Design Patterns 448.058 (VO)

Michael Krisper
Georg Macher

27.11.2019

europlop

1.-5. July 2020

25th European Conference on Pattern Languages of Programs. EuroPLoP is held at Kloster Irsee in Bavaria, Germany.

Conference

Patterns

Hillside Europe

https://europlop.net/

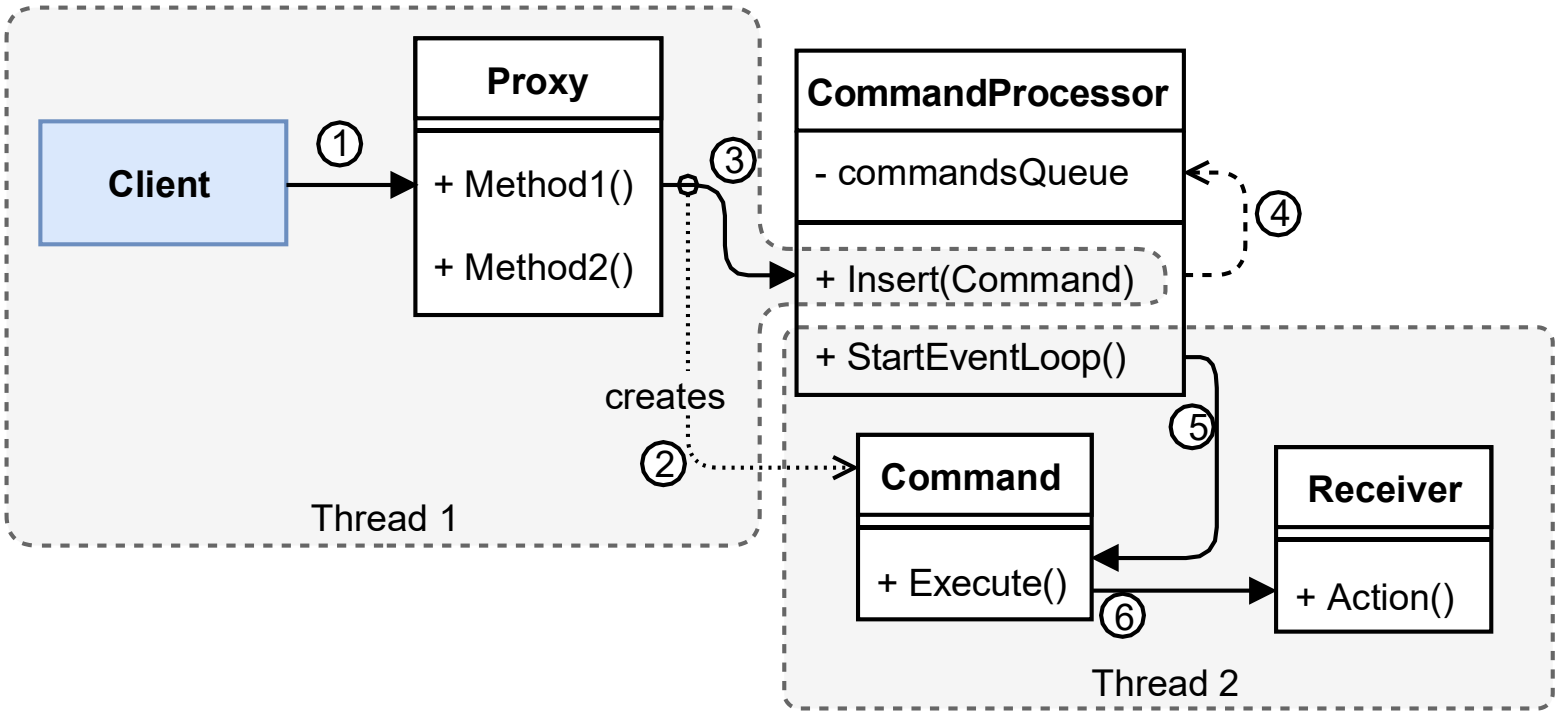
The European Conference on Pattern Languages of Programs

Revision from last time…

- Visitor
- Composite
- State
- Template-Method
- Locks
- Scoped Locking
- Monitor
- Future
- Active-Object
- Async/Await

Active-Object

Encapsulate method invocation and execute asynchronously



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:

\+ Simplifies sychronization complexity

\+ Client calls an ordinary method

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

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

\+ 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.

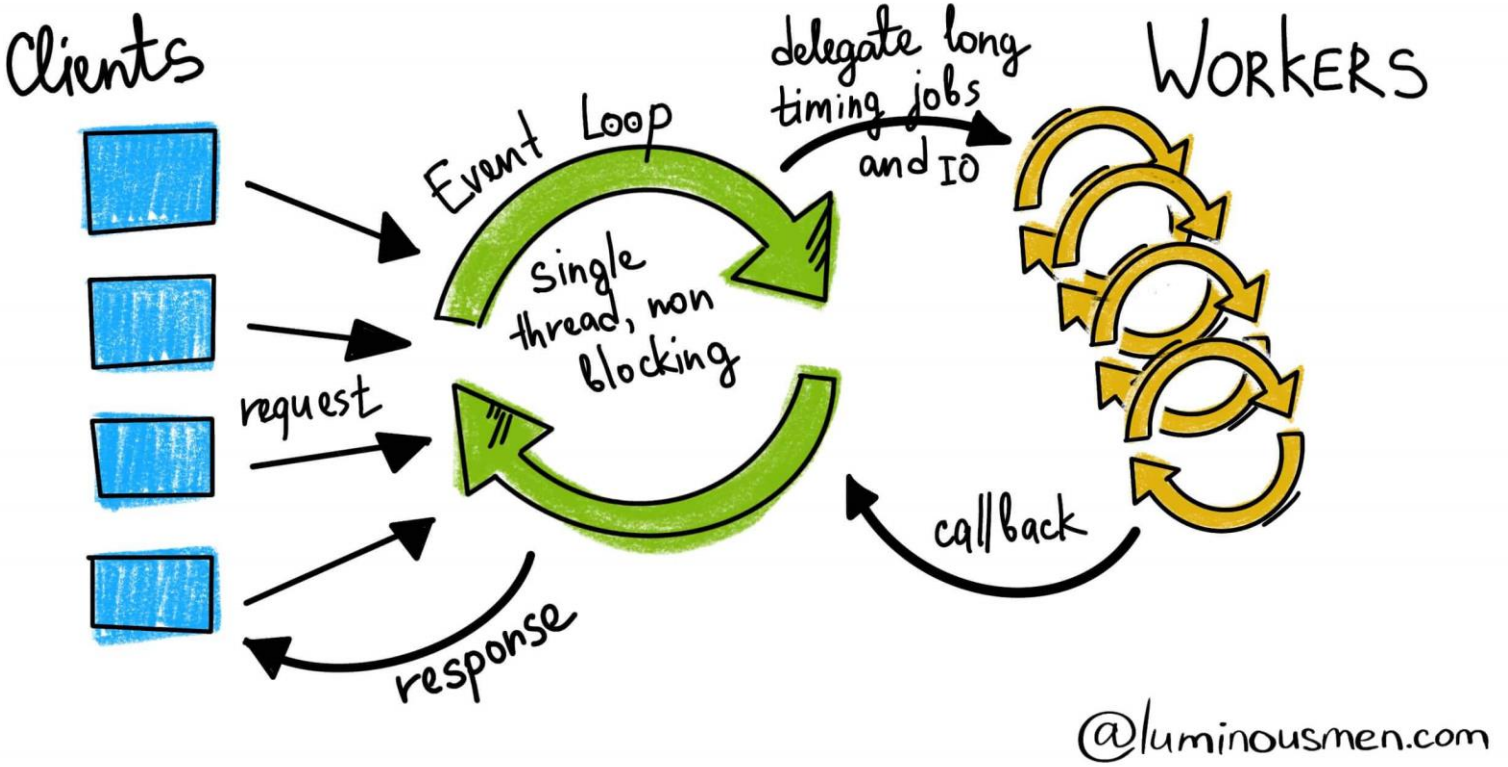


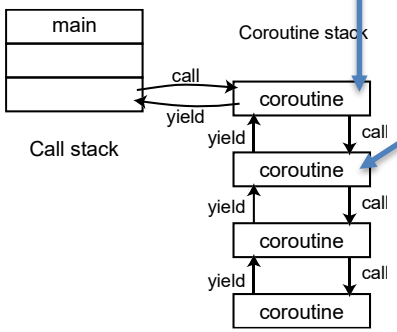
Figure by Luminousmen.com, 17.02.2019, taken from https://luminousmen.com/post/asynchronous-programming-await-the-future

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 A finishes, execution continues with Task B.



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 still block everything.

Learning Goals for Today

Idioms:
- Counted Pointer

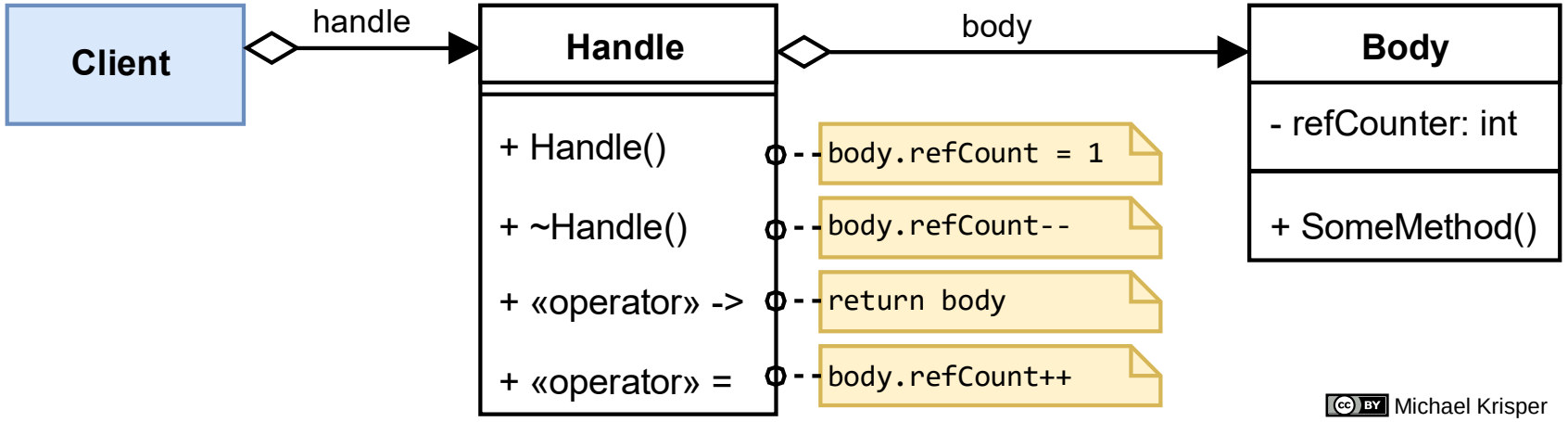
Resource Patterns:
- Lazy Acquisition
- Eager Acquisition
- Partial Acquisition
- Caching & Pooling
- Leasing
- Garbage Collector

Others:
- Chain of Responsibility
- Interpreter

Summary and Wrap-up

Counted Pointer / Smart Pointer / Shared Pointer / Auto Pointer

Count references and call destructor when no one is using the object anymore.



Counted Pointer

Context: Manual dynamic memory management with pointers

Problem: How to know when a object can be safely destroyed?

Forces:

- We don't know exactly who still has a reference to our objects.
- Several clients may share the same objects
- We want avoid "dangling" references
- If a object is not referenced anymore it should be destroyed and its memory and resources released
- The client should not need too much additional effort

Solution:

- Implement body which has a counter for the number of references.
- Implement a proxy which does the following for every instance of a body:
  - Overload operator ->, =, copy
  - Constructor increases counter
  - Destructor decreases counter, on 0 it deletes object.
- Only allow access to body via the proxy object.

Consequences:

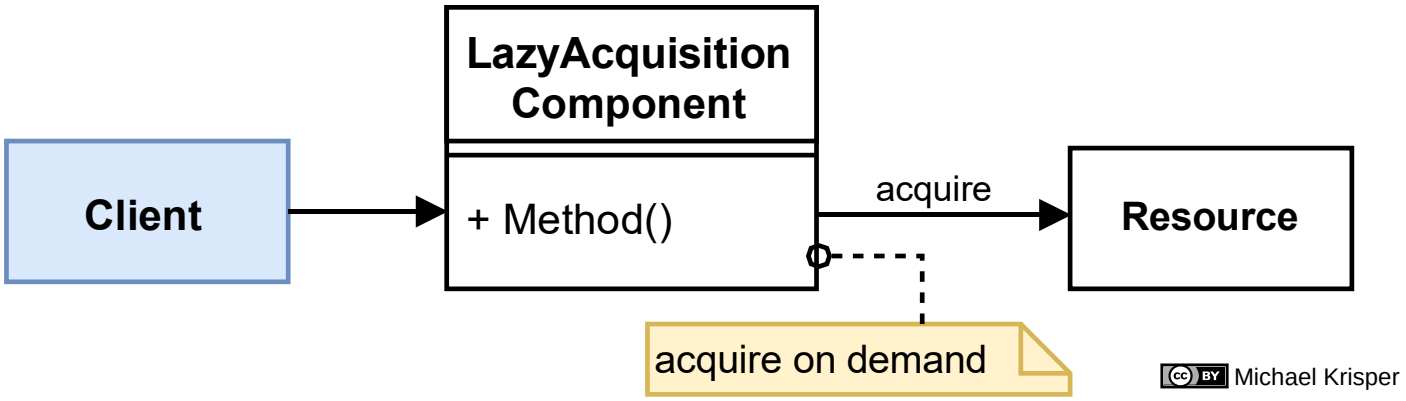
\+ Automatic destruction if object is not referenced anymore

~ Shared vs. Unique Pointers?

\- Circle references!

Lazy Acquisition

Defer acquisition of resources to time of actual usage



Lazy Acquisition

Context: Using resources in an application

Problem: How to save resources and load an application faster?

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- They take time to load.
- They may be scarce.
- They are not needed from the beginning, but later on.

Solution:

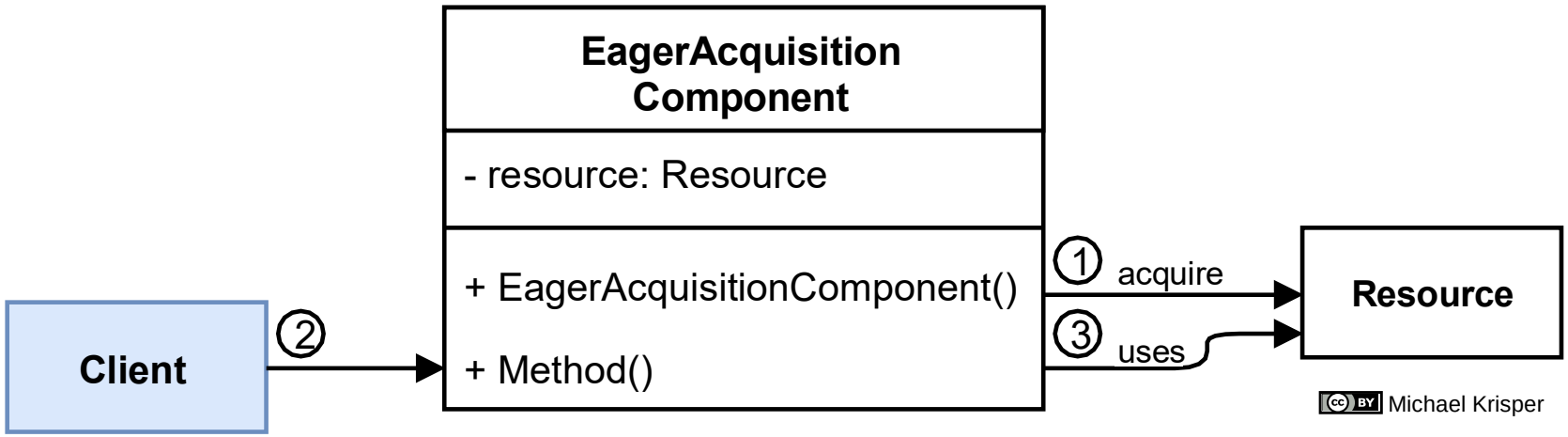
- Implement a proxy which can be used by the client
- The proxy should defer acquisition of the resource until the last possible moment.

Consequences:

+ Resources are only acquired when really needed.
+ Client doesn't have to care about using to much resources early on.
+ Application starts faster.
- Waiting times during acquiring the resources (do it async!)
- Additional layer of abstraction
- Avoid acquiring resources to often (caching & pooling!)

Eager Acquisition

Acquire resources in advance.



Eager Acquisition

Context: Using resources in an application

Problem: How to avoid having to wait for resources during runtime.

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- Exclusive access is no problem.
- They are always needed in the application.

Solution:

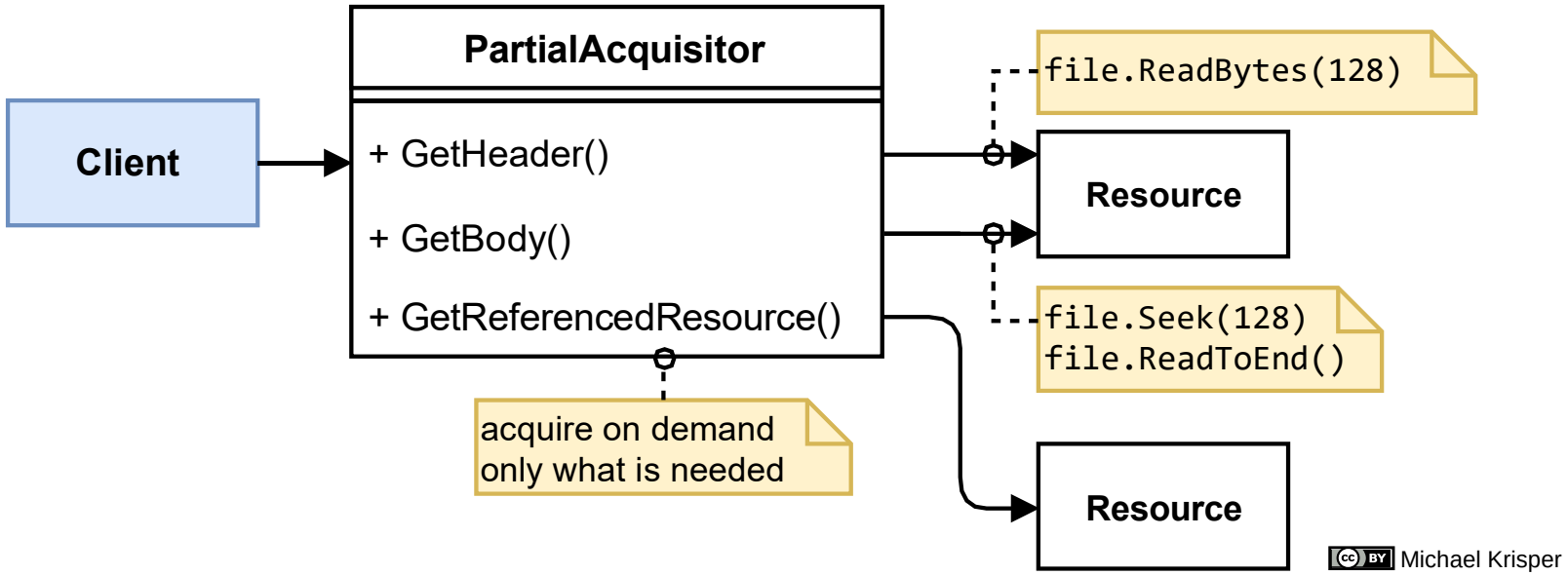
- Acquire the resources on startup and store them in some cache or vault (singleton).
- Give access to the already loaded resources

Consequences:

+ Resources must not be loaded later on.
+ No delay on using the resources.
- Resources take up memory space.
- Startup may be slowed down due to loading the resources (do it async!)

Partial Acquisition

Acquire resource in parts. Only use the part which is currently needed.

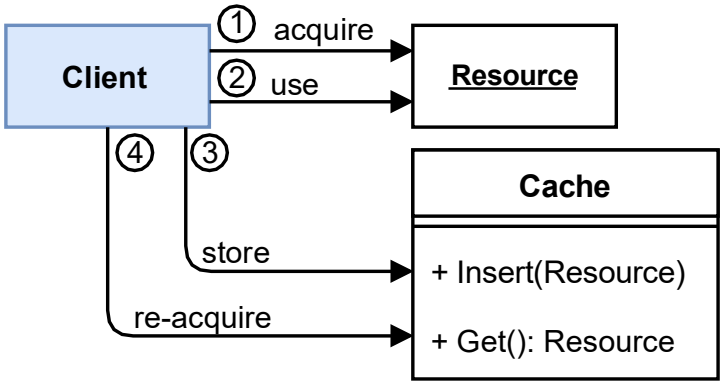


Caching & Pooling

Save resources for later reuse.

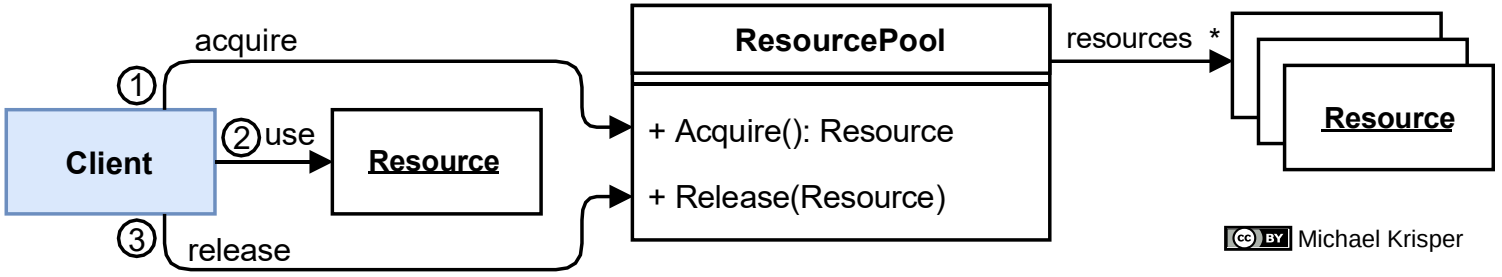
Caching

Let the client decide what to cache



Pooling

Wrap access to the resource in a manager.



Caching & Pooling

Context: Using resources in an application

Problem: How to avoid loading or creating resources over and over?

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- They may take time to load.
- They are needed more than once and in different places.

Solution:

- Provide a cache to store already loaded resources there (singleton).
- Supply means to access the cache to the client (factory).
- Restrict access if needed.

Consequences:

\+ Resources are loaded only once and reused afterwards

\+ Subsequent usages are much faster

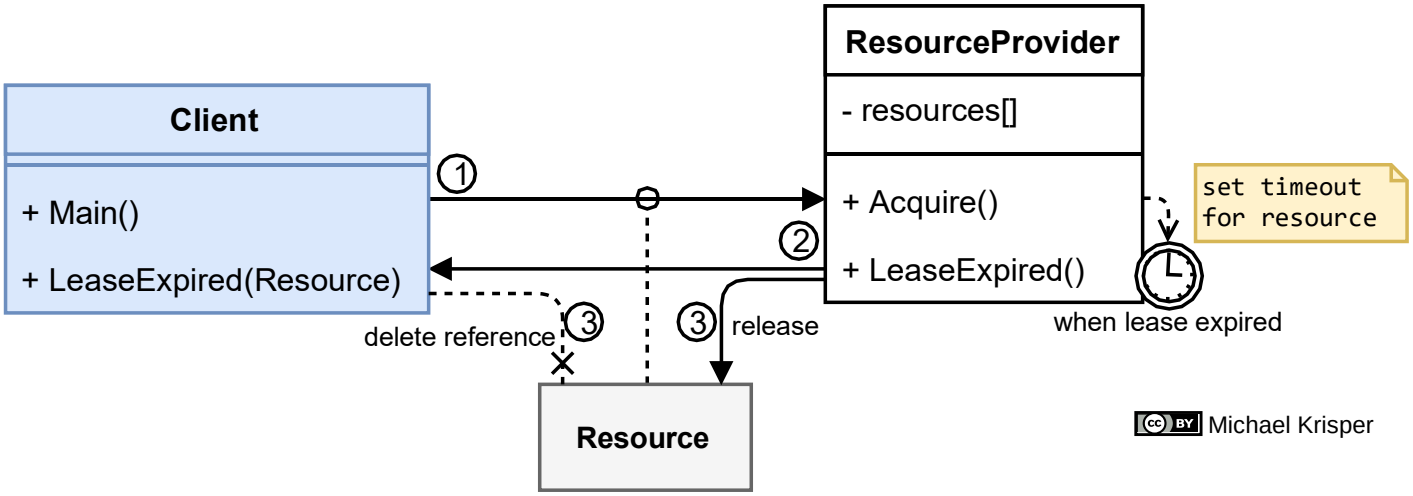
~ Mutual exclusive access for other applications?

\- Uses much memory space

\- Resources may be outdated

Leasing

Set expire-timeout for resources.



Leasing

Context: Using resources in an application

Problem: How to avoid that resources can be exclusively be used by only one client.

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- The resource may be used by multiple clients.
- It should be avoided that one client can exclusively use a resource forever.

Solution:

- Supply access to the resource via a LeasingProxy which invalidates the resource some time after acquisition.
- Inform the client that the usage time is over.
- Restrict direct access to the resource.

Consequences:

\+ Resources cannot be used exclusively anymore

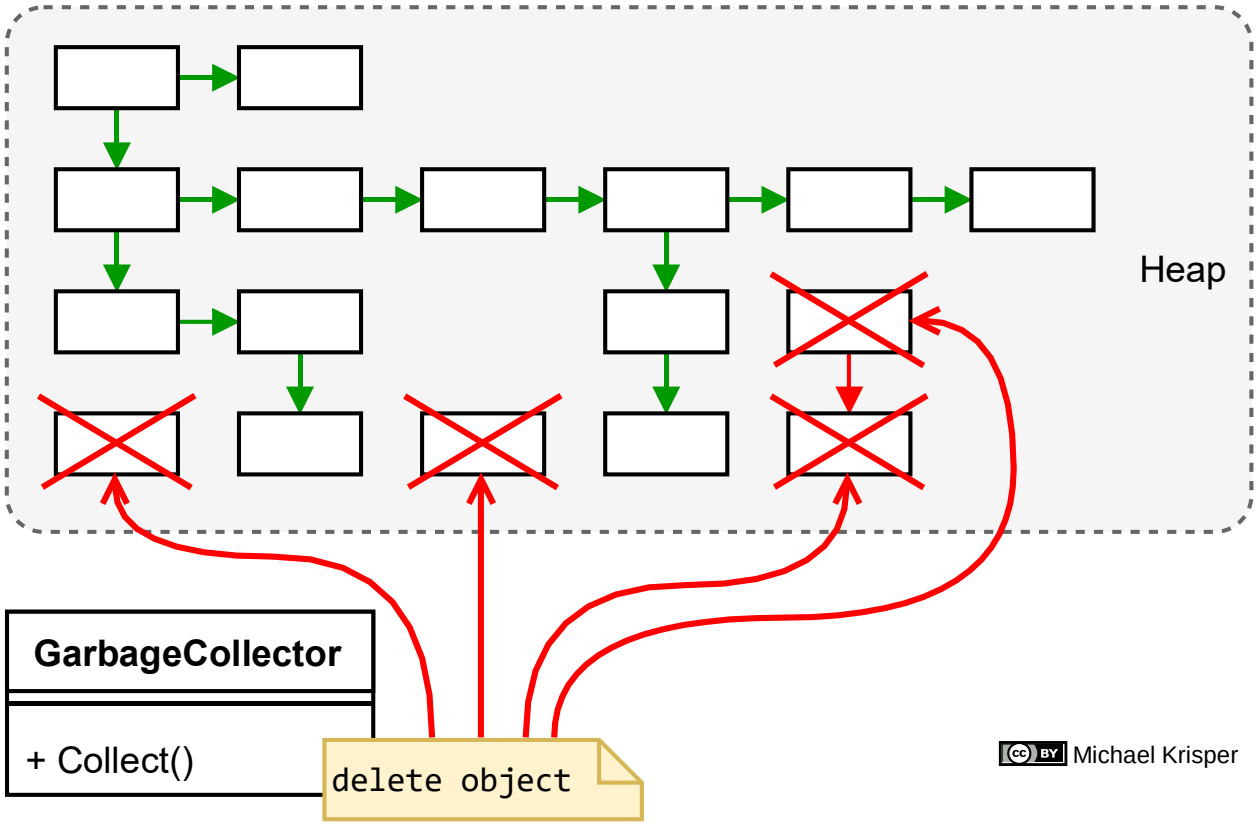
\+ If client forgets to release the resource it gets released automatically after some time.

~ What is the right duration?

\- To early release could lead to errors.

Garbage Collector

Maintain reference-graph of objects and delete unreachable branches.



Garbage Collector

Context: Application which acquires dynamic memory.

Problem: How to avoid dangling references in an application to avoid memory leaks?

Forces:

- Memory can be dynamically acquired to store objects
- Pointers/References can be freely passed and copied
- Client doesn't want to care about memory allocation.

Solution:

- Maintain reference graph for each and every dynamically created object.
- Periodically search over graph for unreachable branches/subgraphs
- Delete unreachable subgraphs.

Consequences:

\+ Client doesn't has to care for manual memory management.

\+ No memory leaks

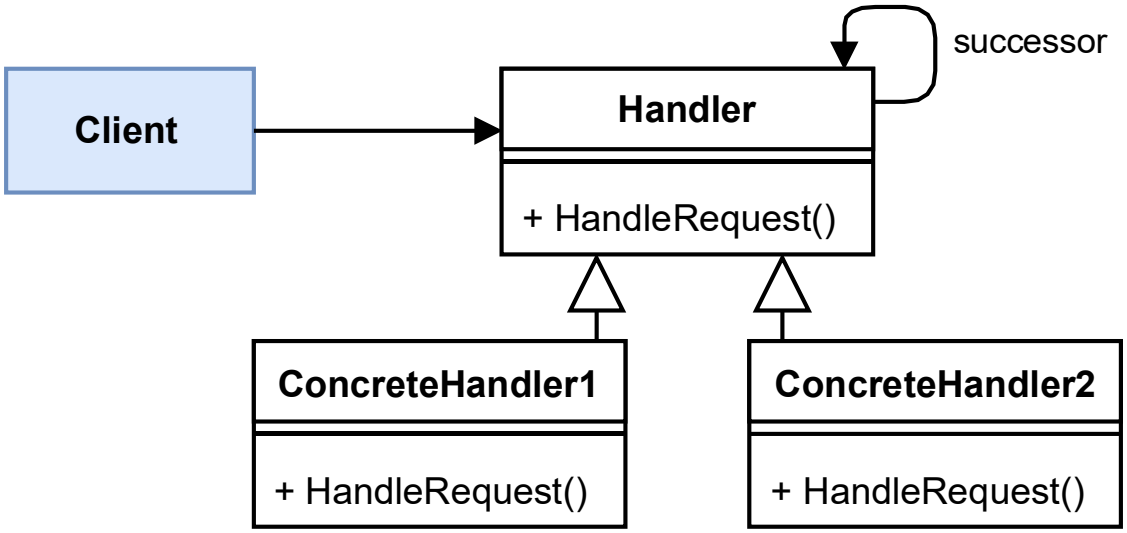
~ How often should collection be done? Performance Optimizations (Generation Concept)?

\- Performance overhead during creation and garbage collection (traversal)

\- Memory overhead by storing all reference counts

Chain of Responsibility

Forward a call until an object can handle it.



Chain of Responsibility

Context: Having a task or problem which can be handled by several objects.

Problem: How to dynamically resolve which object is responsible for a specific problem/task?

Forces:

- Having different types of tasks which have to be handled.
- Having several objects which can handle different tasks.
- Tasks and the actual Handlers are not known at compile-time.
- There should be multiple escalation levels.

Solution:

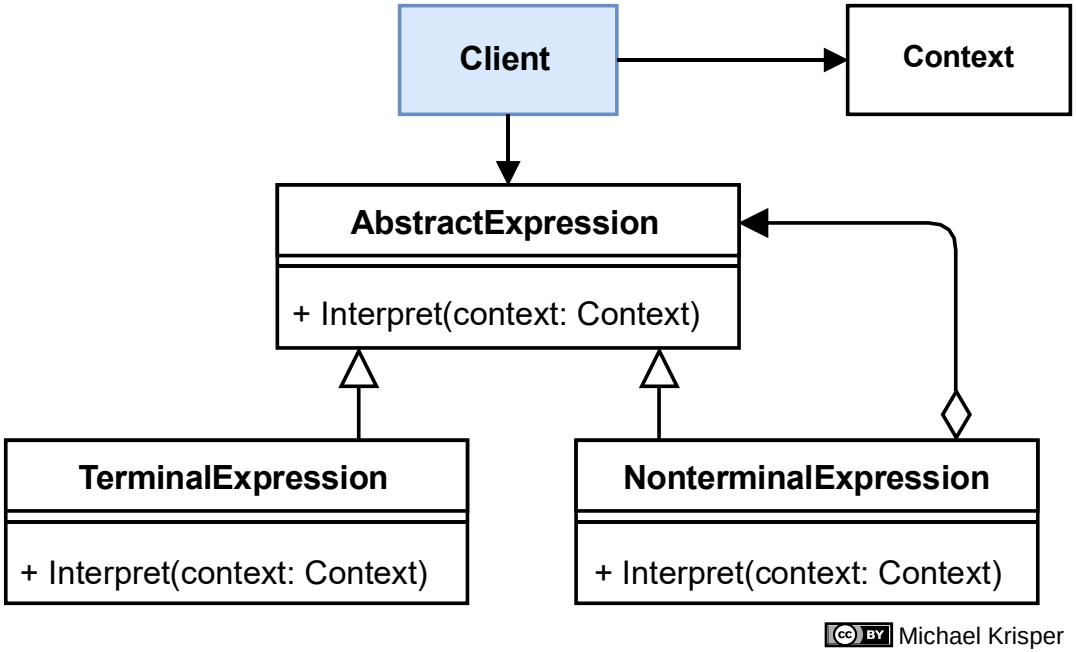
- Implement a chain of handlers.
- Forward the task to the first object which can handle it.
- Add more general handlers in the end of the chain.

Consequences:

\+ Dynamic handling of events
\+ Loosely coupled responsibility
\+ Can be changed at runtime
~ Who builds the chain?
~ Common standards/conventions?
~ Only one handler or multiple? (decorator-style)
~ Fallbacks?
\- Possible huge call stack
\- Critical path is single point of failure

Interpreter / Abstract Syntax Tree (AST)

Read expressions one after another and build a tree of expressions.



Summary and Wrap-Up

Learning Goals for Course

Design Patterns Theory

- What is a design pattern? Why do we need them?
- What are principles behind design patterns?
- How to describe design patterns?
- What is a pattern language?

Application of Design Patterns

- When to use what?

Design Patterns in Detail

- Know core ideas and application of important design patterns! (~50)

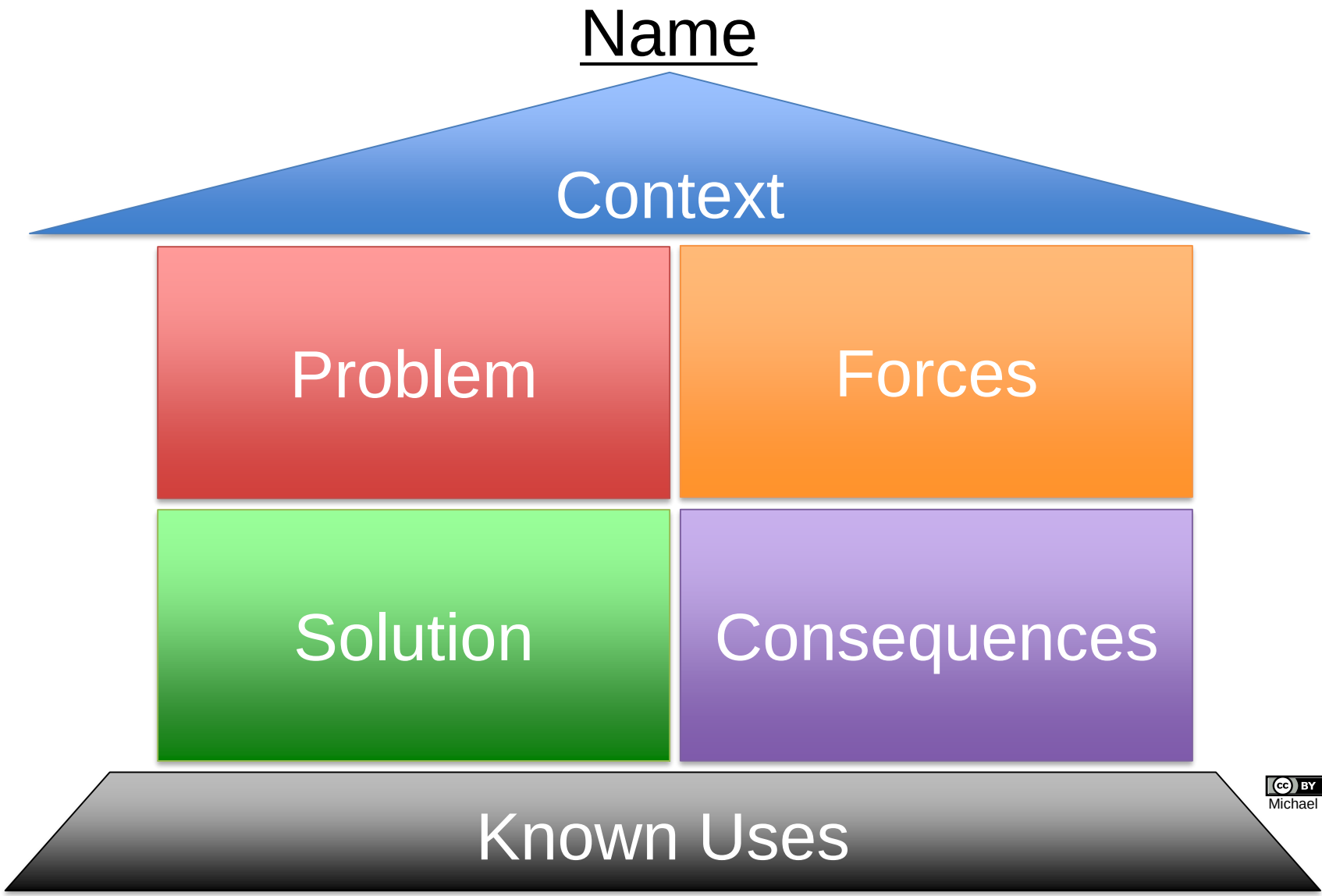
Design Patterns

What is a pattern?

A proven solution for a (recurring) problem.

- **Name**: A catchy name for the pattern
- **Context**: The situation where the problem occurs
- **Problem**: General Problem Description
- **Forces**: Requirements and Constraints - Why does the problem hurt in this context?
- **Solution**: Generic Description of a proven solution.
  Static Structures, Dynamic Behaviour
- **Consequences** (Rationale):
  What are the benefits and liabilities?
  What are the limitations and tradeoffs?
  How are the forces resolved?
- **Known-Uses**: Real Life Examples

The Design Pattern House



SOLID Principles (in OOP)

- ***S*ingle Responsibility**: A class should have one, and **only one**, **reason to change**.
- ***O*pen Closed**: You should be able to **extend** a class's behavior, **without modifying** it.
- ***L*iskov Substitution**: Derived classes must be **substitutable** for their base **classes**.
- ***I*nterface Segregation**: Make **fine grained interfaces** that are client specific.
- ***D*ependency Inversion**: **Depend on abstractions**, not on concrete implementations.

Principles of Good Programming

- **Decomposition**
  make a problem manageable
  decompose it into sub-problems
- **Abstraction**
  wrap around a problem
  abstract away the details
- **Decoupling**
  reduce dependencies, late binding
  shift binding time to “later”
- **Usability & Simplicity**
  make things easy to use right, hard to use wrong
  adhere to expectations, make usage intuitive

Types of Design Patterns

Architectural Patterns

- Fundamental structural patterns
- Stencils for whole architectures
- Examples: Layers, Pipes & Filters, Broker, Model-View-Controller, Microkernel

Design Patterns

- Solution templates for more isolated problems
- Examples: Composite, Adapter, Proxy, Factory

Idioms

- Fine-Grained Patterns for problems in specific programming languages or environments
- Examples: Counted Pointer, Scoped Locking

GoF Patterns

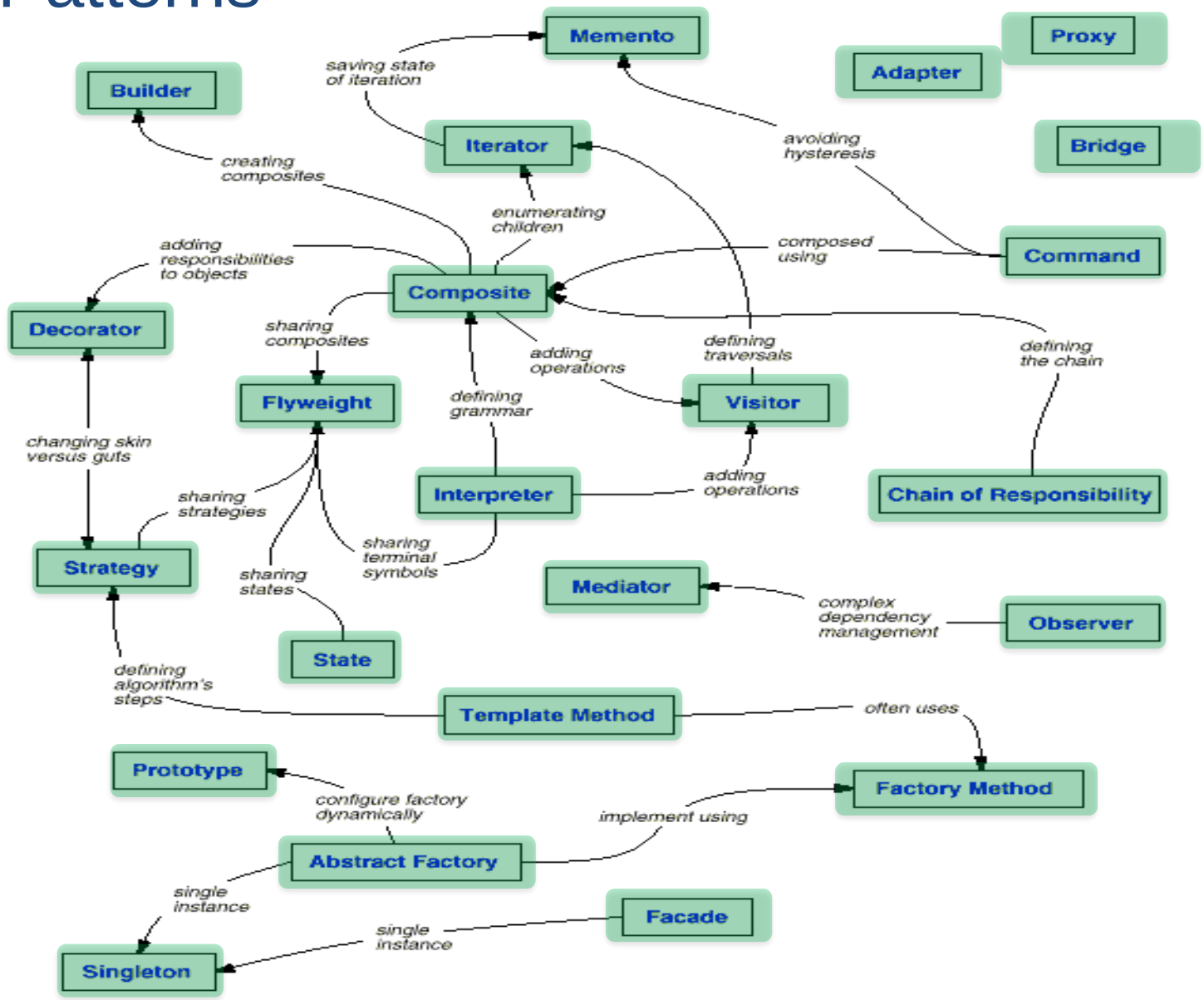


Figure from [Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995, ISBN 0-201-63361-2]

POSA 1 Patterns

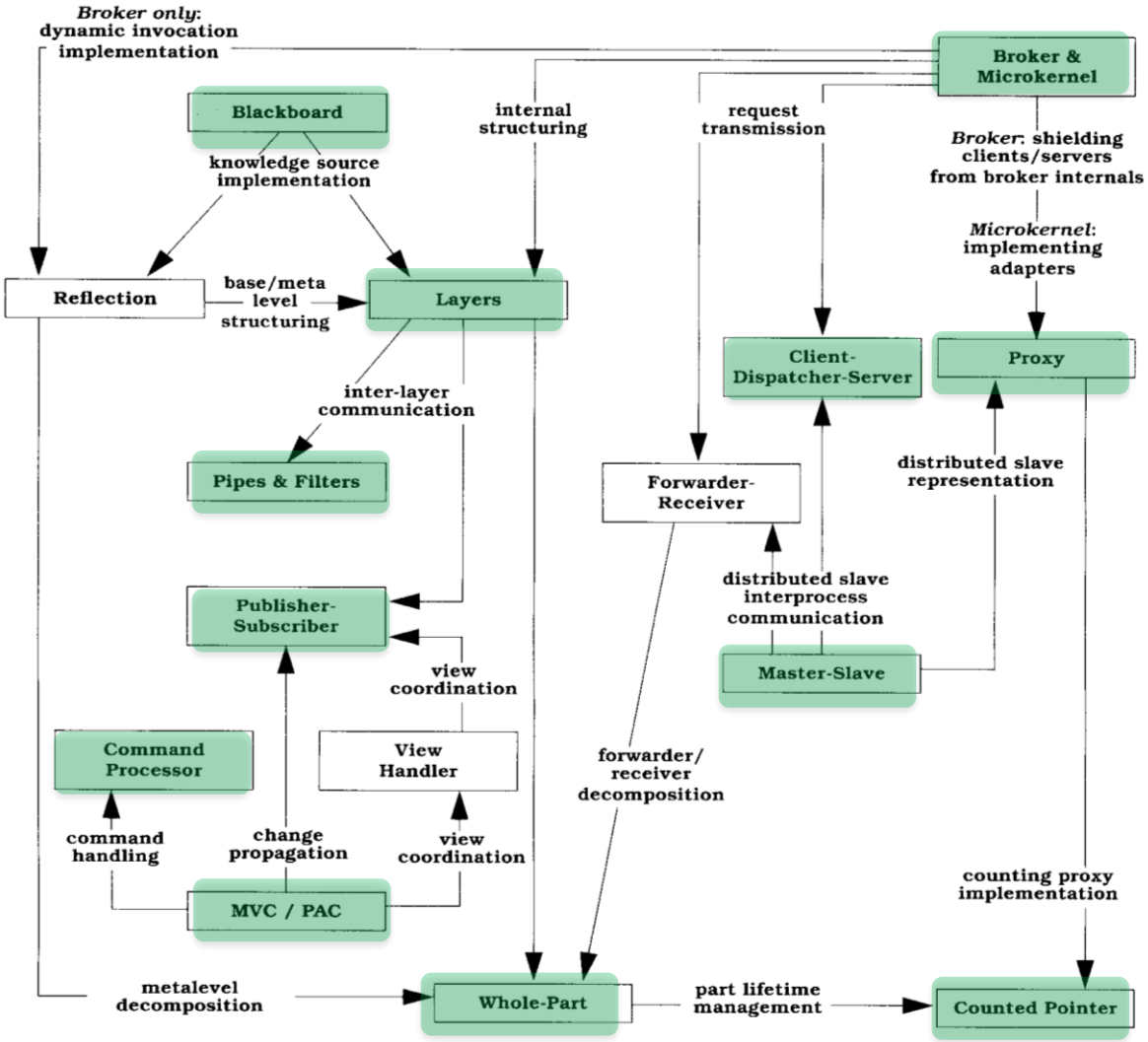


Figure from [Pattern-Oriented Software Architecture Volume 1: A system of patterns (Buschmann, Meunier, et al., 1996)]

POSA 2

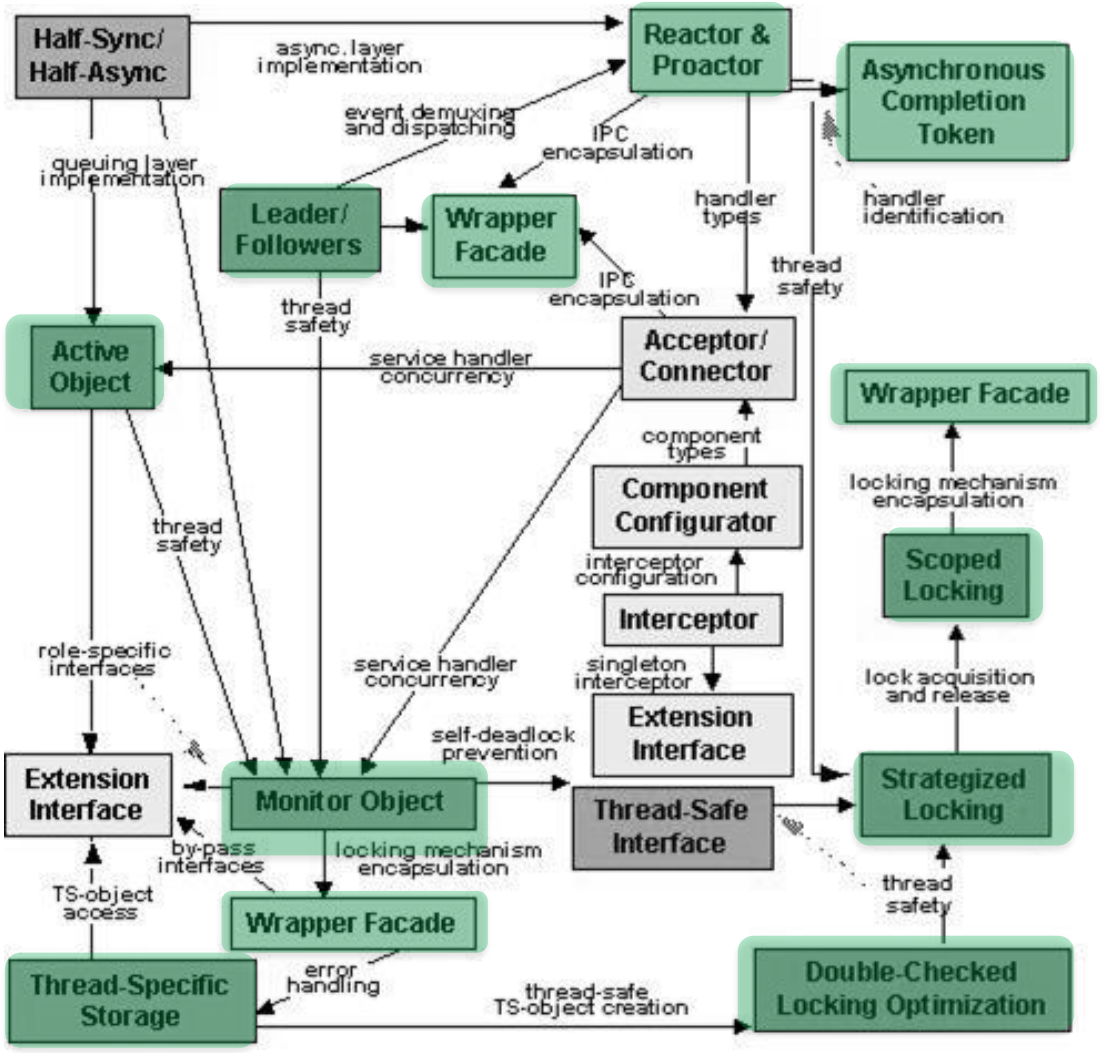


Figure from [Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects (Schmidt et al., 2000)]

53 Patterns…

- **Wrapping**: Adapter, Façade, Decorator, Proxy
- **Creation**: Factory Method, Abstract Factory, Builder, Prototype, Singleton, Flyweight
- **Behaviour**: Strategy, Command, State
- **Architecture**: Layers, Pipes & Filters, Broker, Master-Slave, Client-Server
- **Collections**: Iterator, Visitor, Composite
- **Communication**: Observer, Bridge, Broker, Mediator, Blackboard, Microkernel, Client-Dispatcher-Server/Lookup, Messages, Endpoint, Translator, Router, Handler, MVC
- **Concurrency**: Locks, Monitor, Active Object, Future, Scoped Locking, Thread-Specific Storage, Double-Checked-Locking, Async/Await
- **Resources**: Lazy Acquisition, Eager Acquisition, Partial Acquisition, Caching & Pooling, Leasing, Garbage Collector
- **Others**: Memento, Counted Pointer, Chain of Responsibility, Interpreter/Abstract Syntax Tree

A few philosophical thoughts…

“Patterns are a universal principle”

- **How to transfer knowledge?**

- How to make knowledge explicit?
- How to make knowledge findable?
- How to make knowledge understandable?
- How to make knowledge applicable?

“Study hard what interests you the most in the most undisciplined, irreverent and original manner possible.”

— Richard Feynmann

Thank you & Good Luck!

Remember us for Projects/Seminars/Bachelor/Master @ ITI

Michael Krisper
michael.krisper@tugraz.at

Georg Macher
georg.macher@tugraz.at

Project:
TEACHING