of synchronisation primitives.
  - The elimination of useless threads and locks in real-time concurrent applications.

4. Event management

Three important characteristics distinguish the event-driven applications from applications that have a self-managed control flow:

1. The behavior of the application is triggered by internal or external events that occur in an asynchronous fashion.
2. The majority of the events should be processed immediately to avoid resource saturation and an increase of response time.
3. Finite-state automata can be necessary to control the event processing and detect illegal transitions.

Concurrency patterns

Service access and configuration patterns

- **Wrapper Façade**
- **Component configurator**
- **Interceptor**
- **Extension Interface**

Synchronisation patterns

- **Scoped locking**
- **Strategized locking**
- **Thread-safe interface**
- **Double-checked locking optimization**

Concurrency patterns

- **Active Object**
- **Half-Sync/Half-Async**
- Monitor Object
- Leader/Followers
- Thread-Specific Storage

Event handling patterns

- Reactor
- Proactor
- Asynchronous Completion Token
- Acceptor-Connector

Service access and configuration patterns

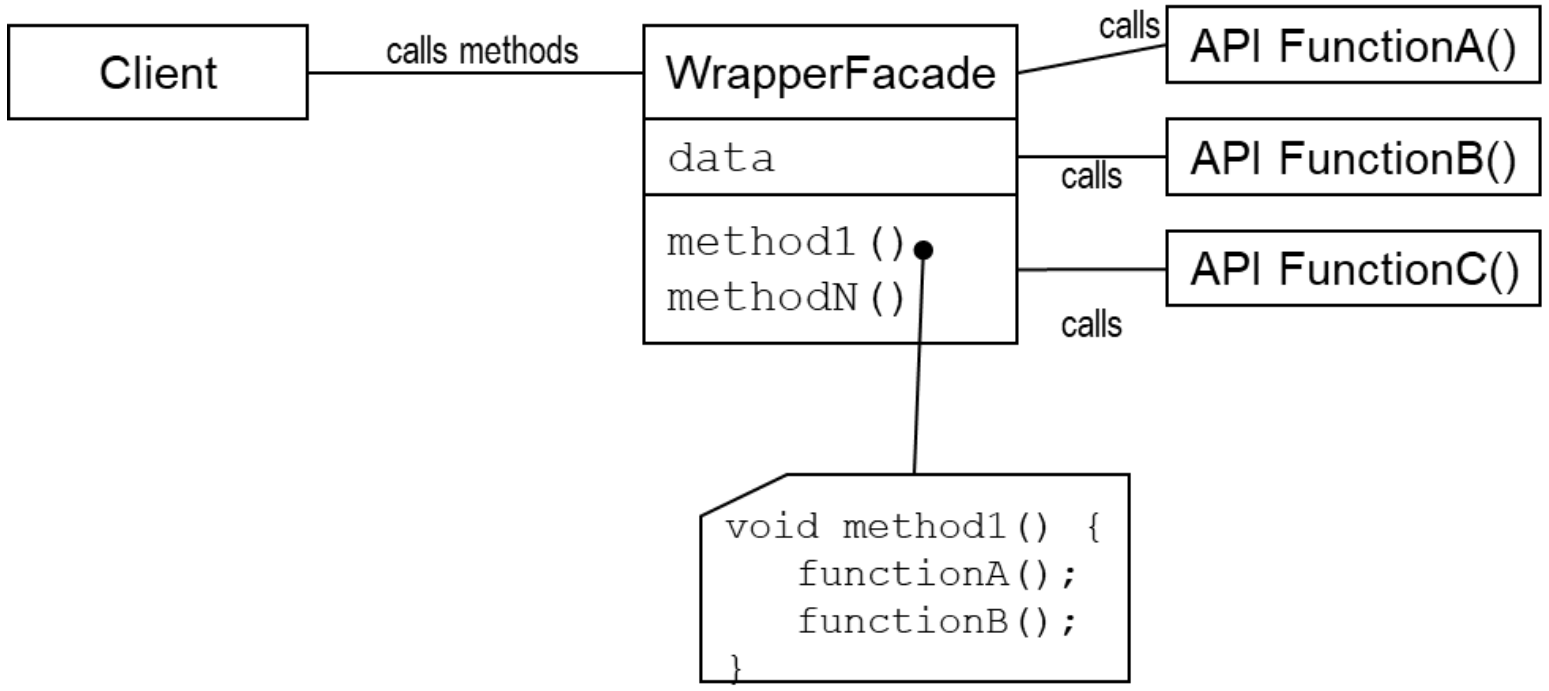
1. Wrapper Façade
2. Component configurator
3. Interceptor
4. Extension Interface

Wrapper Façade

- **Objective**: Encapsulate the functionalities and the data provided by existing libraries (often non object-oriented) by more cohesive, reliable, portable and easily maintainable interfaces.
- **Application/example**: Encapsulate the functionalities of a lower layer (e.g., of the operating system) in an interface of a higher level to make the more portable and reusable. See the example of Android.

Wrapper Façade

- **Structure** :



Wrapper Façade

- **Consequences**

  + Concise, cohesive and reliable object-oriented interfaces of higher level.

  + Increased portability, maintainability, modularity and reusability.

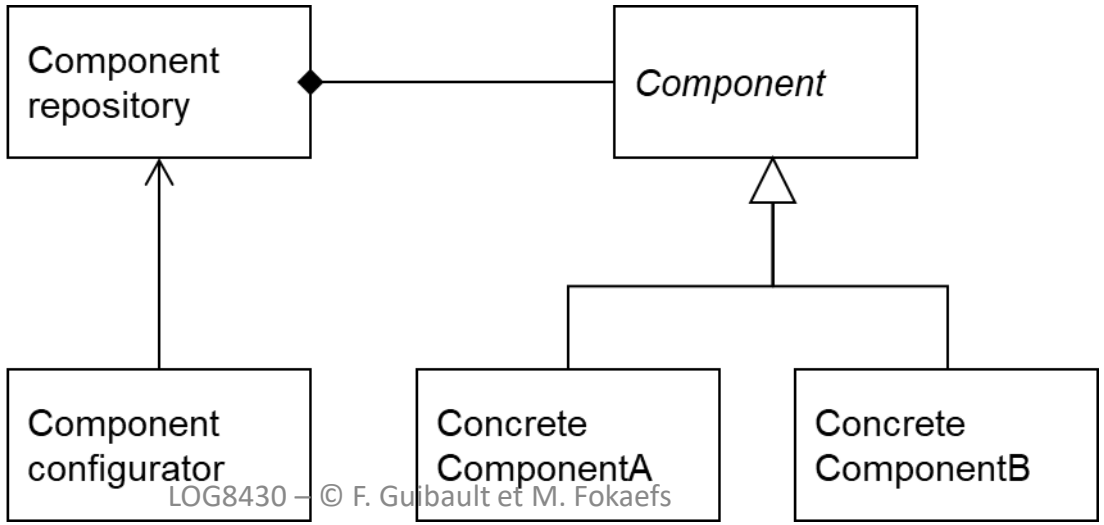
  - Loss of functionality.

  - Reduced performance.

  - Imposed limitations by the languages and the compilers.

Component configurator

- **Objective:** Allow an application to bind and unbind the implementations of its components dynamically at runtime by avoiding to modify, recompile or rebind the application statically. Moreover, the pattern allows the reconfiguration of components in distinct processes without having to terminate or restart the active processes.
- **Application:** The pattern is applied when we want to dynamically change the implementation of a component, but to avoid loading all possible configurations in memory at the start of the application.
- **Structure**:

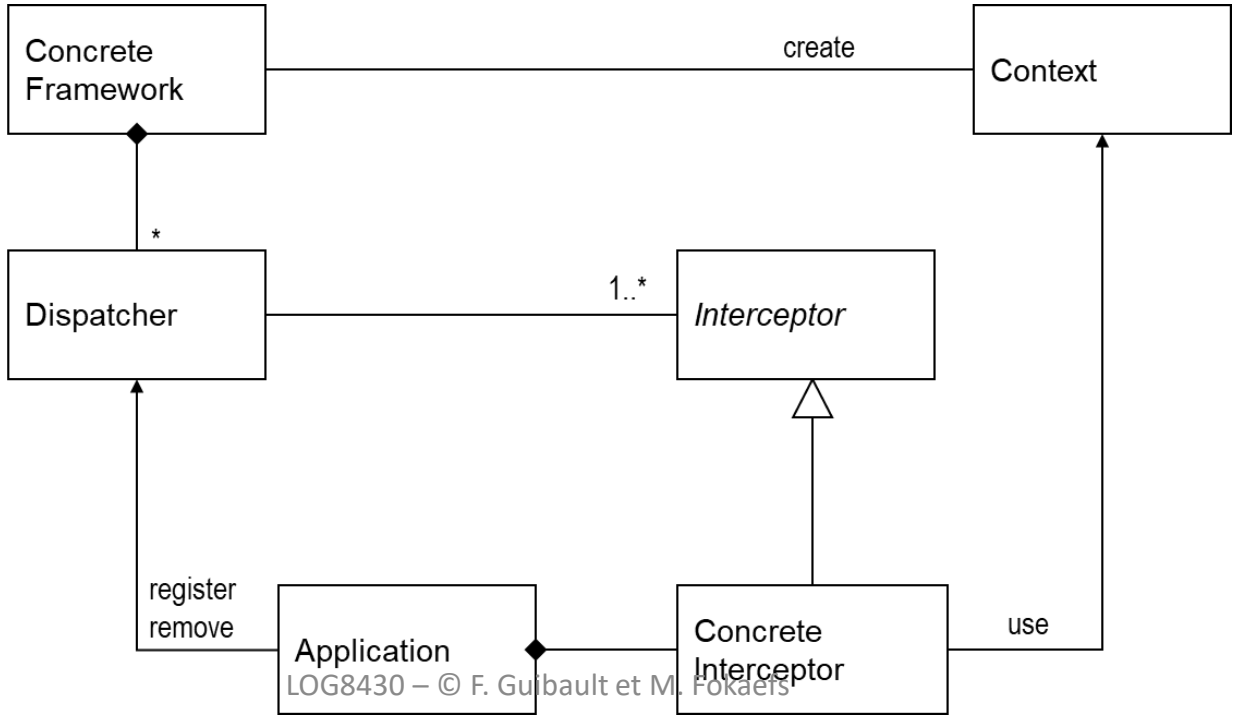


Component configurator

- **Consequences**
  + Uniformity of the configuration and the control of the interface.
  + Centralized administration.
  + Increased modularity, testability and reusability.
  + Dynamic configuration
  + Increased optimization thanks to multiple configuration possibilities.
  - Increase of the uncertainty due to all the possible interactions between the different dynamically configured components.
  - Reduced security and reliability.
  - Increased complexity and reduced performance.

Interceptor

- **Objective:** Allow services to be added in a framework in a transparent way and to be started automatically when certain events occur.
- **Application:** When a framework needs to be able to register and trigger new services that were not originally planned. Also, to allow applications to control the behavior and the functionality of the framework.
- **Structure** :

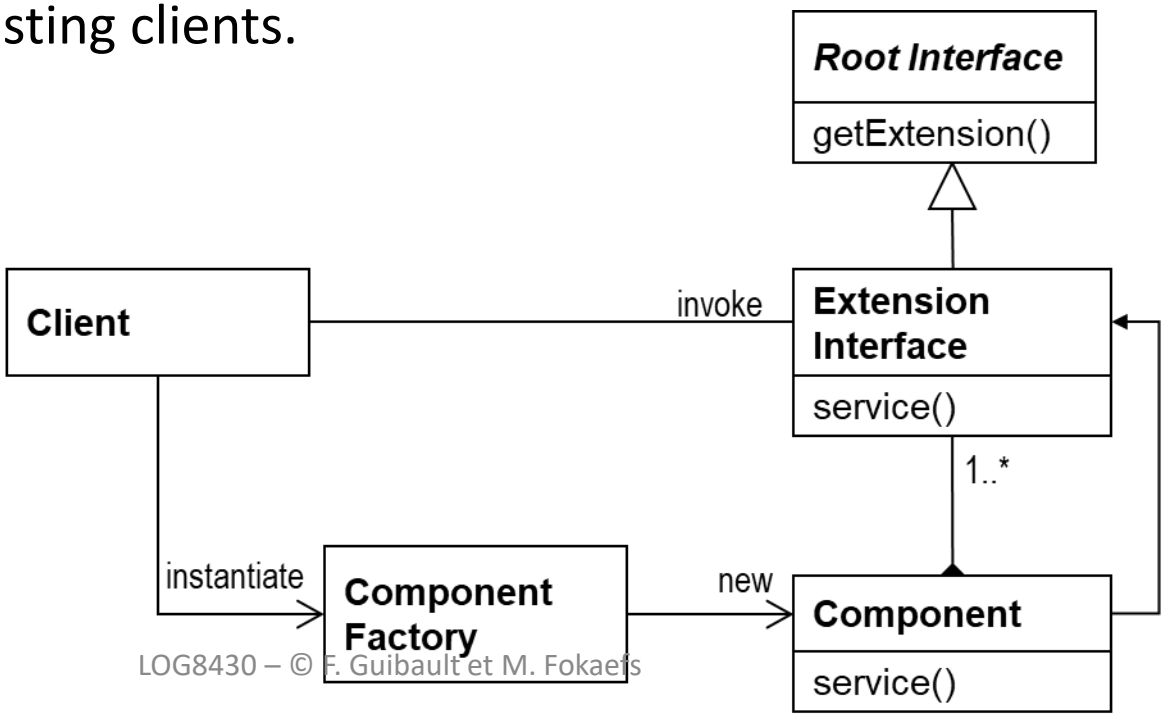


Interceptor

- **Consequences**
  + Increased extensibility and reliability of the framework.
  + Separation of responsibilities between services.
  + Capacity to monitor and control the frameworks.
  + Reusability of interceptors
  - Difficulty in anticipating the events that need to be intercepted and to separate the interfaces of the interceptors.
  - The interceptors are insertion points of vulnerabilities and faults.
  - Possibility for cascading interception due to changes in the framework.

Extension Interface

- **Objective:** Allow a component to expose multiple interfaces to guarantee the Interface Segregation Principle, avoid entanglement of interfaces and affect clients when the developers extend or modify a component.
- **Application:** When we expect that we will need to change the interfaces of components in a way impossible to anticipate after the release of the component and its integration in applications. When the addition of new interfaces to accommodate the needs of new clients should not affect existing clients.
- **Structure**:



Extension Interface

- **Consequences**
  + Provide extension mechanisms for components.
  + Promote the separation of responsibilities between roles.
  + Support of polymorphism between classes that are not linked hierarchically.
  + Decouple the components from their clients.
  + Support the aggregation and the delegation of interfaces.
  - Increased effort to design and implement components.
  - Increased complexity of clients.
  - Additional indirection and execution overhead.

Synchronization patterns

1. Scoped locking
2. Strategized locking
3. Thread-safe interface
4. Double-checked locking optimization

Scoped locking

- **Objective**: Ensure that a lock is acquired when the control enters a specific scope and that it is released automatically when it quits the scope, whatever path is taken to quit the scope.
- **Application**: When a critical section of a method needs to be protected by a locking mechanism. To avoid exiting a method by a return or because an exception was triggered without releasing the lock.
- **Problem**: Code, which needs to be executed concurrently, needs to be protected by a lock that is acquired and released when the control enters and exits a critical section. If the developers need to explicitly acquire and release locks, it will be difficult to ensure that the locks will be released in all paths defined by the code.
- **Solution**: Define a guardian class, whose constructor automatically acquires a lock when the control enters in the scope and whose destructor automatically releases the lock when the control quits the scope. Instantiate the guardian class to acquire/liberate locks in the scopes of methods or blocks defining critical sections.

Scoped locking

- **Consequences**
  + Increased robustness and reliability
  - Potential deadlock when the pattern is used recursively.
  - Limitations imposed by language semantics. It assumes the availability of destructors, that are not always called in unexpected terminations (exit or abort) or particular language constructs (e.g., `longjmp()` in C).

Strategized locking

- **Objective**: Parameterize the synchronisation mechanisms that protect the critical sections of the code against simultaneous access.
- **Application**: When a system has components that need to be efficiently executed in a variety of concurrent architectures.
- **Problem**: The components that are executed in multithreaded environments need to protect their critical section against simultaneous access by multiple clients. The integration of synchronisation mechanisms in functionalities of components need to find a fair balance between the necessity to respect the needs of the different applications (mutex, read/write locks, semaphores) and the necessity to avoid duplication.
- **Solution**: Parameterize the synchronisation aspects of a component by employing pluggable types. Each type represents a particular synchronisation strategy. Define the occurrences of this plugin types as objects contained in a component that can use the objects to effectively synchronize their implementations.

Strategized locking

- **Consequences**
  - + Increased flexibility and customizability.
  - + Reduced effort the maintenance of components.
  - + Increased reusability
  - - Intrusive locking
  - - Overengineering

Thread-safe interface

- **Objective**: Minimize the locking overhead and ensure that the intracomponent method calls are not “self-blocking” by trying to reacquire a lock already reserved by the component.
- **Application**: When the critical code of a component, that needs to be protected against simultaneous access, is spread among multiple methods that call other methods within the same component or recursively.
- **Problem**: Multithreaded components often contain multiple public and private methods that can change the state of the component. The methods can call one another to perform their calculations. In this case, the invocation of methods need to designed so as to avoid self-blocking and to minimize the locking overhead.
- **Solution**: Structure all components that process the intracomponent invocations according to two design conventions:
  - It’s the interface methods that control the locks.
  - Implementation methods trust the public methods to control the locks.

Thread-safe interface

- **Consequences**
  - + Increased robustness, reliability and performance.
  - + Simplification of software.
  - - Additional indirection and additional methods.
  - - Potential intercomponent deadlock.
  - - Potential intracomponent deadlock between different objects.
  - - Potential overhead.

Double-checked locking optimization

- **Objective**: Reduce the conflicts and the synchronisation overhead when critical sections of a code need to acquire a lock only once during the execution of a program.
- **Application**: When a component has a critical section that needs to be protected against simultaneous access, in the initialization code, which is executed only once, and when scoped locking is not efficient.

Double-checked locking optimization

- **Problem**: The concurrent applications need to ensure that certain parts of their code is executed sequentially to avoid concurrency situation in case of access or modification to shared resources. A concurrent means to protect the critical sections is to use scoped locking, but this can represent an unacceptable overhead when the code to protect should be executed only once.

```
class Singleton {
public:
  static Singleton* instance () {
    if( instance == 0 ) {
      // Enter critical section.
      instance_ = new Singleton();
      // Leave critical section
    }
    return instance_;
  }
};
```

Double-checked locking optimization

- **Solution**: Introduce a Boolean variable indicating if it is necessary to execute a critical section before acquiring a lock to protect it. If the code does not need to be executed, the critical section is ignored, which avoids unnecessary locking overhead.

```
// Perform first-check to evaluate 'hint'
If (first_time_in_flag is TRUE) {
   acquire the mutex
   // Perform double-check to avoid race condition
   If (first_time_in_flag is TRUE) {
      execute critical section
      set first_time_in_flag to FALSE
   }
}
```

```
class Singleton {
public:
   static Singleton* instance () {
      if( instance == 0 ) {
         // Use Scoped Locking to acquire and
         // release lock automatically.
         Guard<Thread_Mutex> guard( singletonLck );
         // Double check
         if( instance_ == 0 )
            instance_ = new Singleton();
      }
      return instance_;
   }
};
```

Double-checked locking optimization

- **Consequences**
  - + Minimize locking overhead.
  - + Prevention of race conditions
  - - Additional usage of mutex
  - - 2 potential problems linked with certain architectures:
    - Non-atomic assignment of integers and pointers.
    - Coherence of cache in multiprocessor systems.

Concurrency patterns

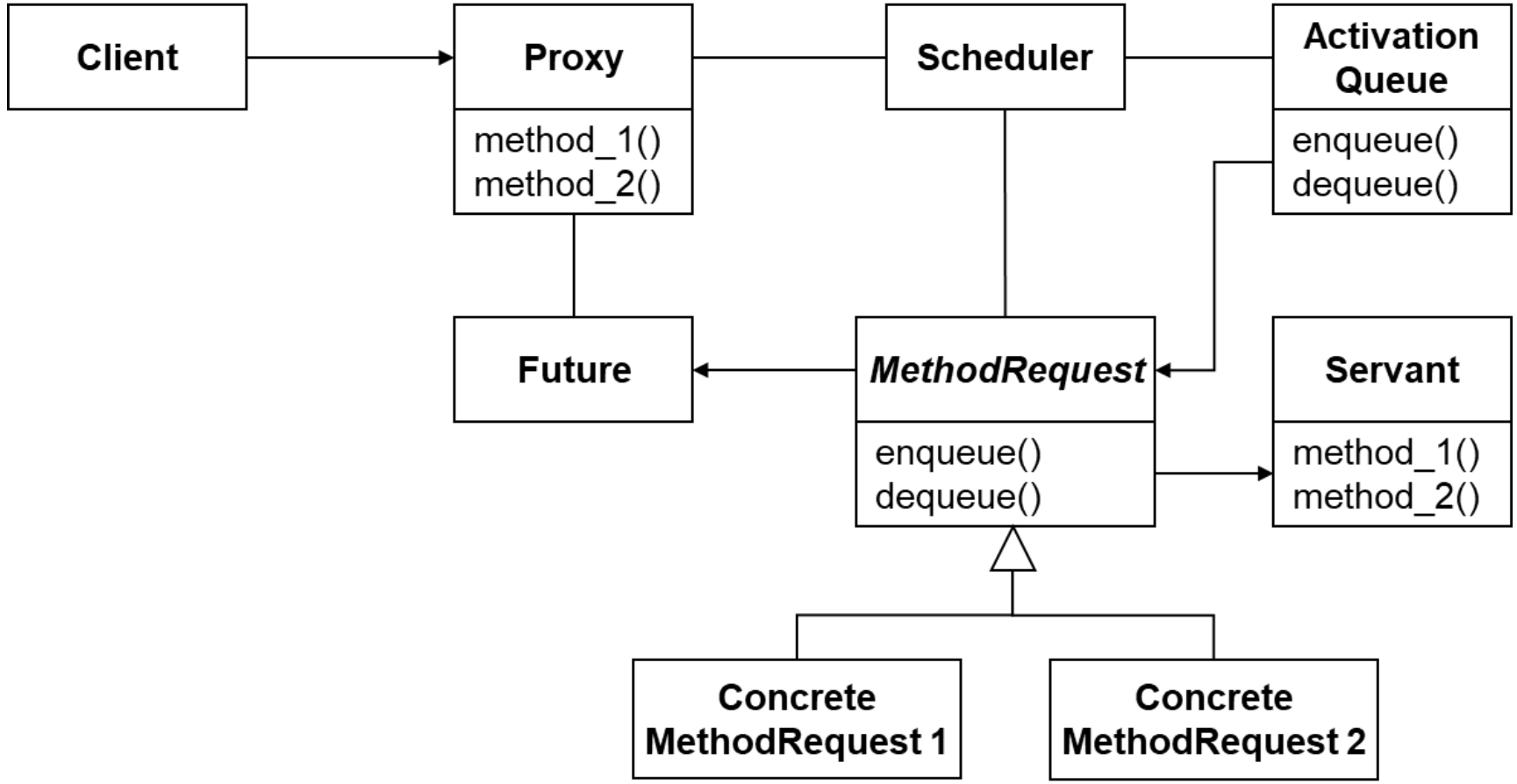
1. Active object
2. Half-sync/half-async
3. Monitor object
4. Leader/followers
5. Thread-specific storage

Active object

- **Objective**: Decouple the method execution from the method invocation to improve concurrency and simplify the simultaneous access of objects that reside in their own control threads.
- **Application**: To improve the performance when clients access objects being executed in separate control threads. When clients do not need to be bound to specific details about the synchronisation, serialisation, or scheduling of methods.

Active object

- **Structure** :

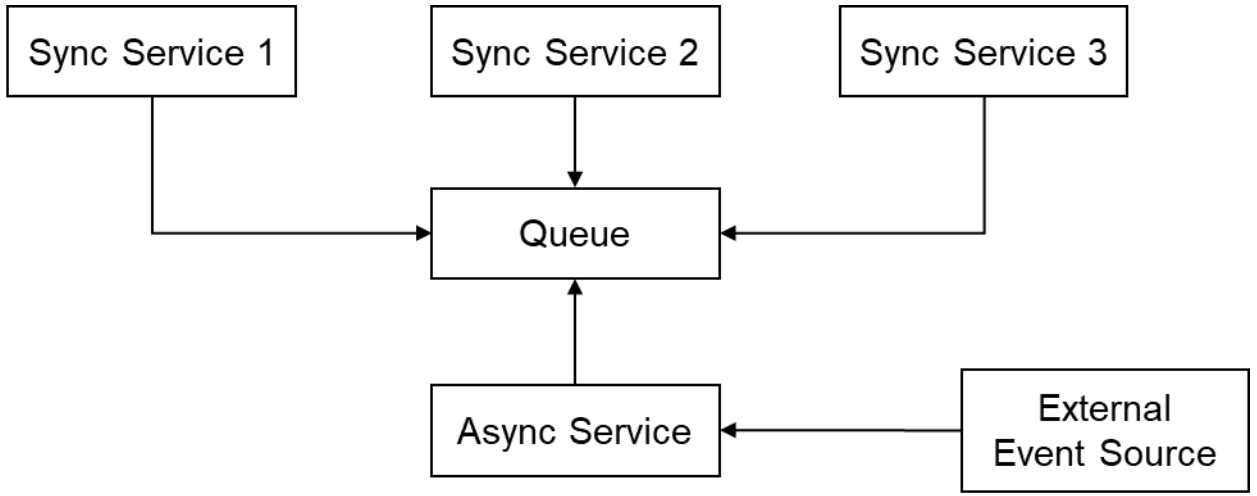


Active object

- **Consequences**
  + Improve concurrency and simplify the complexity of the synchronisation.
  + Exploit the available parallelism in a transparent way.
  + The order of execution of methods can be different than the order of their invocation.
  - Performance overhead
  - Complex to debug

Half-sync/half-async

- **Objective:** Decouple synchronous and asynchronous processing of services in concurrent systems to simplify the development without reducing the performance.
- **Application**: When a concurrent system executes synchronous and asynchronous service that need to communicate.
- **Structure** :



Half-sync/half-async

- **Consequences**
  + Simplification and performance
  + Separation of responsibilities
  + Centralization of inter-layer communication
  - Possibility of penalty for trespassing the borders between layers.
  - The high level services do not always benefit from the efficiency of the asynchronous I/O.

Monitor Object

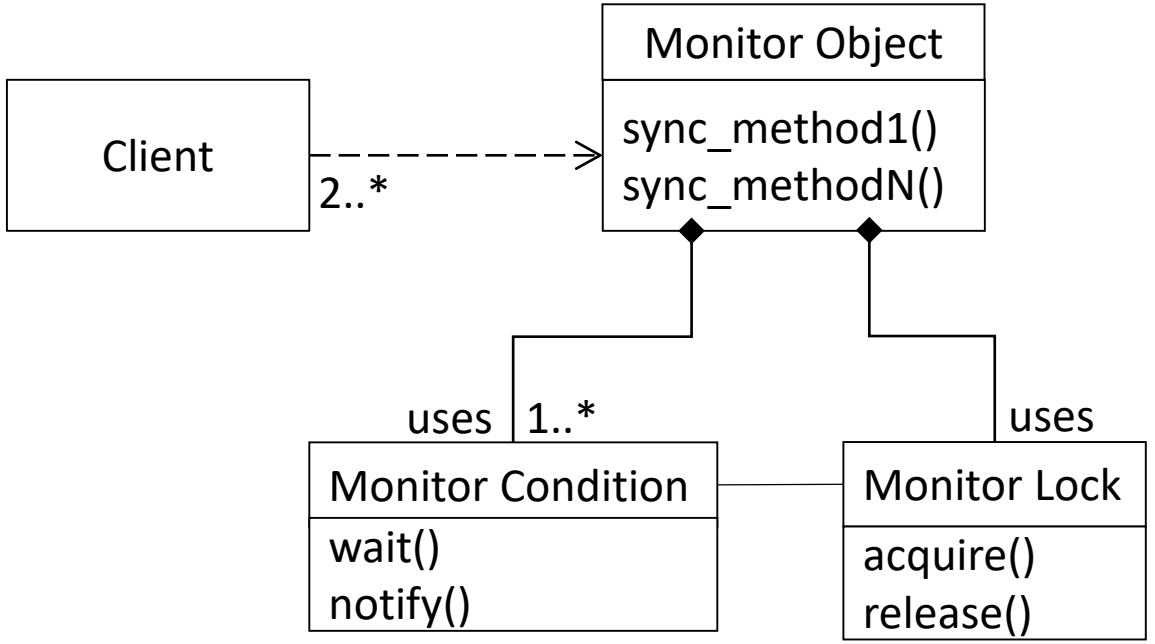
- **Objective**: Synchronise the execution of concurrent methods to ensure that only one method at a time is executed in an object. Also, allow methods of an object to schedule in a collaborative way the order of their execution.
- **Application**: When multiple execution threads access the same object in a concurrent way.
- **Problem**: Many application contain objects, whose methods are invoked concurrently by multiple execution threads. These methods often modify the state their objects. In order for these applications to run correctly, it is necessary to synchronize and schedule the access to objects.

Monitor Object

- **Solution**: Synchronize the access to the methods of an object, so that only one method can be executed at a time. Each object that needs to be accessed in a concurrent way by many client threads is defined as a Monitor Object. Clients can access methods defined by the Monitor Object only through synchronised methods. To avoid race conditions on the internal state of the object, only one synchronised method can be executed at a time in the Monitor Object. The serialisation is implemented using a Monitor Lock. The methods can determine the situations in which they can suspend or resume their execution based on Monitor Conditions.

Monitor Object

- **Structure** :



Monitor Object

Consequences :

+ Simplification of the control of concurrency
+ Simplification of scheduling of method execution
- Extension complexity linked to coupling between the functionality of the object and the methods to synchronize the methods
- Possibility of deadlock in the case of interleaved Monitor Objects.

Leader/Followers

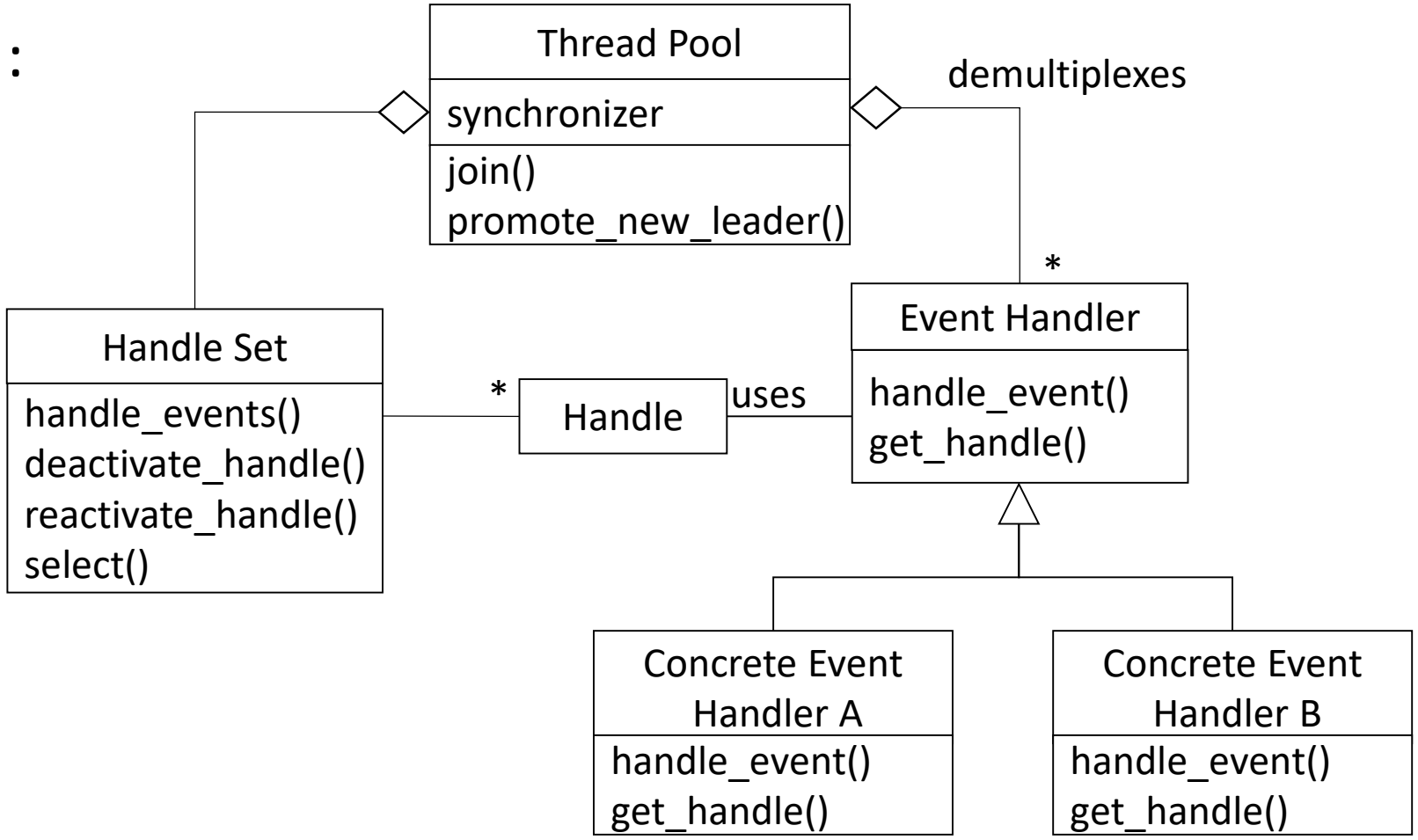
- **Objective**: Architectural pattern that provides an efficient model of concurrency where multiple threads share in turn a set of event sources to detect, demultiplex, submit, and execute service requests that come from an event source.
- **Application**: In an application based on event-driven programming, where a great number of service requests arrive to a set of event sources, that need to be processed efficiently by multiple execution threads that share the event sources.
- **Problem**: The use of multiple execution threads is a modern technique to implement applications that need to process numerous events concurrently. On the contrary, it is difficult to implement high performance servers having multiple execution threads. These applications often need to process great number of events of different types that arrive simultaneously.

Leader/Followers

- **Solution**: Create a queue of execution threads to share efficiently the set of event sources and to demultiplex in turn the events that arrive from these sources, and next send the events synchronously to application services to be processed.

Leader/Followers

- **Structure** :



Leader/Followers

Consequences :

+ Performance improvement:
  - Improve CPU affinity with cache and eliminates dynamic allocation and buffer sharing between threads.
  - Minimize the lock overhead by not exchanging data between threads.
  - Help minimize priority reversals by eliminating extra queues of events.
  - Does not require a change of context to process each event.

+ Simplification of the programming model for processing competing events.

- Implementation complexity: the promotion of leader threads to follower and the reverse must be implemented with atomic operations.

- Lack of flexibility: difficulty of prioritizing events in the absence of a queue.

Thread-Specific Storage

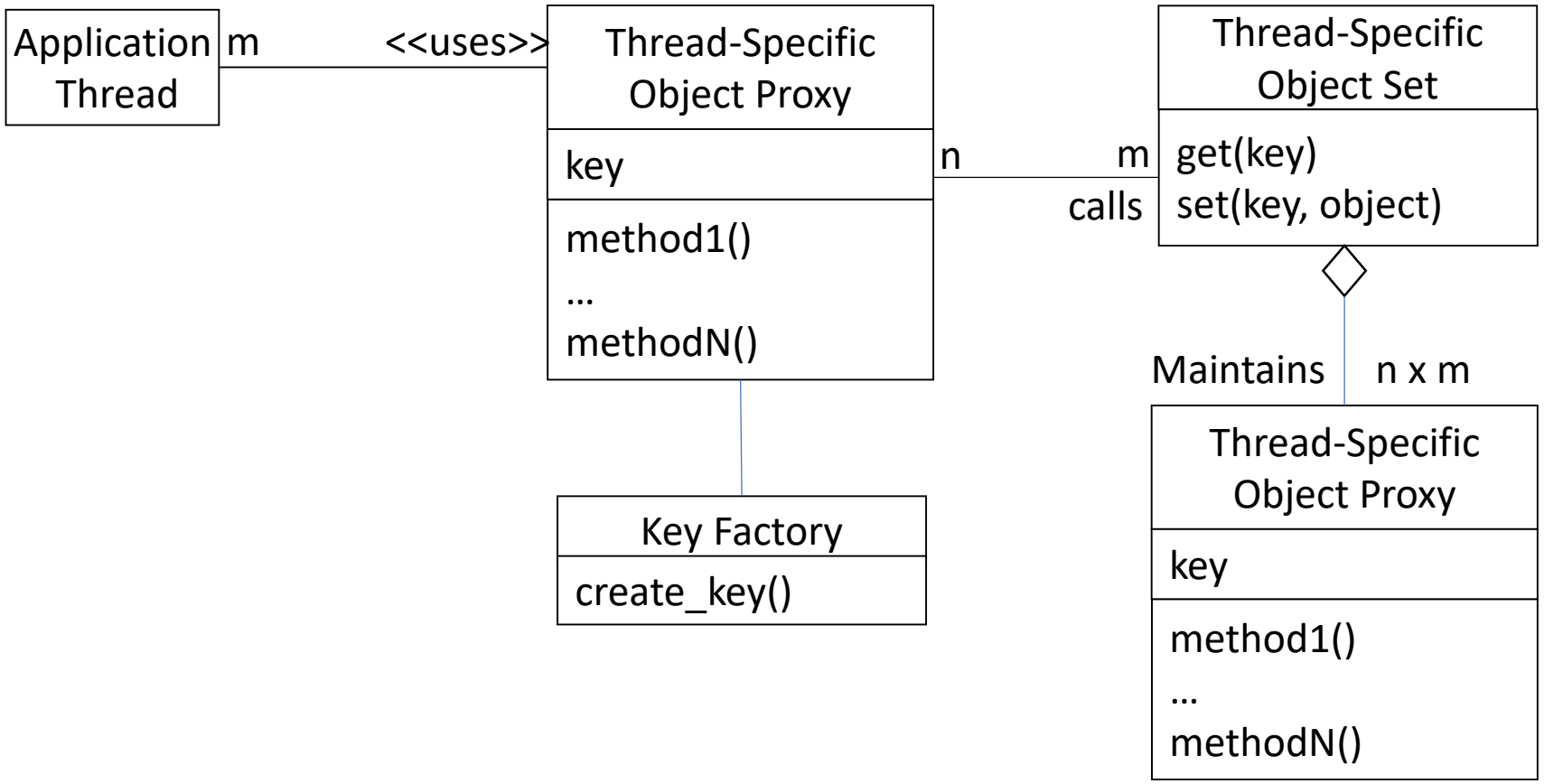
- **Objective:** Design pattern that allows multiple execution threads to use an access point “logically global” to recover a local object into an execution thread, without incurring the locking overhead for each access to the object.
- **Application**: Used in applications having multiple execution threads that need to frequently access data or objects that are stored logically globally, but whose state should be physically local for each execution thread.
- **Problem**: Due to locking overhead, the performance of applications having multiple execution threads is not often better than that of those using a single thread.

Thread-Specific Storage

- **Solution**: Introduce a global access point for each object specific to an execution thread, but maintain the real object in the local storage space of each thread. Ensure that the applications manipulate the objects specific to the execution threads only by using the global access points.

Thread-Specific Storage

- **Structure** :



Thread-Specific Storage

Consequences :

+ Efficiency: can be implemented in way, so tat the locks are not necessary to access data specific to an execution thread.
+ Reusability: this pattern provides code that can be reused in collaboration with other patterns like the Wrapper Façade.
+ Usability: once enxapsulated in a Wrapper Façade, the Thread-Specific Storage is relatively easy to use for the application developers.
+ Portability: the thread-specific storage is available on the majority of operating systems.
- Encourages the use of global objects.
- Camouflages the structure of the system.
- Restraints the implementation options.

Event Handling patterns

1. Reactor
2. Proactor
3. Asynchronous Completion Token
4. Acceptor-Connector

Reactor

- **Objective**: The Reactor architectural pattern allows event-driven applications to demultiplex and to send service requests, which are submitted to an application by many clients.
- **Application**: An event-driven application that receives numerous service requests simultaneously, but it process them synchronously and sequentially.
- **Problem**: The event-driven applications in a distributed system, and particularly servers, need to be ready to process numerous requests simultaneously, even if these requests are ultimately processes sequentially by the application. The arrival of each request is identified by specific *indication* event. Before executing a specific service sequentially, the application needs to demultiplex and send the indication events that arrive concurrently to the appropriate service implementations.

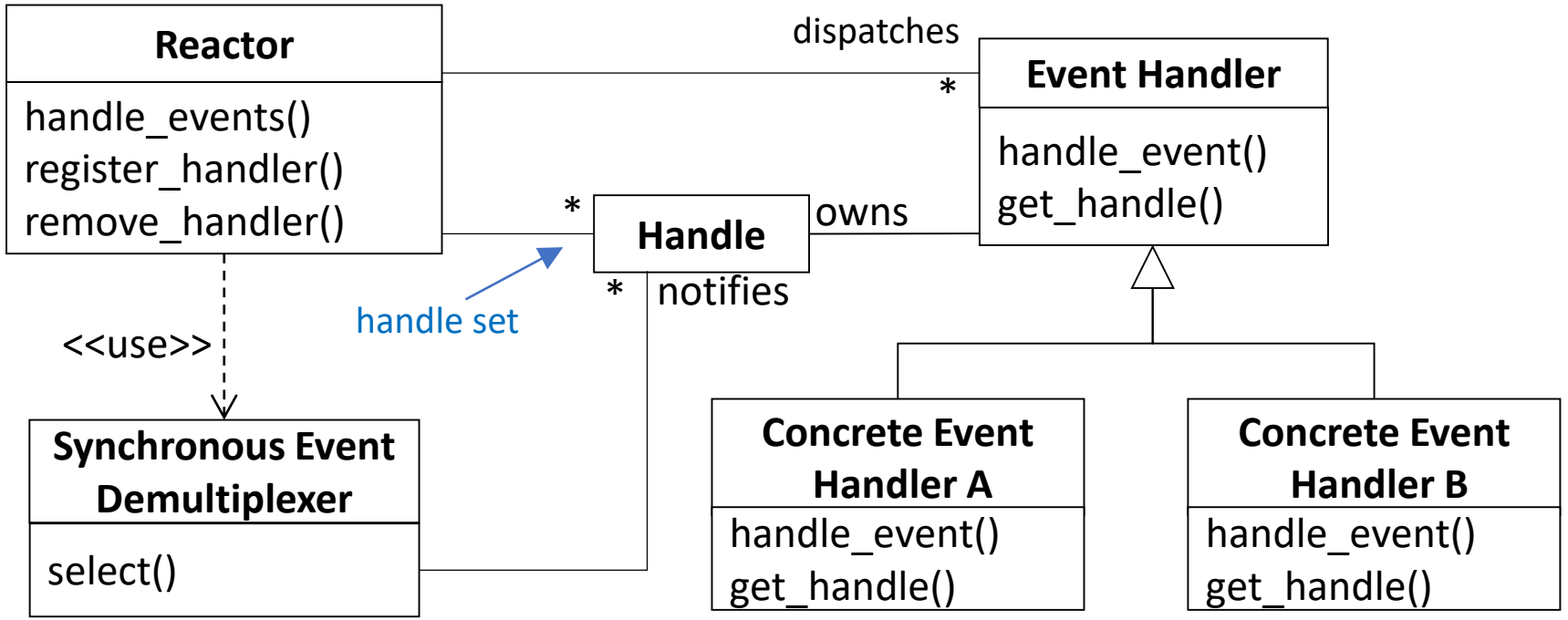
Reactor

- **Solution**: Expect the arrival of indication events in a synchronous manner coming from multiple sources, like for example from socket handles. Integrate a mechanism, that demultiplexes and send the events to services that need to process them. Decouple these demultiplex and send mechanisms from the processing mechanisms.

  For each service that offers an application, introduce a distinct event handler that processes certain type of events coming from certain sources. The event handlers are registered in the Reactor, which uses a synchronous demultiplexer to expect indication events from one or multiple sources. When events arrive, the synchronous demultiplexer informs the Reactor, which triggers the event handler associated with the service so that it can respond to the request.

Reactor

- **Structure** :



Reactor

Consequences :

+ Separation of responsibilities
+ Increased modularity, reusability, configurability and portability.
+ Coarse-grained concurrency control.
- Limited applicability.
- Absence of preemptive action.
- Complex to debug and test.

Proactor

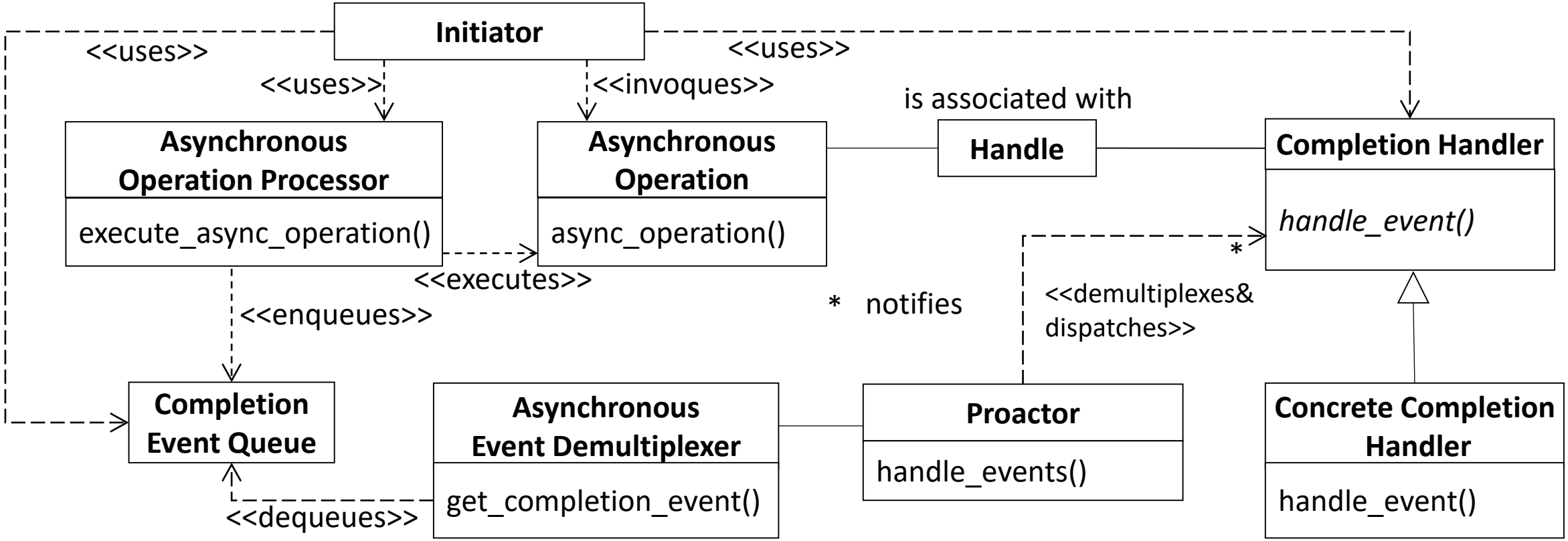
- Objective: The Proactor architectural pattern allows event-driven applications to efficiently demultiplex and send service requests triggered by the completion of asynchronous operations, in order to benefit from performance advantages of concurrency without incurring certain disadvantages.
- Application: An event-driven application that receives and processes numerous service requests asynchronously.
- Problem: The performance of event-driven applications, particularly servers, in a distributed system can often be improved by treating service requests asynchronously. When the asynchronous processing is completed, the applications need to handle the termination events transmitted by the operating system to indicate the end of asynchronous calculations.

Proactor

- **Solution:** Separate application services into two parts: 1) long operations that run asynchronously and 2) the termination handler that processes the results of these operations when they are complete. Integrate the concepts of demultiplexing of the termination events, which are sent when the operation is complete, and the triggering of the handler processing the results, but decouple the respective concrete mechanisms related to specific applications.
- For each service offered by an application, introduce asynchronous operations that initiate the processing of service requests proactively, via an identifier (handle), together with a termination handler that processes the terminating events containing the results of these asynchronous operations. An asynchronous operation is triggered in an application by an initiator to, for example, accept incoming connection requests from remote applications. The operation is performed by an asynchronous operation processor. When an operation terminates, the asynchronous operation processor inserts a termination event containing the processing results into a termination event queue.
- This queue is monitored by an asynchronous demultiplexer of events, called by a Proactor. When the demultiplexer removes an event from the queue, the Proactor demultiplexes and sends the event to the specific termination handler for the application. This handler processes the results and can trigger other asynchronous operations in the same scheme.

Proactor

- **Structure** :



owns

Proactor

Consequences :

+ Separation of responsibilities
+ Portability
+ Encapsulation of concurrency mechanisms.
+ Decoupling of execution thread from concurrency.
+ Increased performance.
+ Simplification of the synchronisation of applications.
- Limited applicability
- Complex to debug and test.
- Scheduling, control and cancelation of running operations asynchronously.

Asynchronous Completion Token (ACT)

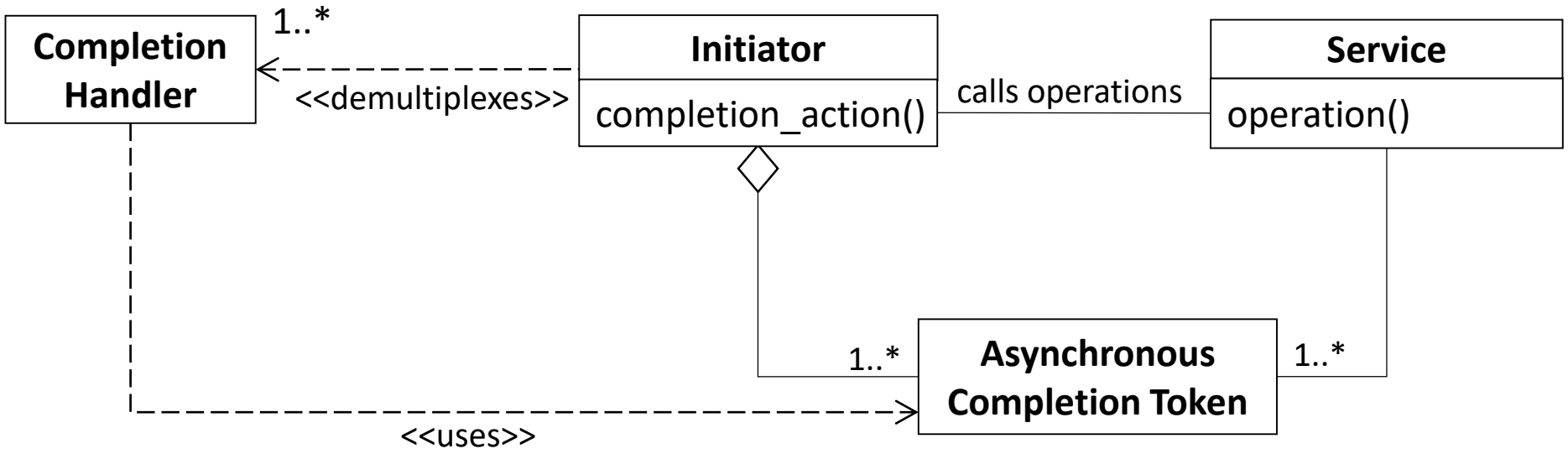
- **Objective**: The Asynchronous Completion Token (ACT) design pattern allows an application to efficiently demultiplex and process responses returned by asynchronous operations that it has invoked from services.
- **Application**: An event-driven system in which applications invoke asynchronous operations from services and, next, they need to process the associated termination events.
- **Problem**: When a client application invokes an operation from one or more services asynchronously, each service returns its response to the application by a termination event. So, the application needs to demultiplex the events towards appropriate handlers (function or object) that it uses to process the operation result, contained in the termination event.

Asynchronous Completion Token (ACT)

- **Solution:** In association with each asynchronous operation that an initiator client invoked from a service, transmit the information that identifies how the initiator should process the response of the service. Return this information to initiator when the operation is complete so that it is used to demultiplex the response efficiently, thus permitting the initiator to process it.
- For each asynchronous operation, create an ACT. This ACT contains the information that identifies uniquely the termination handler, that is the function or the object to process the response. Pass the ACT to the service with the operation, that keeps but does not modify the ACT. When the service responds to the initiator, the response includes the ACT. The initiator can then use the ACT to identify the termination handler that needs to process the response.

Asynchronous Completion Token (ACT)

- **Structure** :



Asynchronous Completion Token (ACT)

Consequences :

+ Simplification of the initiator’s data structures.
+ Efficiency in acquiring the state.
+ Space efficiency
+ Flexibility.
+ Non-dictatorial concurrency policies
- Possibility of memory leaks.
- Authentication.
- Invalidation due to displacements in memory.

Acceptor-Connector

- **Objective**: The Acceptor-Connector design pattern decouples the connection and initialisation part from the processing part in a distributed system of peer-to-peer services that collaborate.
- **Application**: An application or a distributed system in which connection-oriented protocols are used to communicate between connected pairs of services by transport termination points.
- **Problem**: The applications in a connection-oriented distributed system often contain a significant quantity of configuration code to establish the connection and initialise the services. This configuration code is largely independent from the processing performed by the services over the data exchanged between transport termination points. Closely coupling the configuration code with the processing code is not desirable.

Acceptor-Connector

Solution: Decouple the connection and initialization part of the peer services from the processing part that these services perform once they are connected.

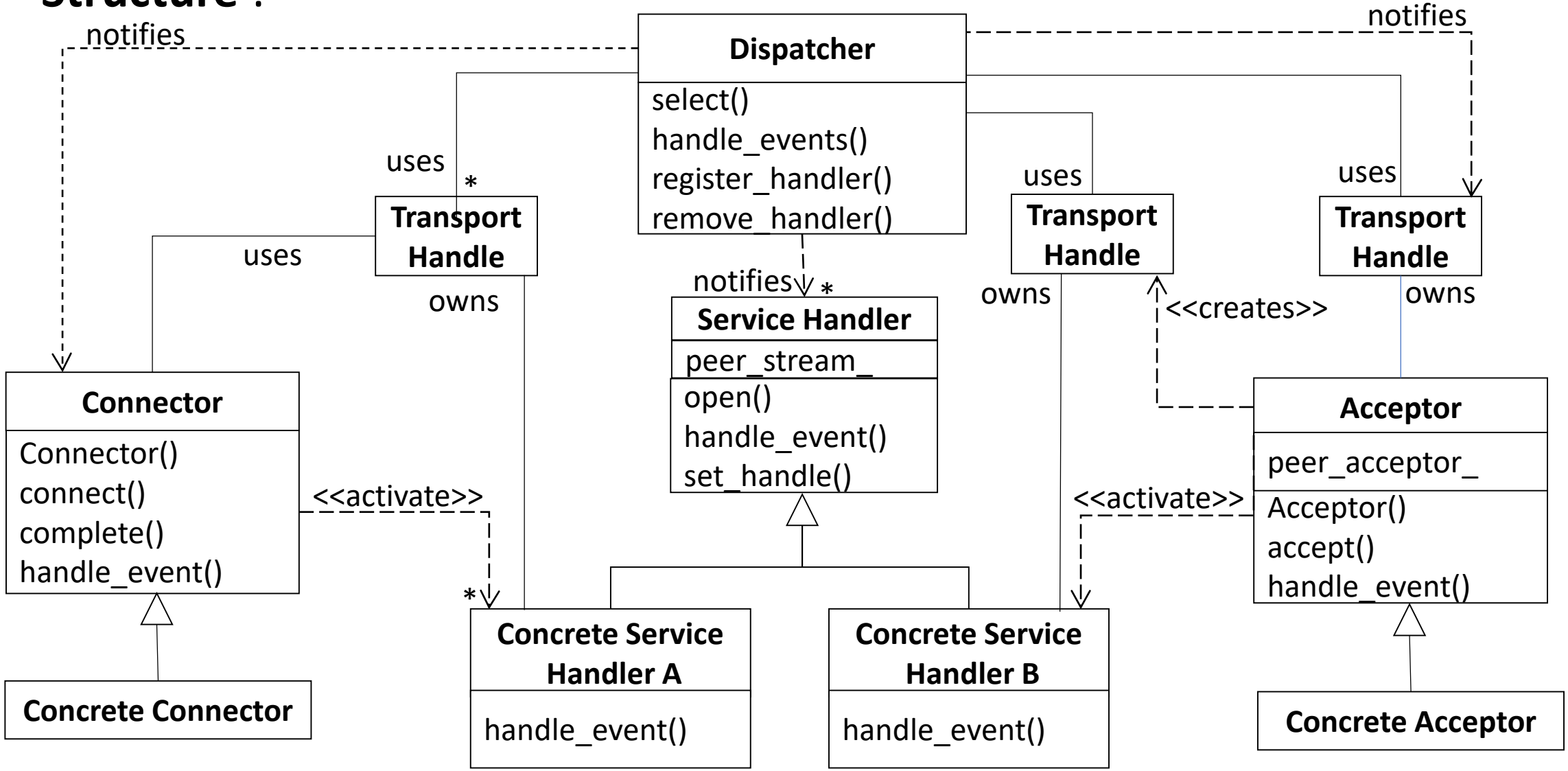
Encapsulate application services in peer service managers. Each service manager implements half of a point-to-point service in a distributed application. Connect and initialize peer service managers using two factories: Acceptor and Connector. The two factories work together to create the complete association between two service managers and the two transport endpoints by which they are connected, each encapsulated in a transport manager.

The Acceptor factory makes passive connections to the name of the service manager with which it is associated when a connection request from a remote service manager arrives. Similarly, the Connector factory actively connects to a designated remote service on behalf of the service manager with which it is associated.

Once the connection is established, the Acceptor and Connector initializes the service managers associated with them and passes them the transport identifiers. The service managers then perform the application processing, using the transport identifiers to exchange data through the connection. In general, service managers no longer interact with Acceptor and Connector plants once they are connected and initialized.

Acceptor-Connector

- **Structure** :



Acceptor-Connector

Consequences :

+ Efficiency.
+ Reusability, portability, extensibility
+ Robustness.
- Additional indirection
- Additional complexity