



Concurrency Patterns

RAINER GRIMM



20
21



Definition

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." (Christopher Alexander)

Three Types of Patterns

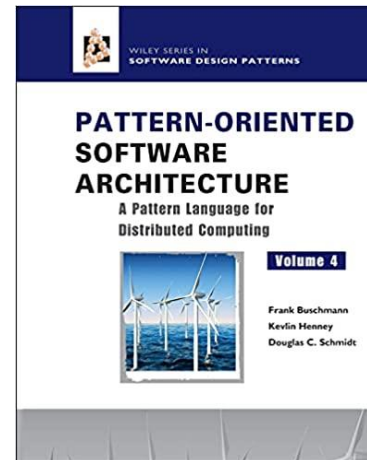
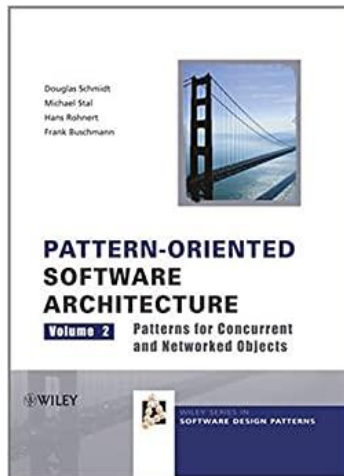
- Architecture pattern
 - Fundamental structure
 - Software system
- Design pattern
 - Interplay of components
 - Focus on a subsystem
- Idiom
 - Implementation of an architecture or design pattern in a concrete programming language.

Components of a Pattern

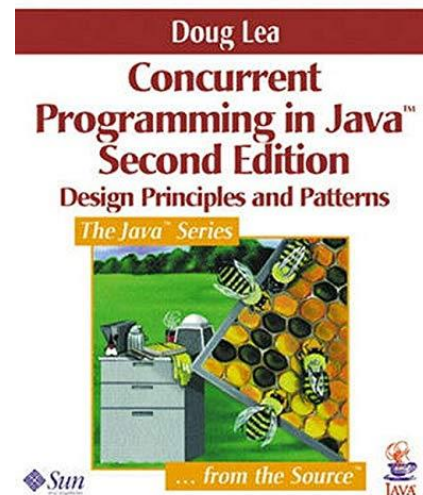
1. Name
2. Also known as
3. Summary
4. Motivation
5. Context
6. Interaction
7. Solution
8. Example
9. Consequenses
10. Related pattern
11. Known usages

Concurrency Patterns

- [Pattern-Oriented Software Architecture \(Volume 2 and 4\)](#)



- [Concurrent Programming in Java](#)



Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor Object

Reactor

Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor Object

Reactor

Copied Value

There is no need to synchronize when a thread takes its arguments by copy and not by reference.

➡ Data races or lifetime issues are not possible.

Thread-Specific Storage

Thread-specific storage enables global state within a thread.

- Typical use-cases:
 - Porting a single-thread to multithreaded program
 - Compute thread-local and share the results
 - Thread-local logger

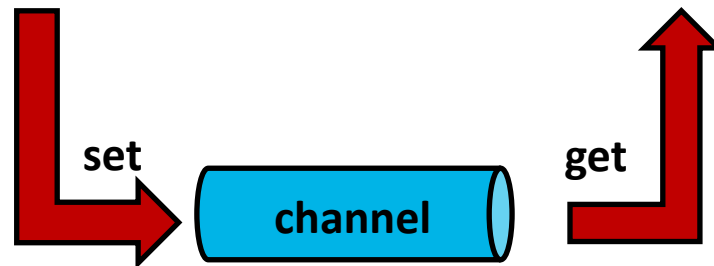
Future

A future is a non-mutable placeholder for a value, which is set by a promise.

```
auto future = std::async([]{ return "LazyOrEager"; });  
future.get();
```

Promise: sender

Future: receiver



Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor
Object

Reactor

Scoped Locking

Scoped Locking is RAII applied to locking.

- Idea:
 - Bind the acquiring (constructor) and the releasing (destructor) of the resource to the lifetime of an object.
 - The lifetime of the object is bound.
 - The C++ run time is responsible for invoking the destructor and releasing the resource.
- C++ Implementation
 - `std::lock_guard` and `std::scoped_lock`
 - `std::unique_lock` and `std::shared_lock`

Strategized Locking

Strategized Locking

- Enables it to use various locking strategies as replaceable components.
- Is the application of the strategy pattern to locking.
- Idea:
 - You want to use your library in various domains.
 - Depending on the domain, you want to use exclusive locking, shared locking, or no locking.
 - Configure your locking strategy at compile time or run time.

Strategized Locking

Advantages:

- Run-time polymorphism
 - Enables it to change the locking strategy during run time.
- Compile-time polymorphism
 - No cost at run time
 - Flatter object hierarchies

Disadvantages:

- Run-time polymorphism
 - Needs a pointer indirection.
- Compile-time polymorphism
 - Produces in the error case a quite challenging to understand error message (when no concepts are used).

[strategizedLockingRuntime.cpp](#)

[strategizedLockingCompileTimeWithConcepts.cpp](#)

Thread-Save Interface

The thread-save interface extends the critical region to an object.

- Antipattern: Each member function uses a lock internally.
 - The performance of the system goes down.
 - Deadlocks appear when two member functions call each other.

Thread-Save Interface

A deadlock due to entangled calls.

```
struct Critical{
    void method1() {
        std::lock_guard(mut);
        method2();
        . . .
    }
    void method2() {
        std::lock_guard(mut);
        . . .
    }
    std::mutex mut;
}

int main() {
    Critical crit;
    crit.method1();
}
```


Thread-Save Interface

- Solutions:
 - All interface member functions (`public`) use a lock.
 - All implementation member functions (`protected` and `private`) must not use a lock.
 - The interface member functions call only implementation member functions.

Guarded Suspension

A guarded suspension consists of a lock and a condition, which has to be fulfilled by the calling thread.

- The calling thread will put itself to sleep if the condition is not meet.
- The calling thread uses a lock when it checks the condition.
- The lock protects the calling thread from a data race or deadlock.

Guarded Suspension

- Guarded suspension is available in many variations.
 - The waiting thread is notified about the state change or asks for the state change.
 - Push principle: condition variables, future/promise pairs, atomics (C++20), or semaphores (C++20)
 - Pull principle: not natively supported in C++
 - The acquiring thread tries it only once.
 - Not natively supported, but can be implemented using condition variables, future/promise pairs, or atomics (C++20)
 - The waiting thread waits with or without a time limit.
 - Condition variables, or future/promise pairs
 - The notification is sent to one or all waiting threads.
 - Shared futures, condition variables, atomics (C++20), or semaphores (C++20)

Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor
Object

Reactor

Active Object

The active object pattern separates the method execution from the method invocation.

- Each object owns its own thread.
- Each method invocation is stored in an activation list.
- A scheduler triggers the method execution.

Active Object

Proxy:

- Proxy for the member functions on the active object
- Triggers the construction of a request object which goes to the activation list and returns a future.
- It runs in the client thread.

Method Request

- Includes all context information to be executed later.

Activation List:

- Has the pending requests objects.
- Decouples the client from the Active Object thread.

Scheduler:

- Runs in the thread of the Active Object.
- Decides which request from the Activation List is executed.

Active Object

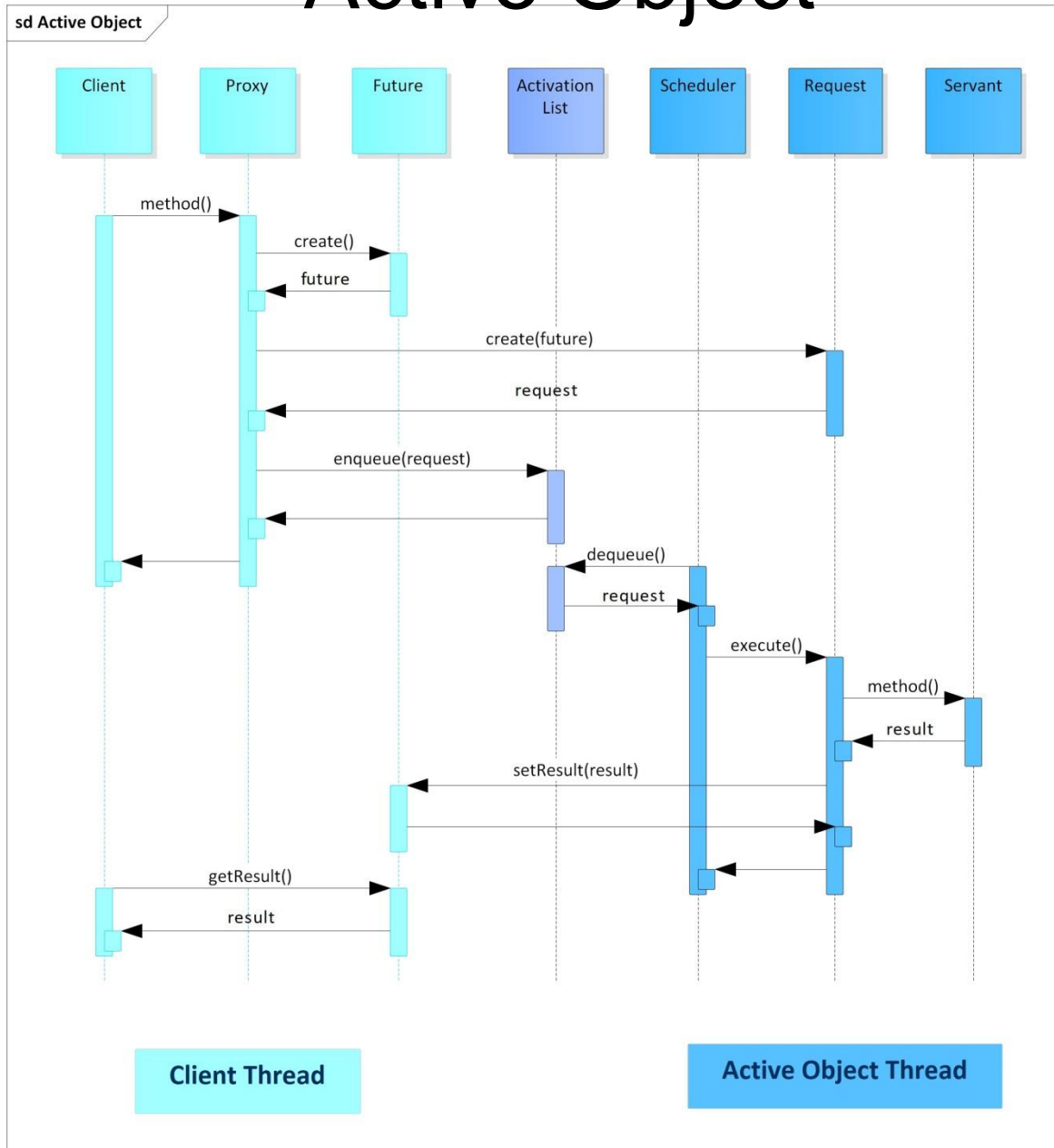
- **Servant:**
 - Implements the member functions of the active objects.
 - Supports the interface of the Proxy.
- **Future:**
 - Is created by the Proxy.
 - Is only necessary if the request objects returns a result.
 - The client uses the future to get the result of the request object.

Active Object

Dynamic Behavior

1. Request construction and scheduling:
 - The client invokes the method on the proxy.
 - The proxy creates a request and passes it to the scheduler.
 - The scheduler enqueues the request on the activation list and returns a future to the client if the request returns something.
2. Member function execution
 - The scheduler determines which request becomes runnable.
 - It removes the request from the activation list and dispatches it to the servant.
3. Completion:
 - Stores eventually the result of the request object in the future.
 - Destructs the request object and the future when the client has the result.

Active Object



Active Object

Advantages:

- Only the access to the activation list has to be synchronized
- Clear separation of client and server
- Improved throughput due to the asynchronous execution
- The scheduler can use various execution policies.

Disadvantages:

- If the member function execution is too fine-grained, the indirection may cause significant overhead.
- The asynchronous member function execution and the various execution strategies make the system quite difficult to debug.

Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor Object

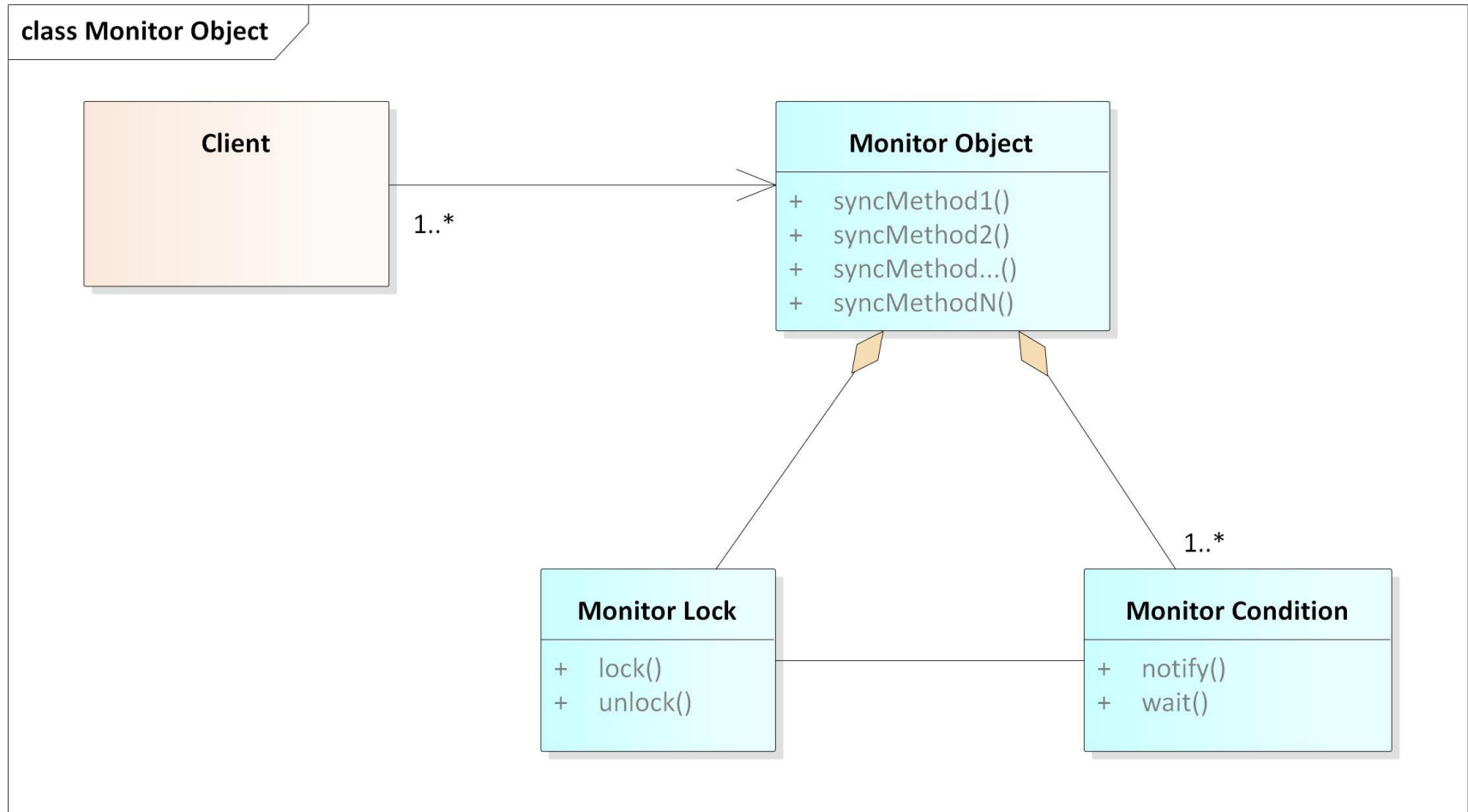
Reactor

Monitor Object

The monitor object synchronizes the access to an object so that at most one member function can run at any moment in time.

- Each object has a monitor lock and a monitor condition.
- The monitor lock guarantees that only one client can execute a member function of the object.
- The monitor condition notifies the waiting clients.

Monitor Object



Monitor Object

Monitor Object:

- Support member functions, which can run in the thread of the client.

Synchronized Methods:

- Interface member functions of the monitor object.
- At most, one member function can run at any point in time
- The member functions should apply the thread-safe interface pattern.

Monitor Lock:

- Each monitor object has a monitor lock.
- Guarantees exclusive access to the member functions.

Monitor Condition:

- Allows various threads to store their member function invocation.
- When the current thread is done with its member function execution, the next thread is awoken.

Monitor Object

Advantages:

- The synchronization is encapsulated in the implementation.
- The member function execution is automatically stored and performed.
- The monitor object is a simple scheduler.

Disadvantages:

- The synchronization mechanism and the functionality are strongly coupled and can, therefore, not easily be changed.
- When the synchronized member functions invoke an additional member function of the monitor object, a deadlock may happen.

[monitorObject.cpp](#)

[monitorObjectCpp20.cpp](#)

Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor
Object

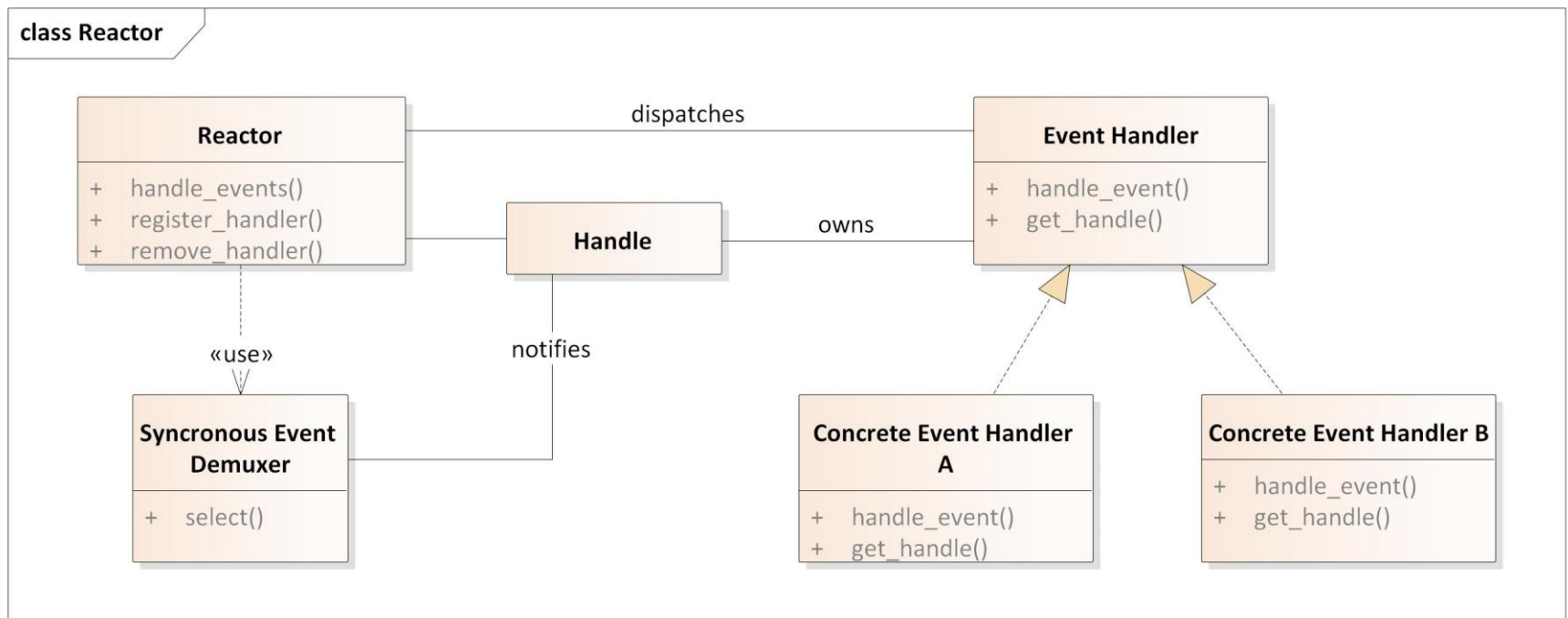
Reactor

Reactor

The Reactor pattern is an event-driven framework to demultiplex and dispatch service requests concurrently onto various service providers.

- For each supported service type implement an event handler that fulfils the specific client request.
- Register this service handler within the Reactor.
- The Reactor uses an event demultiplexer to wait synchronously on all incoming events.
- When an event arrives, the Reactor is notified and dispatches it to the specific service.

Reactor



Reactor

Advantages:

- A clear separation of framework and application logic.
- The Reactor can be ported to various platforms, because the underlying event demultiplexing functions are widely available.
- The separation of interface and implementation enables easy adaption or extension of the services.
- Overall structure supports the concurrent execution.

Disadvantages:

- Requires an event demultiplexing system call.
- A long-running event handler can block the Reactor.
- The inversion of control makes testing and debugging more difficult.

Reactor

```
rainer : bash — Konsole <3>  
File Edit View Bookmarks Settings Help  
rainer@seminar:~> reactor  
^C  
rainer@seminar:~> █
```

```
rainer : bash — Konsole  
File Edit View Bookmarks Settings Help  
rainer@seminar:~> more reactorOutput.txt  
Rainer1  
Grimm1  
rainer@seminar:~> █
```

```
rainer : bash — Konsole <2>  
File Edit View Bookmarks Settings Help  
rainer@seminar:~> telnet 127.0.0.1 4711  
Trying 127.0.0.1...  
Connected to 127.0.0.1.  
Escape character is '^]'.  
Rainer  
Rainer  
Grimm  
Grimm  
Connection closed by foreign host.  
rainer@seminar:~> █
```

```
rainer : bash — Konsole <2>  
File Edit View Bookmarks Settings Help  
rainer@seminar:~> telnet 127.0.0.1 4712  
Trying 127.0.0.1...  
Connected to 127.0.0.1.  
Escape character is '^]'.  
Rainer1  
Grimm1  
Connection closed by foreign host.  
rainer@seminar:~> █
```

Concurrency Patterns

Synchronization Patterns

Dealing with Sharing

- Copied Value
- Thread-Specific Storage
- Future

Dealing with Mutation

- Scoped Locking
- Strategized Locking
- Thread-Safe Interface
- Guarded Suspension

Concurrent Architecture

Active Object

Monitor
Object

Reactor

Concurrency Patterns

- [Concurrency with Modern C++ \(50 % off during the CppCon2021\)](#)

Concurrency with Modern C++

What every professional C++ programmer should know about concurrency.



**Rainer
Grimm**

ModernesCpp.com

```
#include <iostream>
```

```
int main(){
```

```
    std::cout << "myVec: ";
```

```
    std::vector<int> myVec(20);
```

```
    std::iota(myVec.begin(), myVec.end(), 0);
```

```
    std::cout << "myVec: ";
```

```
    for ( auto i: myVec) std::cout << i;
```

```
    std::cout << "\n\n";
```

```
    std::function< bool(int)> myBindPred = bind( std::logical_not< bool>(),
```

```
    myVec.erase(std::remove_if(myVec.begin(), myVec.end(), myBindPred),
```

```
    std::cout << "myVec: ";
```

```
    for ( auto i: myVec) std::cout << i;
```

```
    std::cout << "\n\n";
```

```
    std::vector<int> myVec2(20);
```

```
    std::iota(myVec2.begin(), myVec2.end(), 0);
```

```
    std::cout << "myVec2: ";
```

```
    for ( auto i: myVec2) std::cout << i;
```

```
    std::cout << "\n\n";
```

www.ModernesCpp.com

Rainer Grimm

Training, Coaching, and
Technology Consulting

www.ModernesCpp.net