Design Patterns 448.058 (VO)

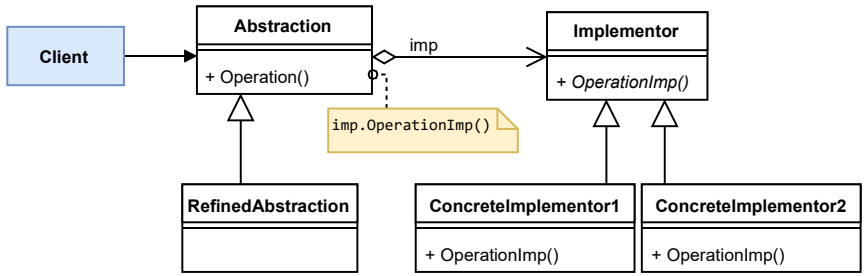
Michael Krisper
Georg Macher

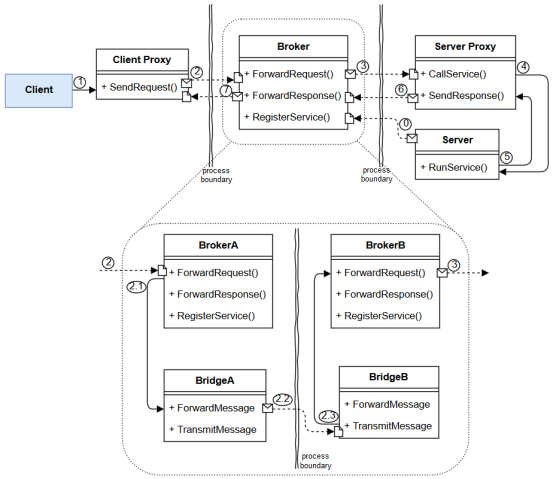
20.11.2019

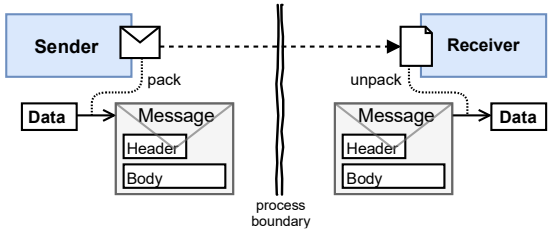
Revision from last time…

Communication Patterns:

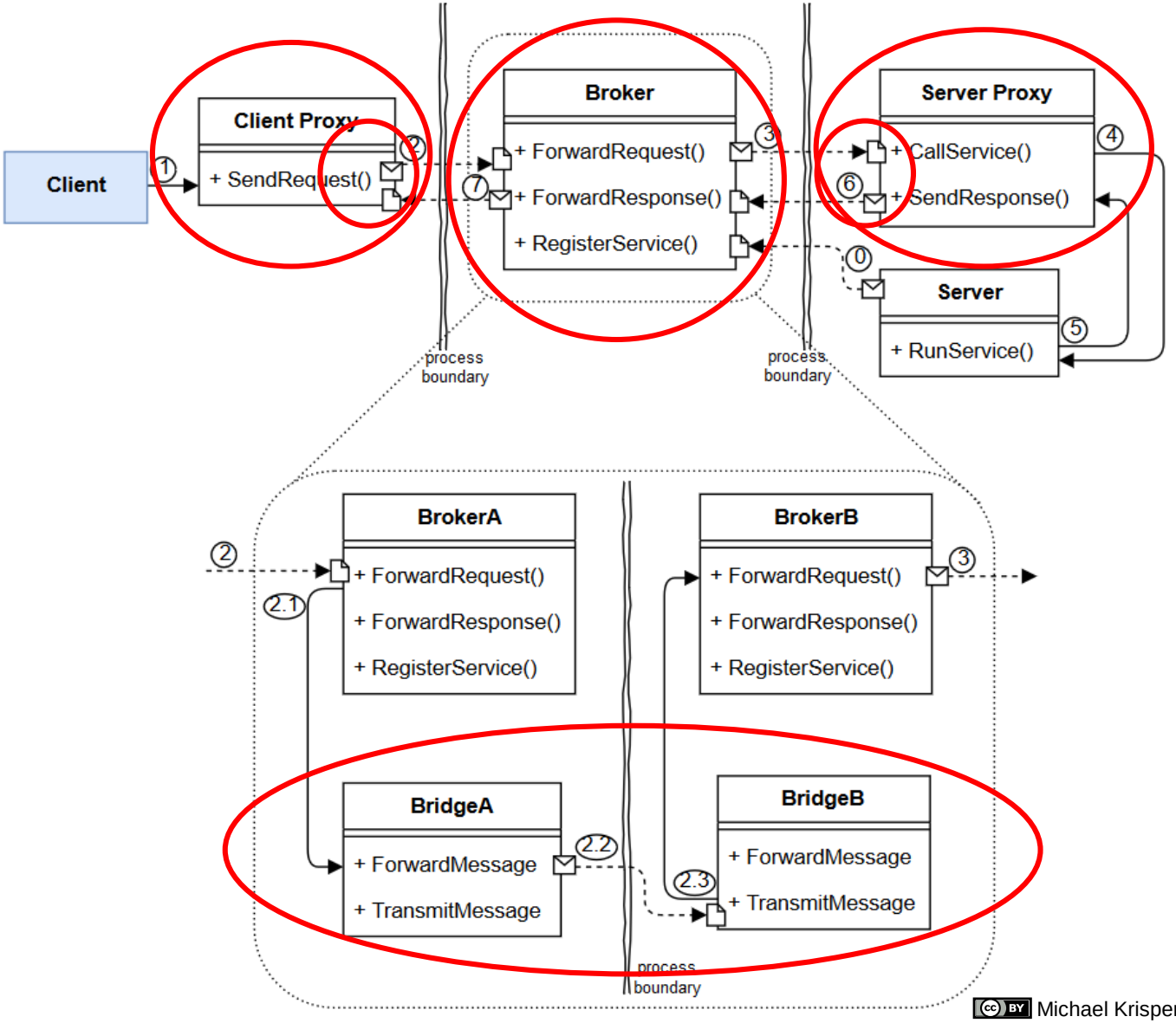
- Mediator
- Blackboard
- Microkernel
- Bridge
- Broker
- Messages
- Message Endpoint
- Message Translator
- Message Router
- Request Handler
- Requestor







Messaging in Broker



Learning Goals for Today

Collection-Patterns

- Iterator (revision)
- **Visitor**
- **Composite / Whole-Part**

Behavioral-Patterns

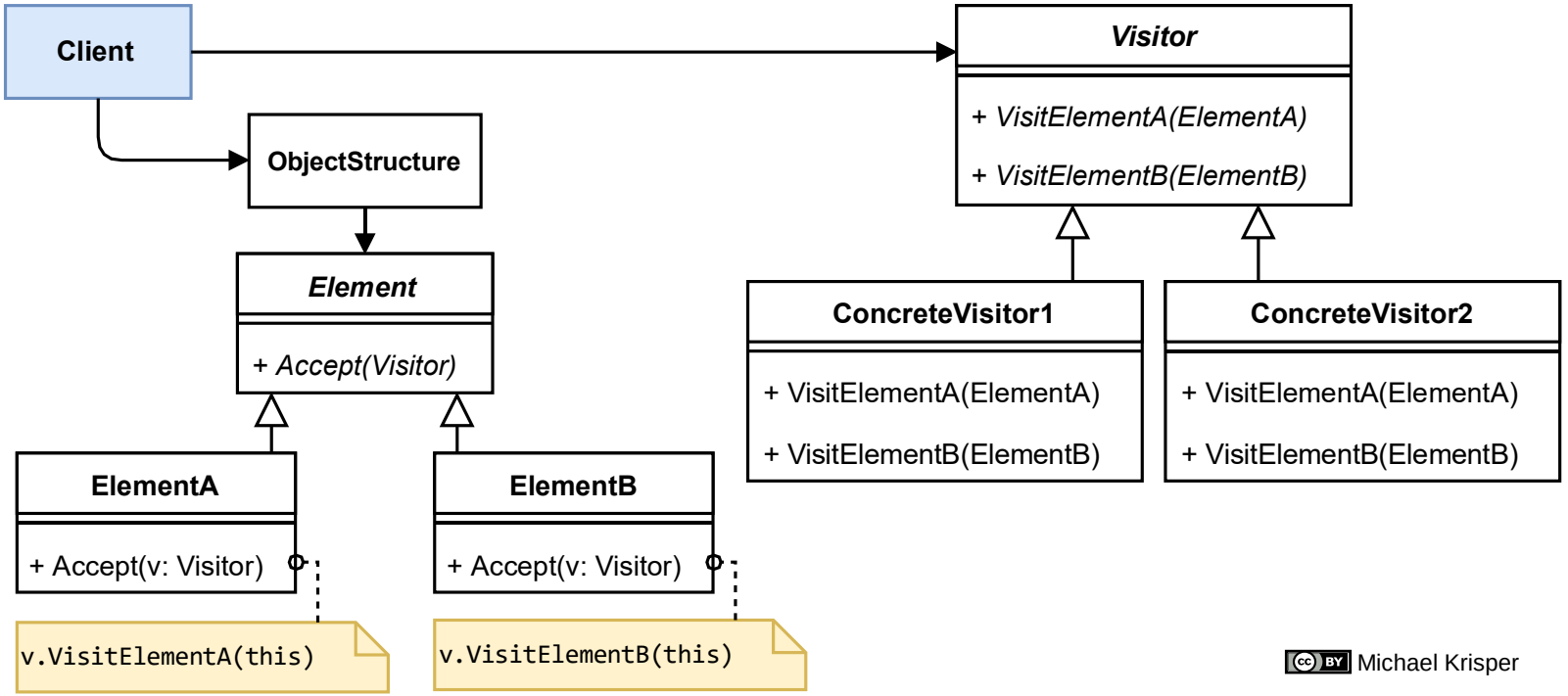
- **State**
- Template-Method

Synchronisation / Concurrency Patterns

- **Locks**: Mutex, Semaphor
- **Monitor**
- **Active Object**
- **Future**
- **Scoped Locking**
- Thread-Specific Storage
- Proactor / Reactor
- Double Checked Locking

Visitor

Add behaviour on aggregates of different objects



Visitor

Context: Performing operations on elements of an aggregate.

Problem: How to execute some behaviour on an aggregate of different objects?

Forces:

- Object aggregate contains different interfaces
- Avoid polluting classes with unrelated operations
- Structure rarely changes

Solution:

- Pack related operations together in a visitor and implement one for every needed object type
- Implement means to iterate over aggregate and apply visitor to every contained object.

Consequences:

\+ Makes adding new functionality easy

\+ Combines related functions

\+ Account for different object types

\+ Can accumulate state

~ Who traverses the aggregate? How?

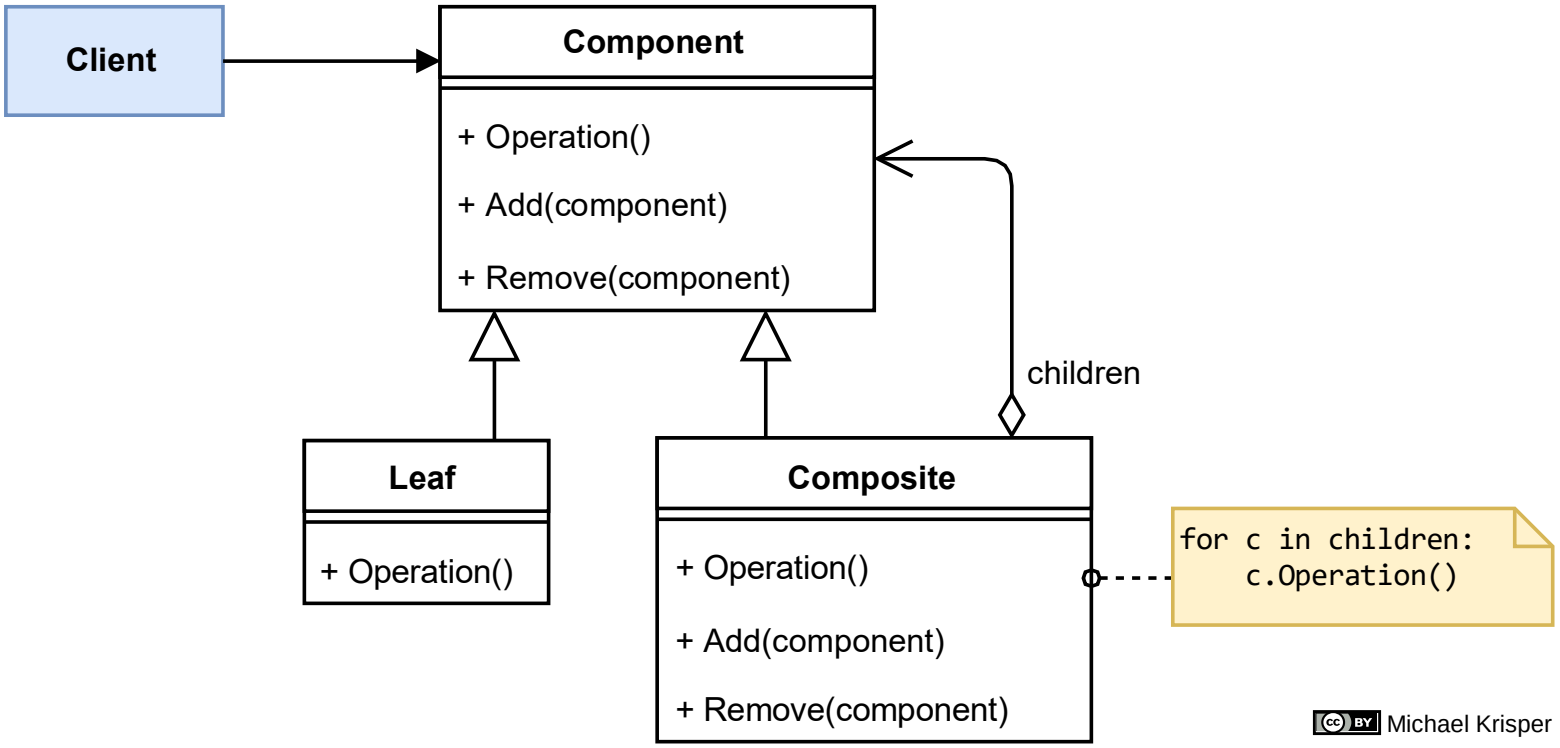
~ Double-dispatch or not?

\- Adding new class types is expensive

\- Visitor may need access to private members (breaks encapsulation)

Composite

Handle different granularities of objects uniformly



Composite

Context:

Hierarchies of objects with different granularities

Problem:

How to uniformly handle different granularities of objects in hierarchies?

Forces:

- Represent hierarchies of objects
- Treat all objects uniformly
- Ignore differences behaviour of individual objects and aggregates

Solution:

- Define common Interface for all granularities to manage children and call methods.
- Implement Composites: Forward call to children
- Implement Leaves: Execute calls directly

Consequences:

\+ Defines hierarchies of primitive and composite objects

\+ Simple handling for client

\+ Adding new kinds of composites is easy

~ Default implementations?

~ Parent references?

~ Changing roles? Leaf ↔ Composite?

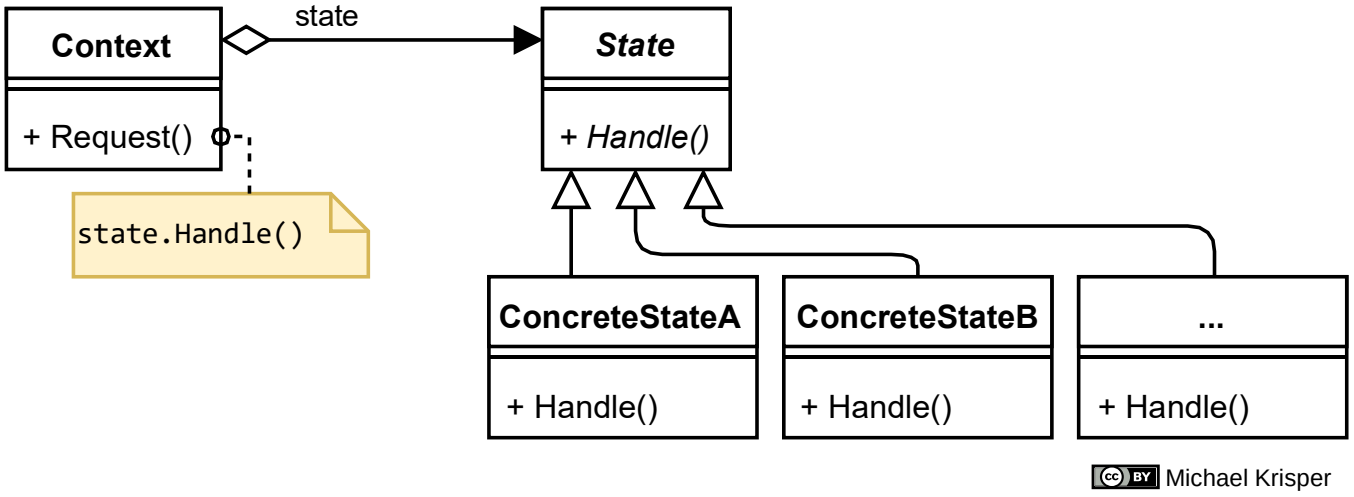
~ Caching?

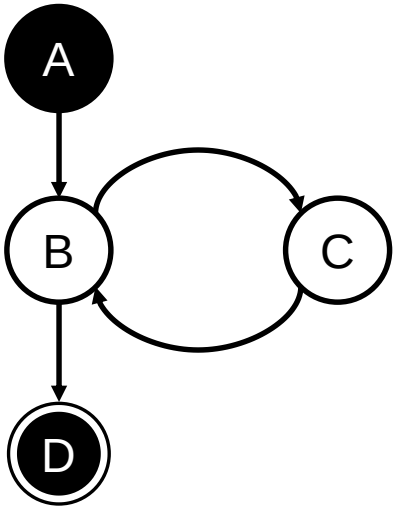
\- Constraints are difficult to implement

\- Interface limits functionality

State

Change object behaviour depending on a situation





State

Context: Objects which change their behaviour according to a situation

Problem: How to switch behaviour of an object without complex implementation?

Forces:

- Behaviour should change with internal state
- Behaviour should change at runtime
- Transition between states should not depend on complex multipart conditional statements (no if-else-if-else-…)
- States should not be mixed.

Solution:

- Define Context(manager) which knows the states and transitions and exposes the client-interface
- Define general Interface for all States
- Implement the different states in individual classes
- Define the transitions between states

Consequences:

\+ State specific behaviour is encapsulated within the state objects.

\+ New States and transitions can easily be defined

\+ Transition logic is partitioned and simple.

\+ Transition are explicit – no mixed states

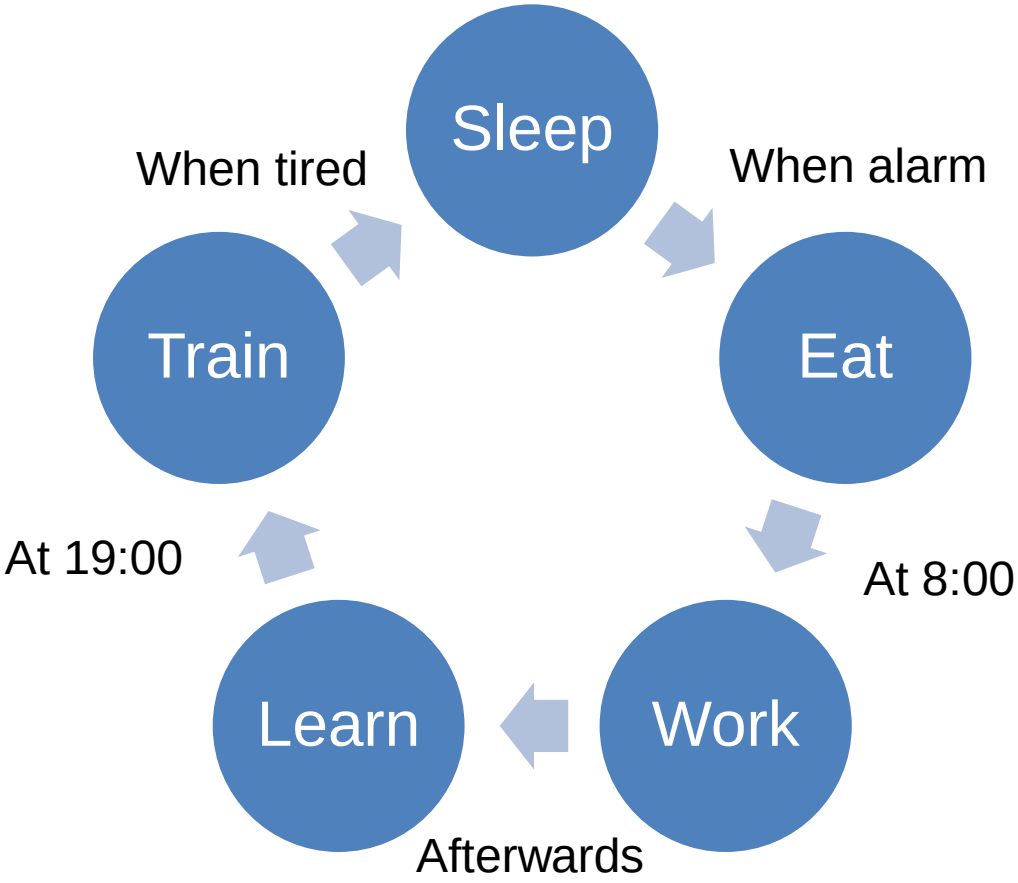
\+ State Object can be shared (-> Flyweight)

~ Who makes the transitions?

\- More classes

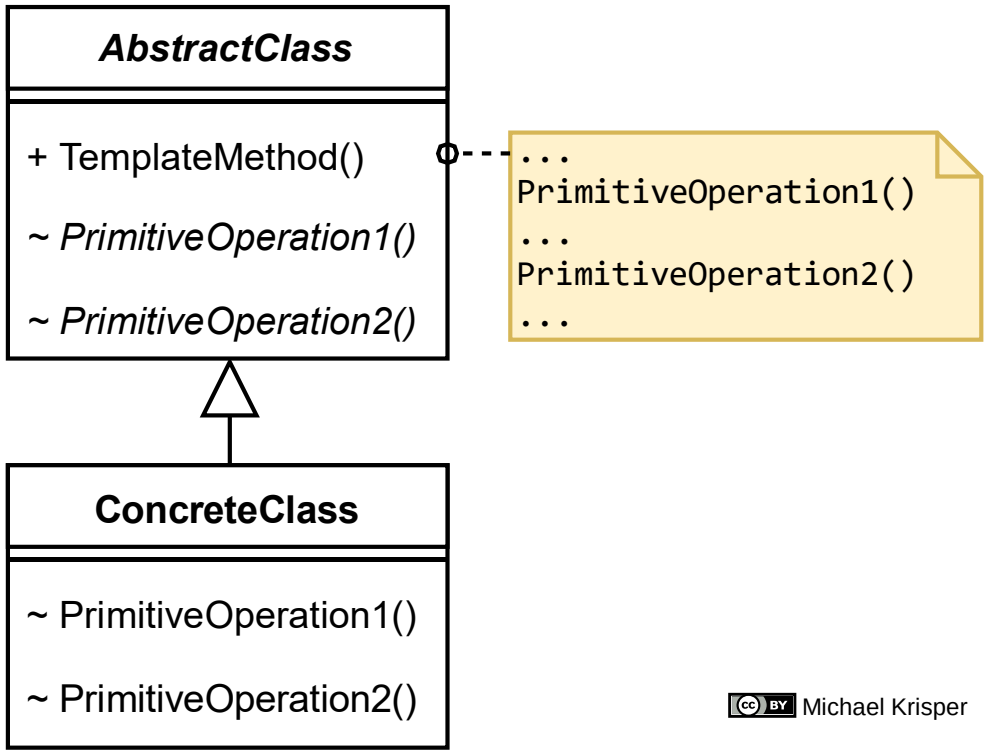
\- Special transitions may be difficult

State



Template-Method

Define methods and let children implement the behaviour.



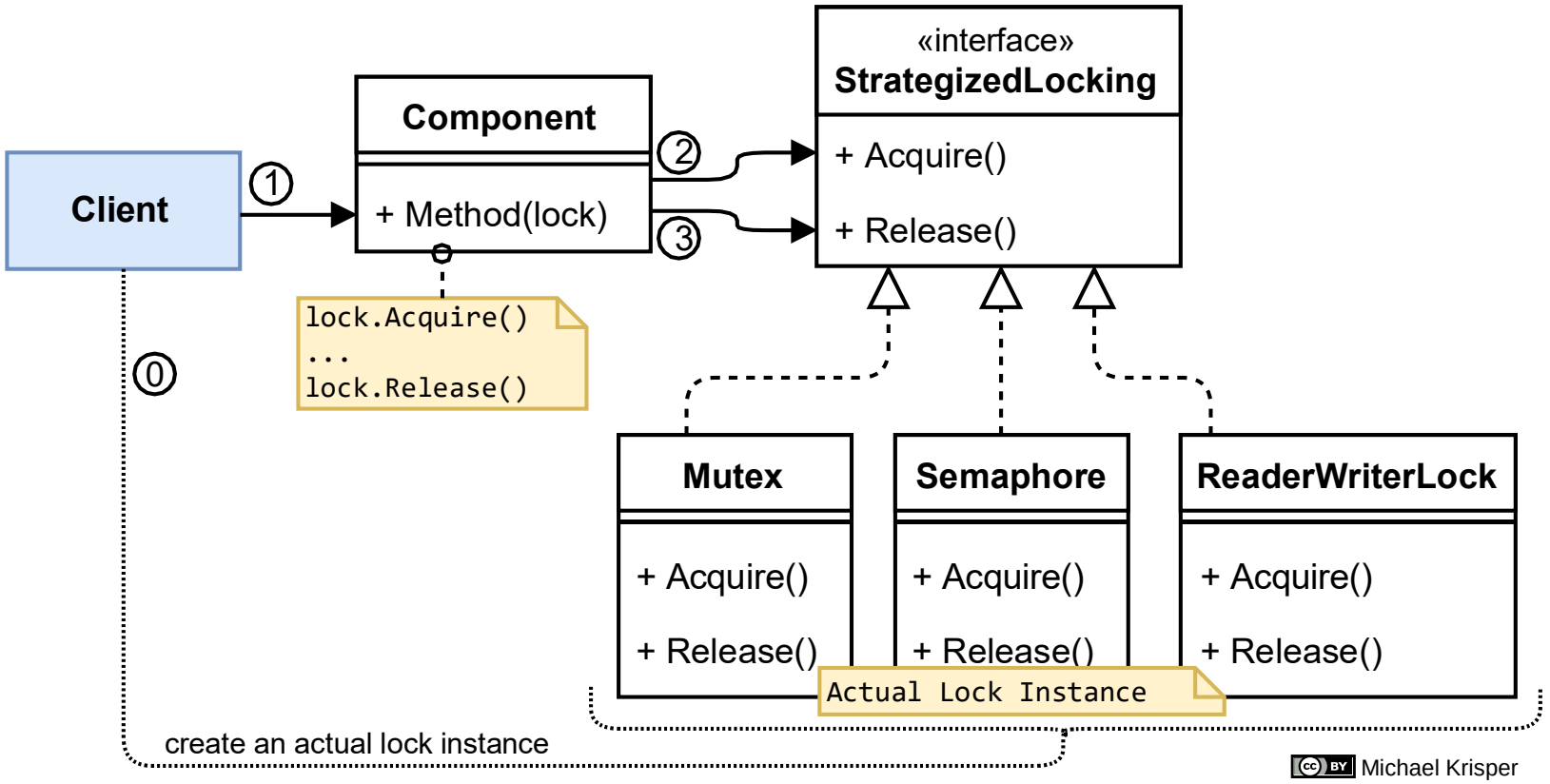
Template-Method vs Strategy vs Command vs Visitor

- What is the intent of each?
- How do they differ?
- In which situations can they be applied?

10 minutes group work (á 2-3 students)!

Locks: Mutex, Semaphore, Read/Write Lock

Ensure mutual exclusive access to some resource.



Locks

Context: Simultaneous access to resources

Problem: How to avoid conflicts and ensure the same view for all accessors?

Forces:

- Parallel access to shared resources (multiple Threads or Processes)
- Locally on one machine
- Read or Write access
- Avoid conflicts (who writes first)
- Enforce consistency (same view for all accessors)

Solution:

- Acquire lock before accessing a resource or wait until lock is available.
- Release the lock after resource is not needed anymore.
- Use a Lock which is synchronized and atomic to the client

Consequences:

\+ Access to resources is mutually exclusive

\+ Logic order is established

~ Which lock is appropriate?

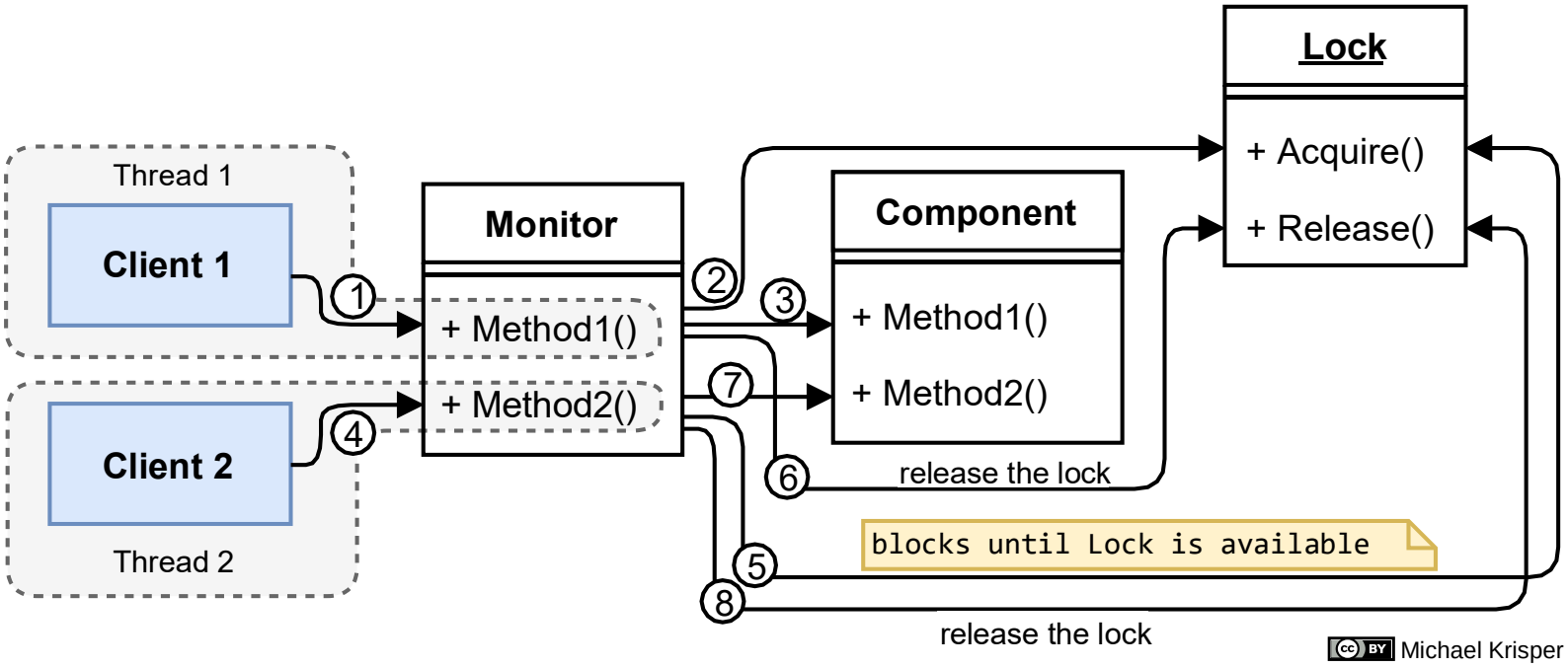
~ Maybe lock is not needed? (Immutable data types? Thread specific storage? Lock-Less Implementations?)

\- Using locks produces overhead & waiting times

\- Race-Conditions & Deadlocks

Monitor

Synchronize method calls to an object



Monitor

Context: Multiple threads accessing an object concurrently

Problem: How to concurrently access an object and call the method without manual synchronization?

Forces:

- Concurrent invocation of methods in an object by multiple threads
- Prevent race conditions: only one method should be active
- Method calls should be synchronized
- Object state should stay stable and resumable

Solution:

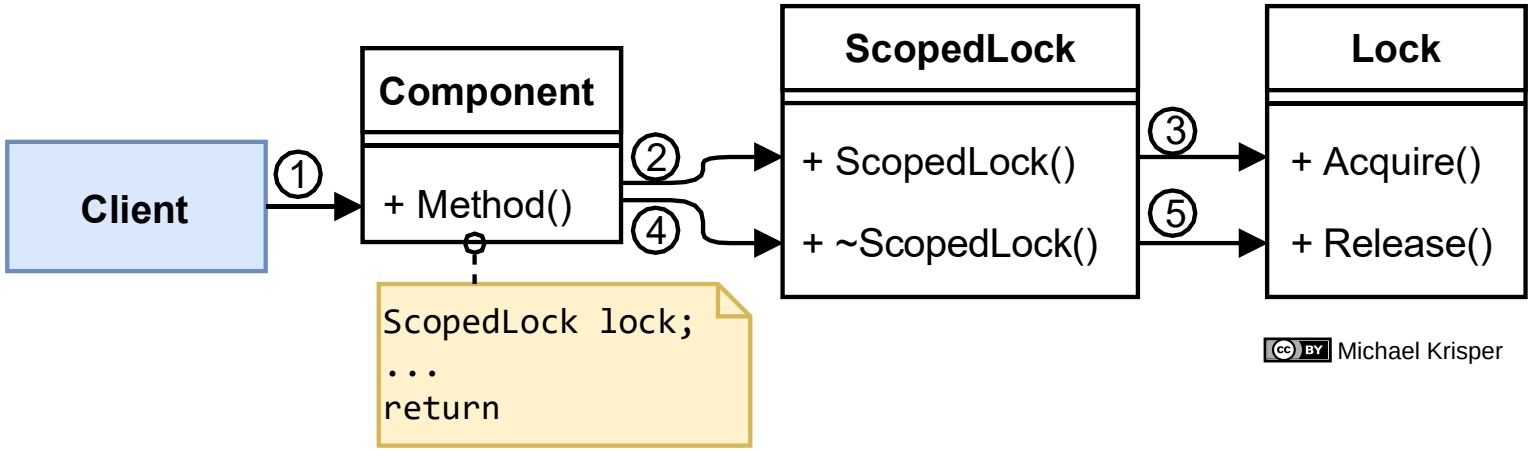
- Use a general lock for one object instance
- Acquire lock before method call, Release after method is finished.

Consequences:

+ Simplification of concurrency control
+ Simplification of scheduling method execution
- Limited Scalability – too coarse lock! (extreme case: GIL in python!)
- Inheritance/Extension is dangerous
- Nested monitors – reaquiring locks?

Scoped Locking

Use scope-rules for acquiring & releasing locks




```
std::mutex _mutex;
int _current = 0;

void increment() {
  std::lock_guard<std::mutex> lock(_mutex);
  ++_current;
}
```

Scoped-Locking

Context: Using locking mechanisms to protect a critical section

Problem: How to avoid forgetting to release the lock?

Forces:

- A critical section of code should be protected for concurrent access with a lock
- The section may have multiple exit points
- Developers tend to forget to release locks on the right places

Solution:

- Implement a class which:
  - Acquires a mutex in constructor
  - Releases the mutex in destructor
- Hide copy constructor and assignment operator
- Use like a normal stack variable and rely on stack-unwinding for destructor calls on leaving a scope

Consequences:

\+ Increased robustness

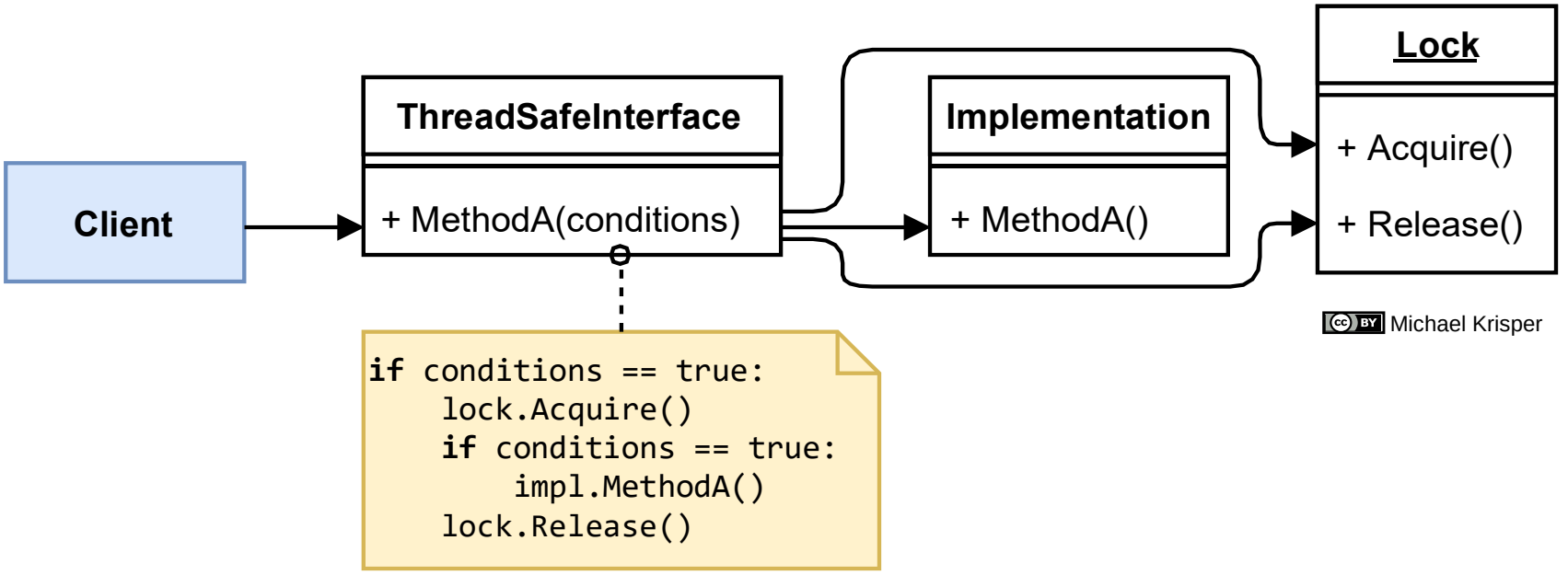
\+ Very simple usage

\- Potential deadlock when used recursively (reacquire lock needed?)

\- Limitations due to language specific semantics (process abort SIGC, longjmp)

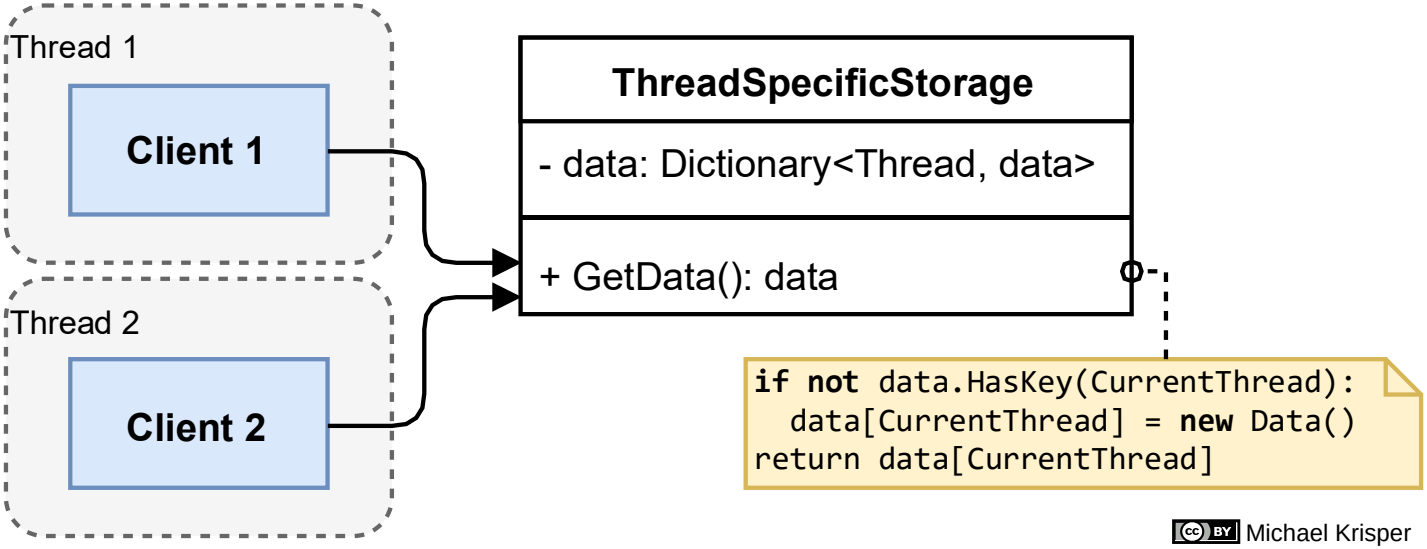
Double Checked Locking

Check twice to ensure conditions



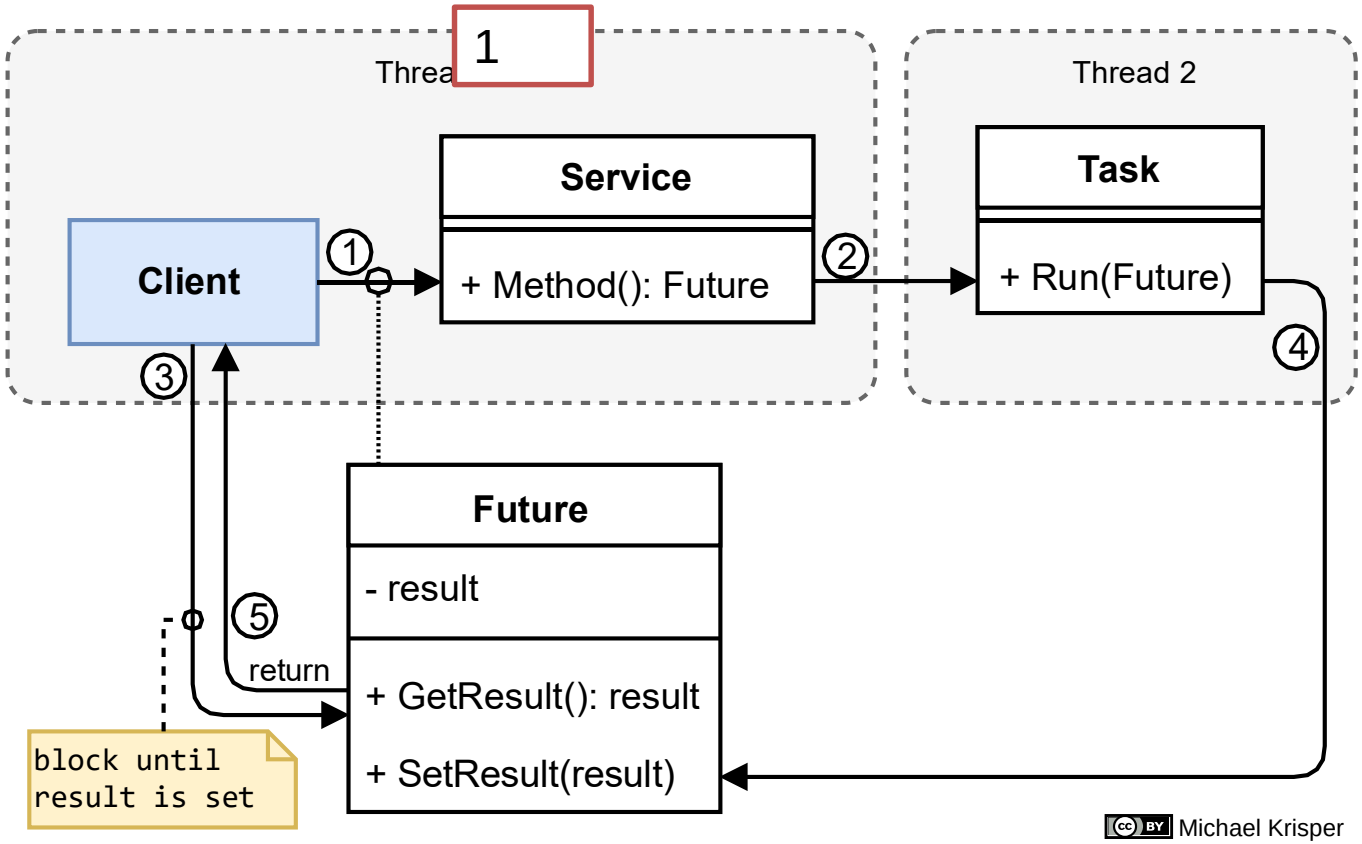
Thread-Specific Storage

Store data corresponding to only one thread



Future

Supply a placeholder for future results



Future

Context: Asynchronous method calls

Problem: How to get the result of an asynchronous method call?

Forces:

- You want to do the call asynchronously.
- You don't know when the call is finished.
- You want to access the result.
- You don't want to busy wait.
- You don't want to expose internal concurrency mechanisms

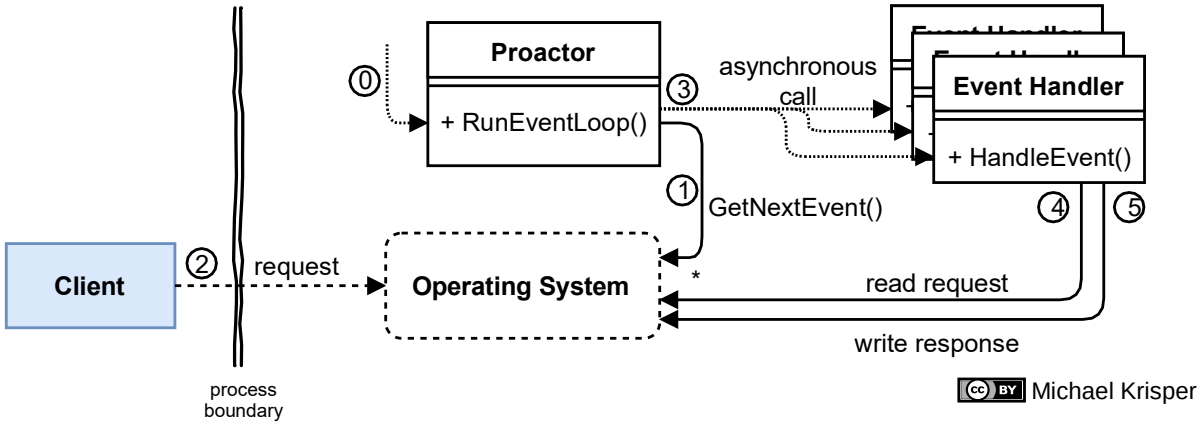
Solution:

- On calling a method immediately return a handle which will contain the result in the future.
- Execute the task asynchronously.
- As soon as the Task is finished, write the result to the future-handle.
- Give the client a possibility to check if the result is available or wait for it.

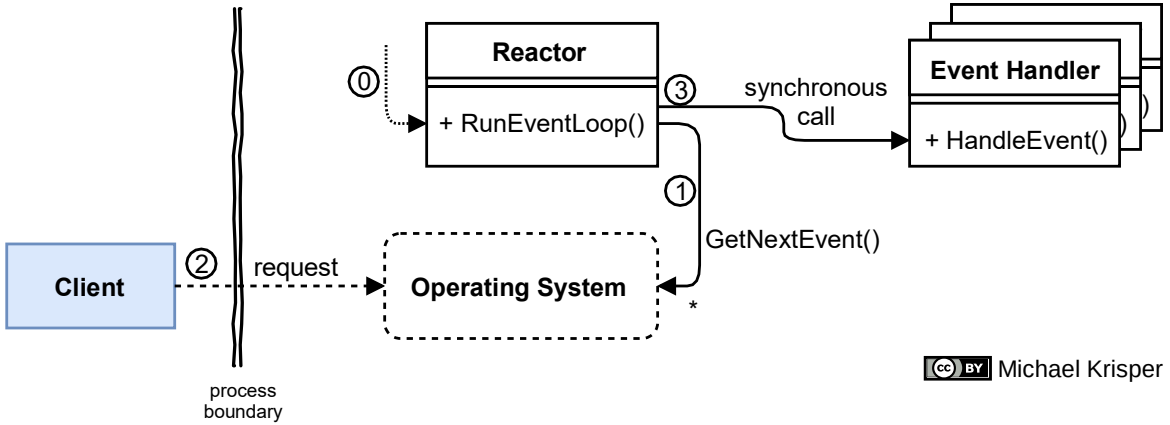
Consequences:

+ User has the possibility to work with "future" results
+ Asynchronous programming gets easier
- No immediate control over the executing thread (no way to cancel, pause)
- Additional memory is needed, to hold results when thread is finished.
- When result is not needed the future-handle is useless.

Proactor (async)

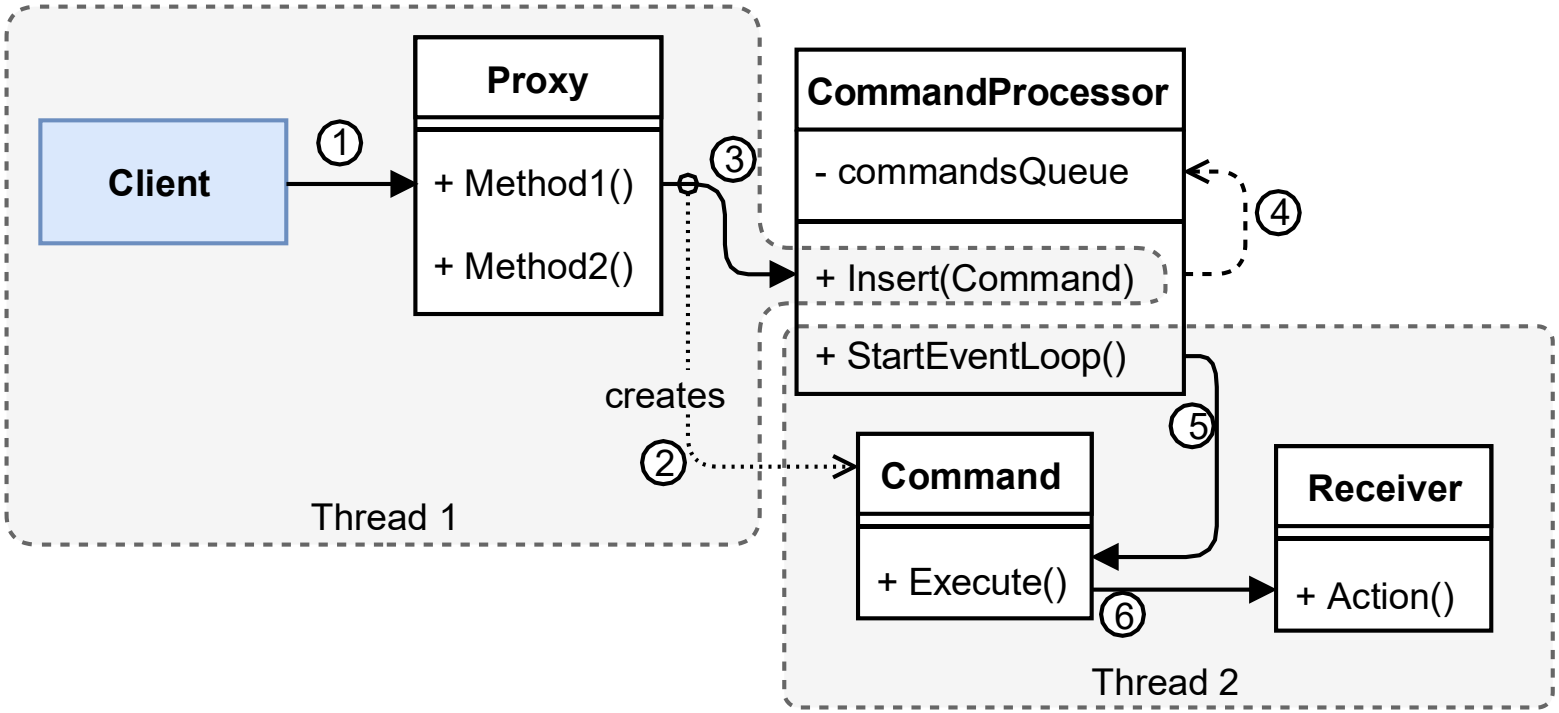


Reactor (sync)



Active-Object

Synchronize method invocation and execute in a different thread



Active-Object

Context: Multiple clients access objects running in different threads or contexts.

Problem: How to execute commands in a different context than the client.

Forces:

- Clients invoke remote operation and retrieve results later (or wait)
- Synchronized access to worker threads
- Make use of parallelism transparently

Solution:

- Implement a proxy with encapsulates all method calls in commands
- Use a Scheduler/CommandProcessor to execute the commands in a separate thread(pool).
- Give the client the possibility to retrieve or wait on the results (async/sync)

Consequences:

\+ Typesafety compared to message passing (usage of classes/objects)

\+ Simplifies sychronization complexity

\+ Client appears to call an ordinary method

\+ Command is executed in a different thread than the client thread

\+ Transparent leveraging of parallelism

~ Order of method execution may differ from invocation

\- Performance overhead

\- Complicated debugging

Async / Await

Execute functions cooperatively in an event loop.

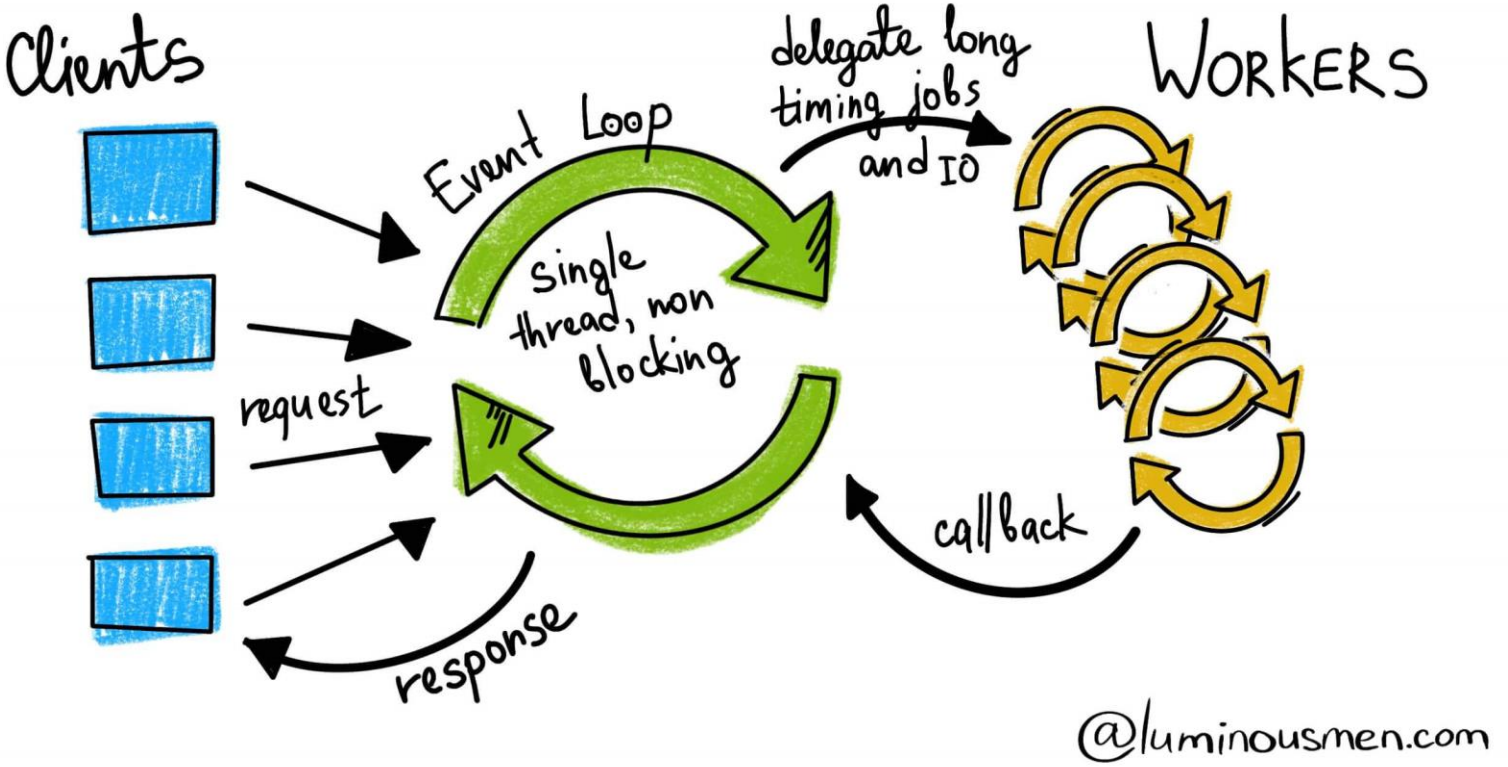


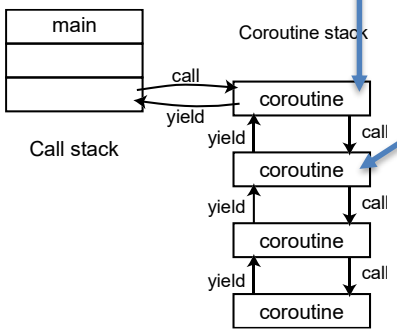
Figure by Luminousmen.com, 17.02.2019, taken from https://luminousmen.com/post/asynchronous-programming-await-the-future

Async / Await Example

```
public async Task<int> SumPageSizesAsync(IList<Uri> uris)
{
    int total = 0;
    foreach (var uri in uris) {
        statusText.Text = string.Format("Found {0} bytes ...", total);
        var data = await new WebClient().DownloadDataTaskAsync(uri);
        total += data.Leng
    }
    statusText.Text = st                                    otal);
    return total;
}
```

Creates a Task A.

Creates a Task B (Closure) which is chained after the Task A and returned to caller. When the Task B finishes, execution continues with Task A.



Async / Await

Context: Executing multiple functions waiting for I/O resources.

Problem: How to execute I/O-bound functions in parallel without having to use multithreading and synchronisation.

Forces:

- Executing the blocking functions sequentially is slow.
- Executing the functions in own threads may cause synchronisation problems or wasting resources due to context switching.
- Multithreading programming is error-prone.
- Some environments don't have true multithreading (python, javascript)

Solution:

- Compile the functions as state machines, with transitions at the "await" statement
- Execute the state machines in an event loop, advancing them based on a "ready"-condition (or signal).

Consequences:

+ No need to use multiple threads.
+ No need to synchronise.
+ No unnecessary waiting times due to blocking functions.
+ Simple usage (nearly like single-threaded programming, except for the "await" keyword).
- Syntax and Compiler support needed.
- Must be supported throughout the whole application (async/await and non-blocking functions virtually everywhere)
- Relies on cooperativeness!
- CPU-bound functions may still block everything.