# Konfigurierbare Systemsoftware (KSS)

## VL 4 – Aspect-Aware Development: The CiAO Approach

**Daniel Lohmann**

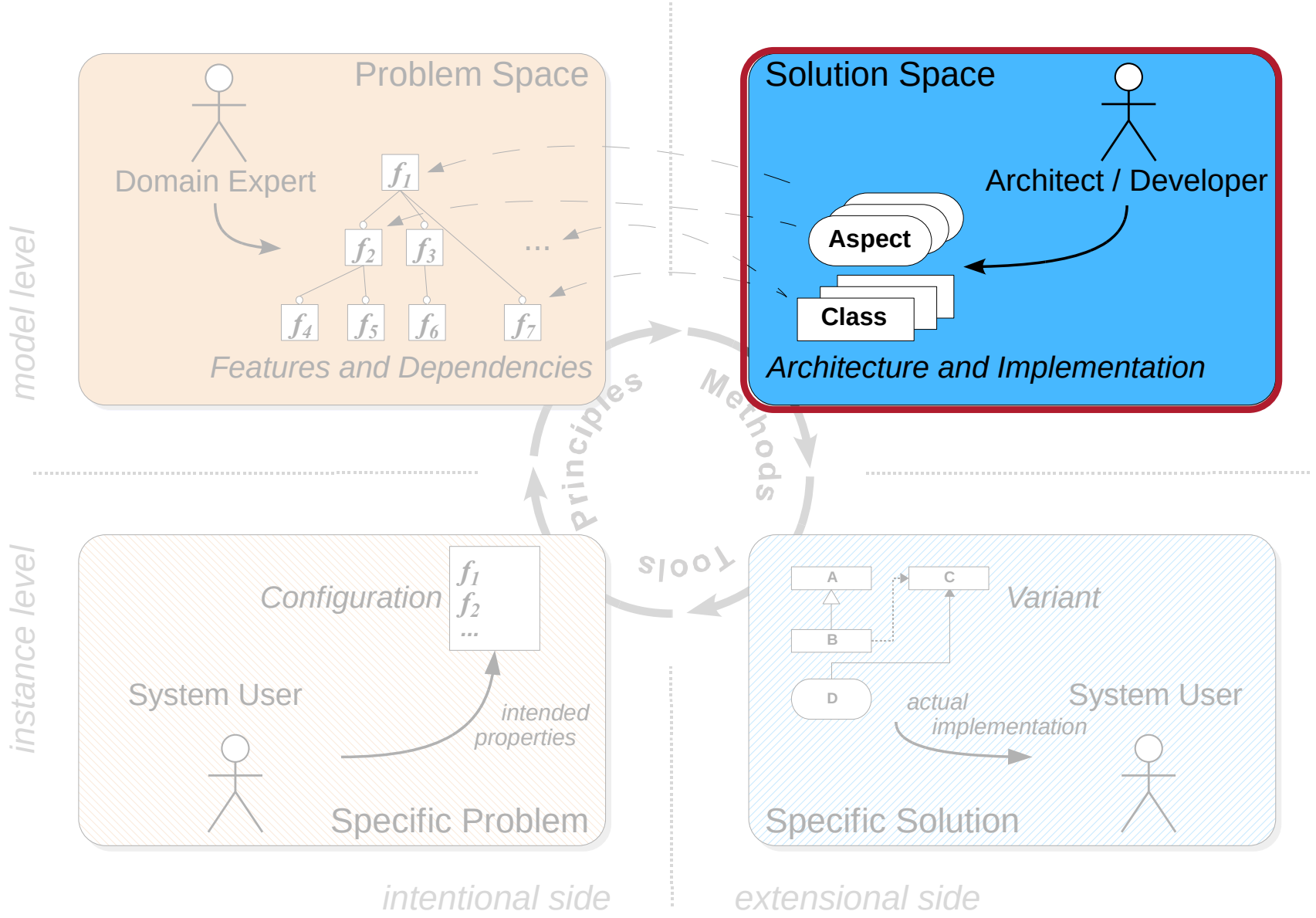
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 14 – 2014-05-08

http://www4.informatik.uni-erlangen.de/Lehre/SS14/V_KSS

# About this Lecture



# Implementation Techniques: Classification

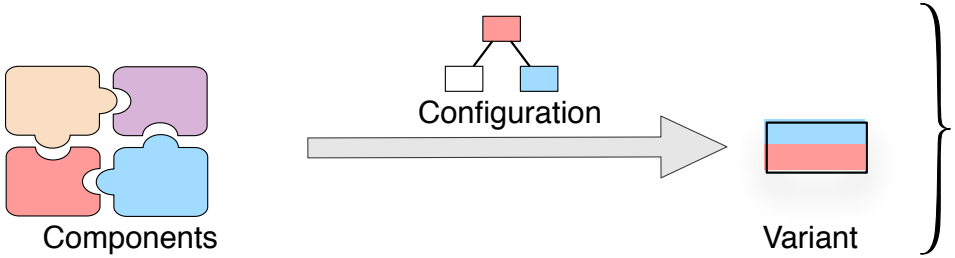


- Decompositional Approaches




  - Text-based filtering (untyped)
  - Preprocessors

- **Compositional Approaches**




  - Language-based composition mechanisms (typed)
  - OOP, **AOP**, Templates

- Generative Approaches




  - Metamodel-based generation of components (typed)
  - MDD, C++ TMP, generators

# Implementation Techniques: Goals



## General

1. Separation of concerns (SoC)
2. Resource thriftiness

How to achieve these with AOP?

## Operational

3. **Granularity** Components should be fine-grained. Each artifact should either be mandatory or dedicated to a single feature only.
4. **Economy** The use of memory/run-time expensive language features should be avoided as far as possible. Decide and bind as much as possible at generation time.
5. **Pluggability** Changing the set of optional features should not require modifications in any other part of the implementation. Feature implements should be able to "integrate themselves".
6. **Extensibility** The same should hold for new optional features, which may be available in a future version of the product line.

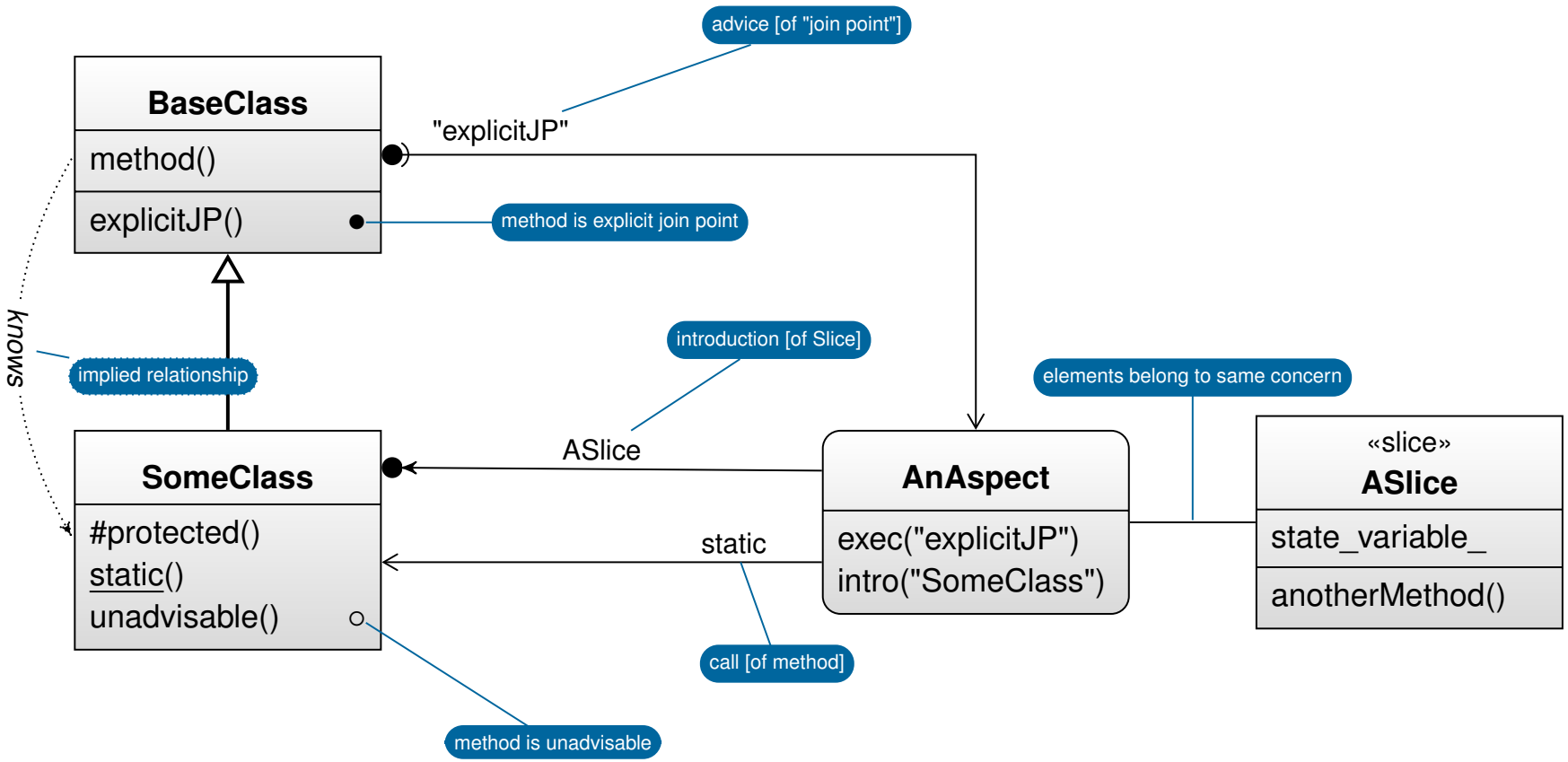
# Agenda



# Agenda



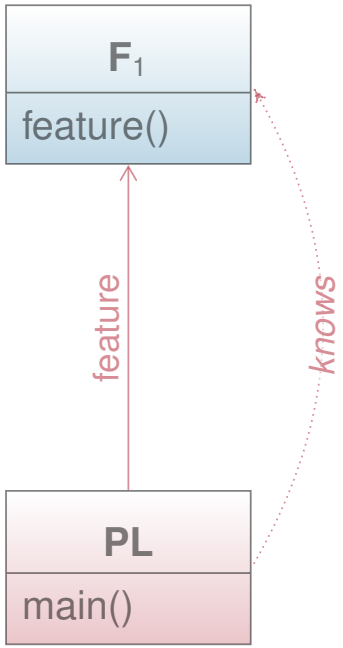
# Notation

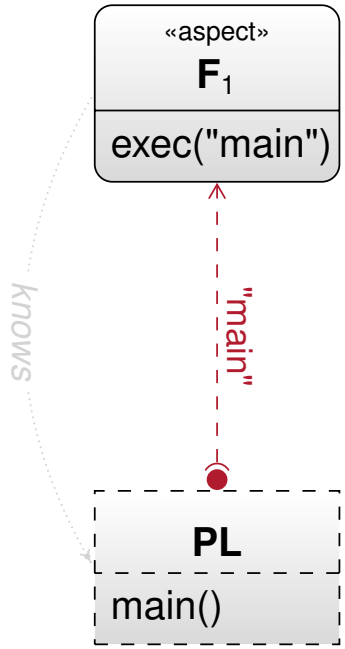

advice [of "join point"]
BaseClass
method()
explicitJP()
"explicitJP"
method is explicit join point
knows
implied relationship
introduction [of Slice]
elements belong to same concern
SomeClass
#protected()
static()
unadvisable()
ASlice
AnAspect
exec("explicitJP")
intro("SomeClass")
«slice»
ASlice
state_variable_
anotherMethod()
static
call [of method]
method is unadvisable

# AOP Mechanisms Demystified: "Obliviousness"

**Szenario:**

Optional feature component $\mathbf{F}_1$ shall be integrated into SPL component **PL**



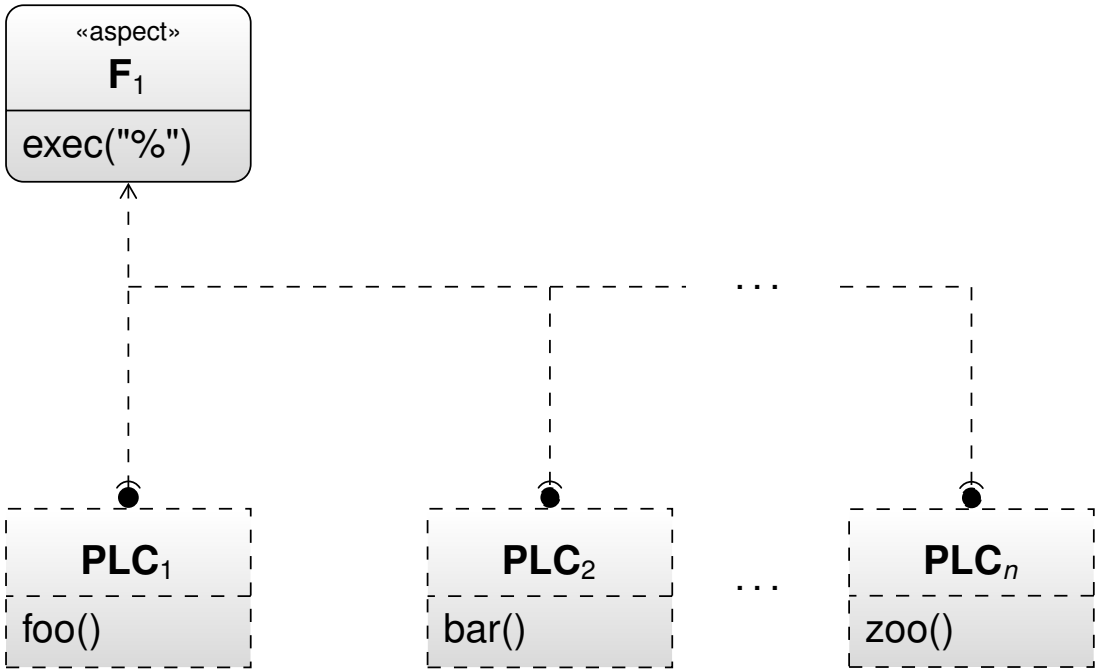




- With OOP:
  - **PL** has to call $\mathbf{F}_1 \rightsquigarrow$ **PL** has to **know** $\mathbf{F}_1$
  - control flows can only be established in the direction of knowledge
- With AOP:
  - $\mathbf{F}_1$ can give advice to **PL** $\rightsquigarrow$ $\mathbf{F}_1$ ~~has to~~ **may know PL**
  - control flow is established **opposite** to the direction of knowledge
  - binding is inherently loose $\rightsquigarrow$ silently missed, if **PL** does not exist

# AOP Mechanisms Demystified: "Quantification"

**Szenario:**

(Nonfunctional) feature component $\mathbf{F}_1$ shall be integrated into (optional) SPL components $\mathbf{PLC}_{1...n}$




- With AOP:
  - binding is inherently loose $\rightsquigarrow$ may quantify over $n$ join points
  - possible by delarative pointcut concept (here: wildcard in match expression)

# AOP Mechanisms Demystified: Summary

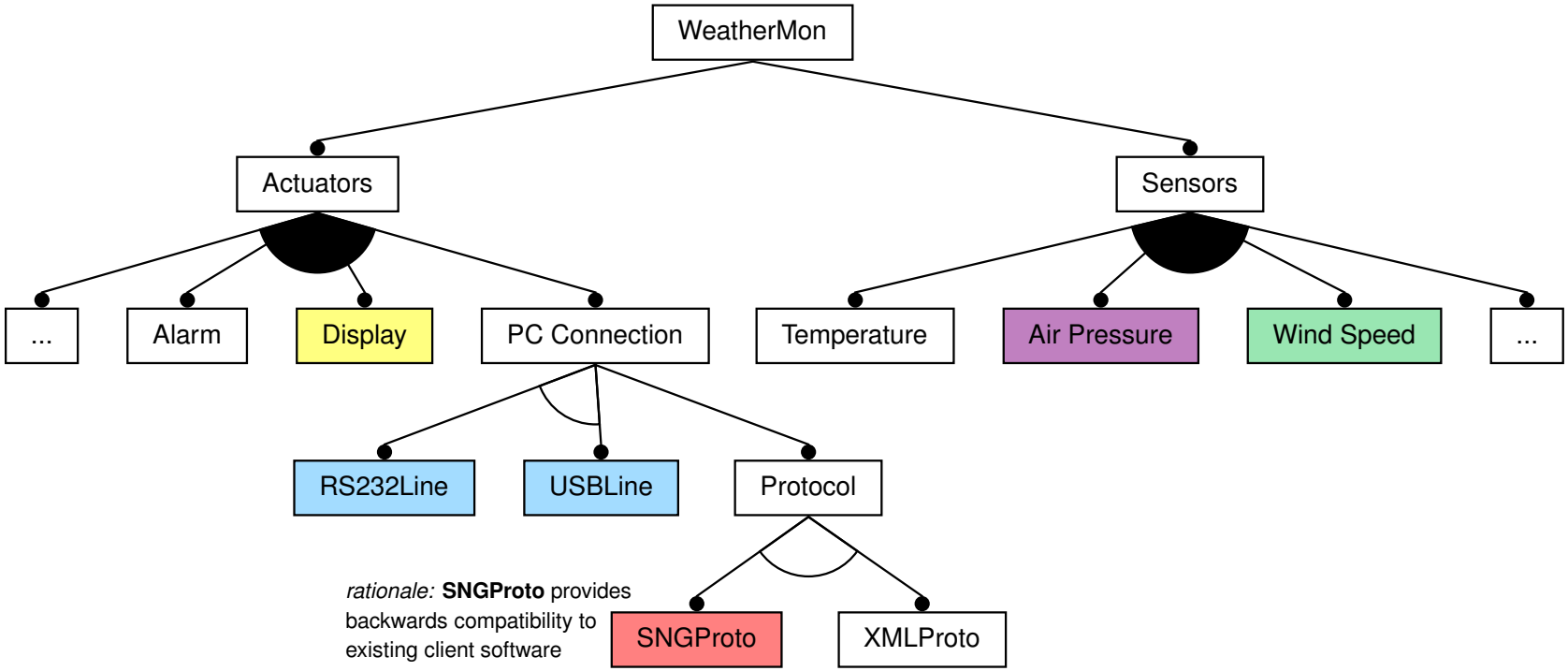
- **Advice** **inverses** the direction in which control-flow relationships are established: **C** *calls* **A** $\Longrightarrow$ **A** *advises* **C**
  - Aspects integrate themselves into the surrounding program
    $\rightsquigarrow$ "I make you call me"
  - Surrounding program can be kept oblivious of the aspects
    $\rightsquigarrow$ advice-based binding as a means to integrate (optional) features
- **Pointcuts** provide for an implicit **quantifiction** of this integration
  - Applies to $0 \ldots M \ldots n$ join points, depending on the pointcut expression
    $\rightsquigarrow$ Aspects can be kept oblivious of the surrounding program
  - Thereby, advice-based binding is inherently loose
    $\rightsquigarrow$ advice-based binding as a means to integrate interacting features

# Agenda



# I4WeatherMon: Feature Model

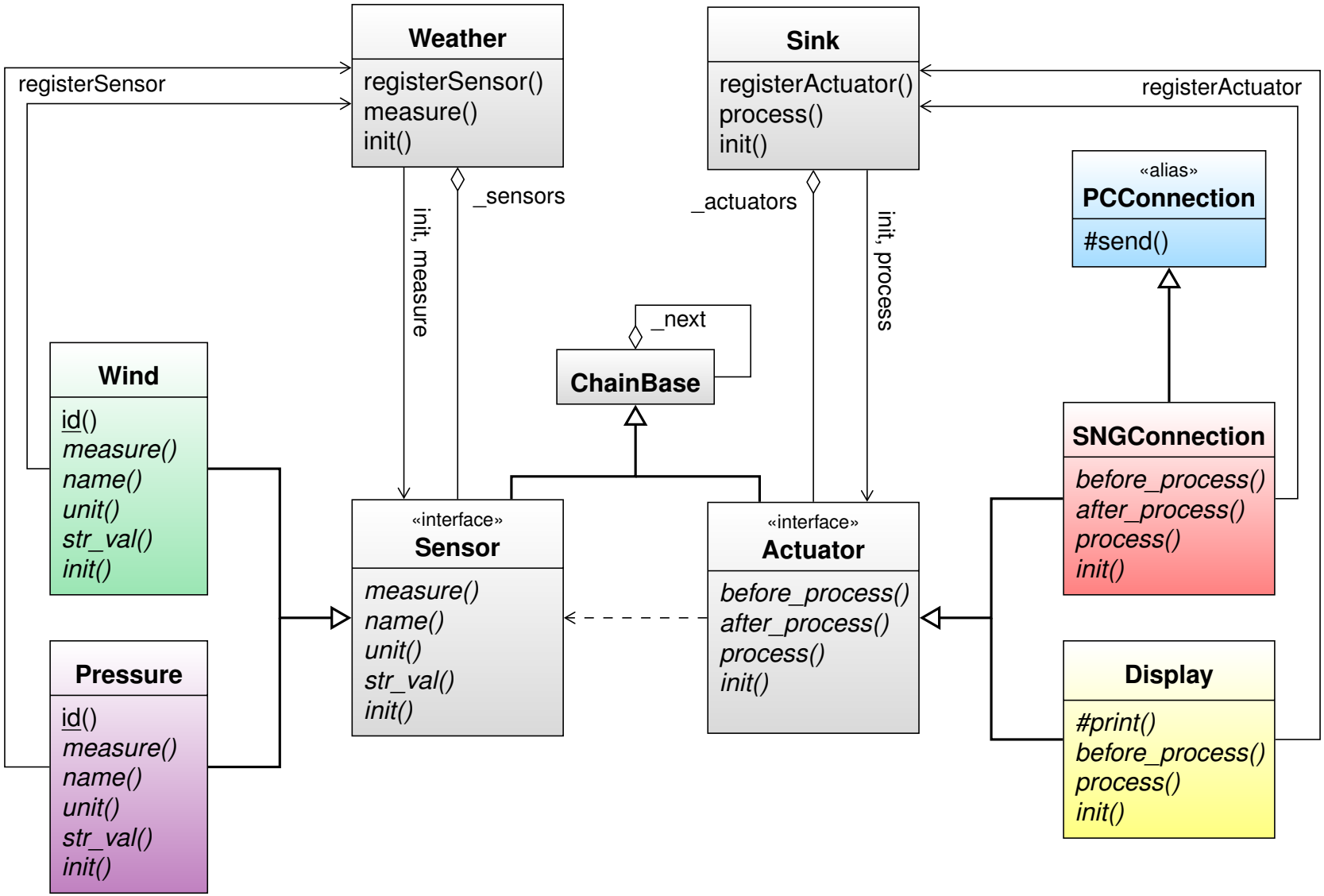




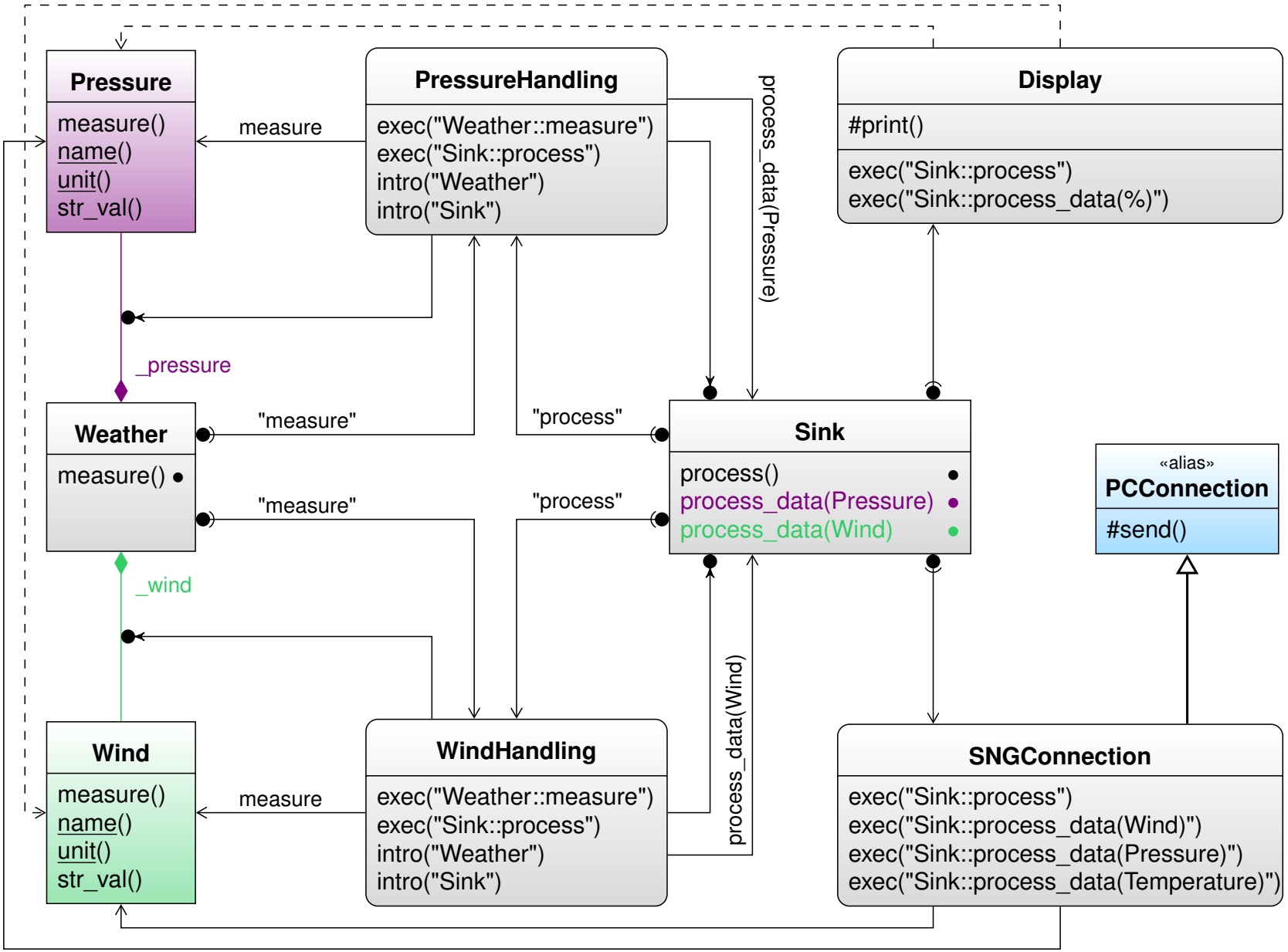

- How to achieve *Granularity*, *Economy*, *Pluggability*, *Extensibility*?
  - Configuration-dependent sensor and actuator sets
    - initialization, integration, interaction of optional feature code
  - Generic and nongeneric actuators
    - interacting optional feature code

# I4WeatherMon: **OOP** Solution Space

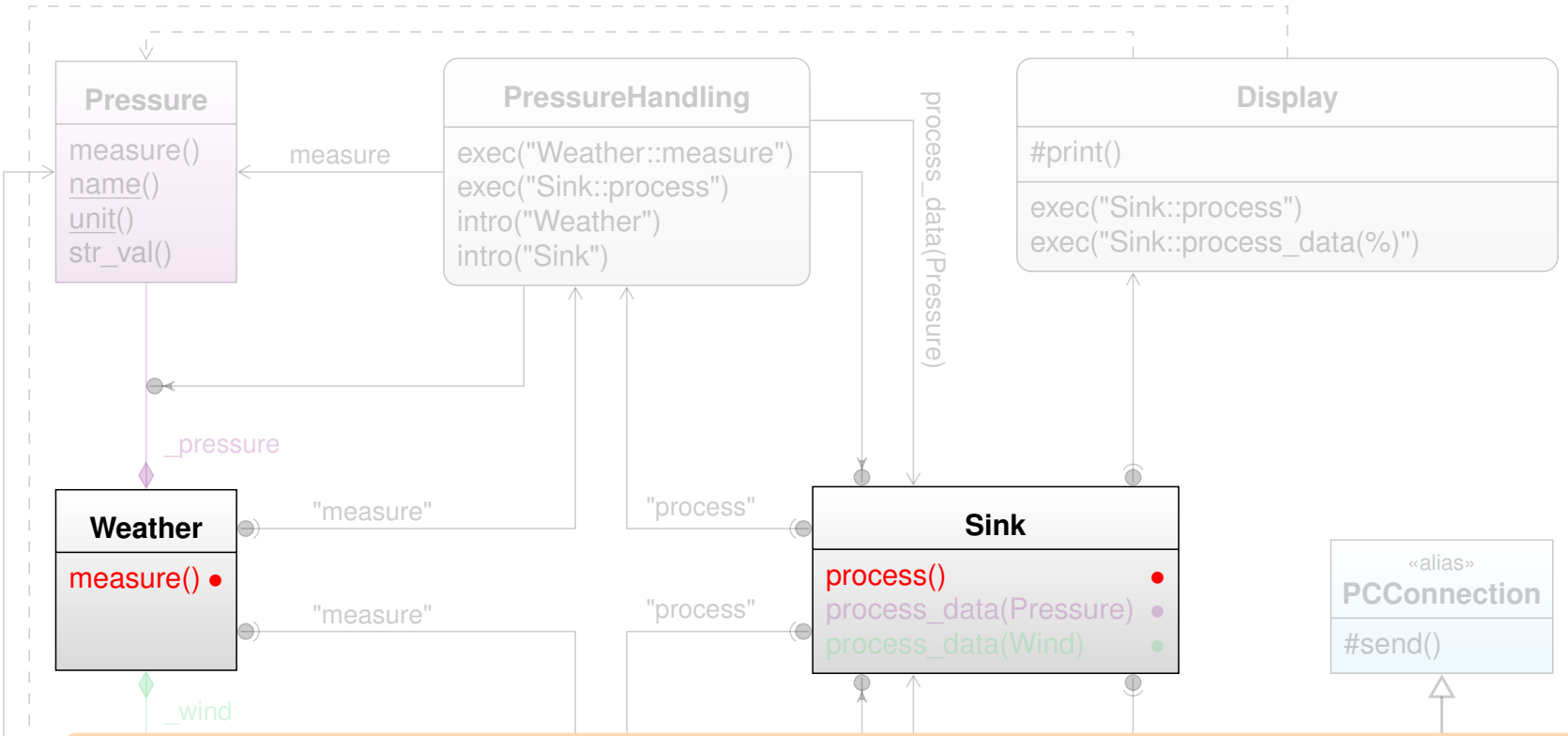




# I4WeatherMon: **AOP** Solution Space



# I4WeatherMon: **AOP** Solution Space



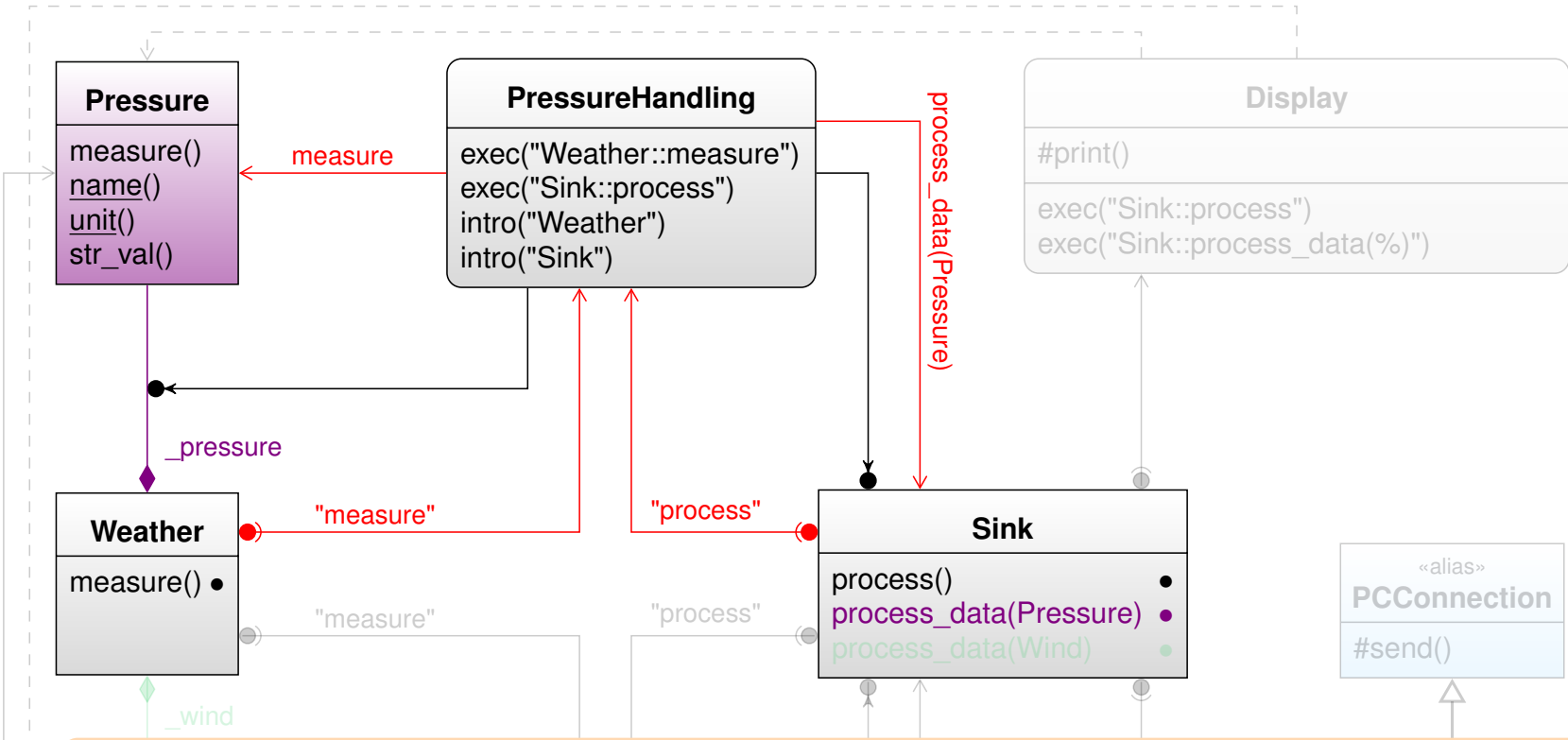

## Basic structure

`Weather` and `Sink` are (almost) empty classes.

- Provide a lexical scope for sensor / actuator introductions
- Provide explicit join points (empty methods `measure()` / `process()`) that are invoked by the main loop, when measuring / processing should take place

⇝ All further functionality is provided by the aspects!

# I4WeatherMon: **AOP** Solution Space




## Sensor integration

A *Sensor* is implemented as a class with an accompanying `Handling` aspect

- Slices the sensor singleton instance into `Weather`
- Gives advice to `Weather::measure()` to invoke *Sensor*`::measure()`
- Slices an explicit join point `process_data(`*Sensor*`)` into `Sink`
- Gives advice to `Sink::process()` to invoke `process_data(`*Sensor*`)`

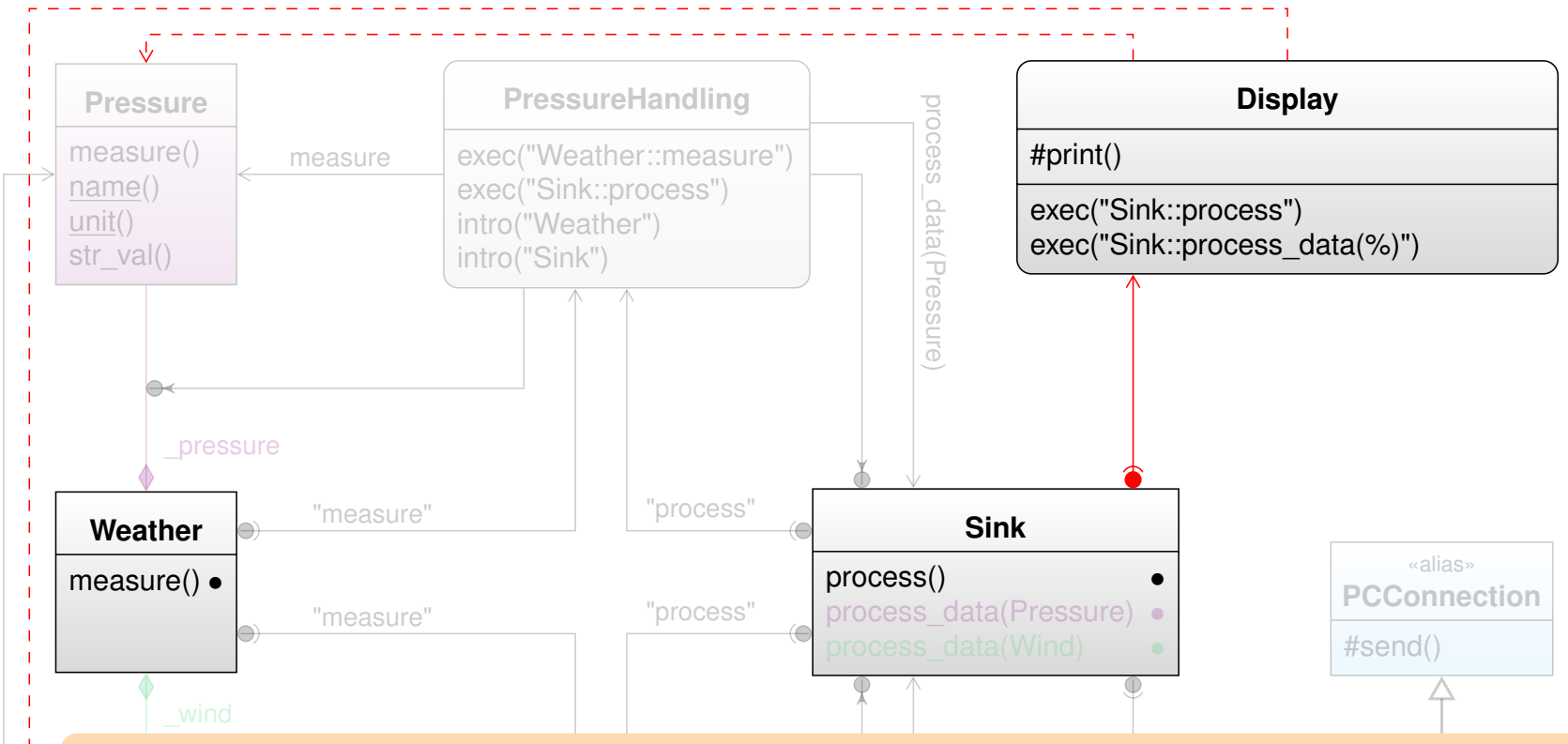
# I4WeaterMon: AOP Sensor Integration

```
class Weather {
public:
  void measure() {} // empty implementation
};
```

```
class Sink {
public:
  void process() {} // empty implementation
};
```

```
aspect PressureHandling {
  // Weather integration
  advice "Weather" : slice struct{
    Pressure _pressure; // introduce sensor instance (singleton)
  };
  advice execution( "void Weather::measure()" ) : before() {
    theWeather._pressure.measure(); // invoke sensor's measure()
  }
  // Sink integration
  advice "Sink" : slice struct {
    // introduce sensor-specific explicit join point for actuator aspects
    void process_data( const Pressure & ) {}
  };
  advice execution( "void Sink::process()" ) : after() {
    theSink.process_data( theWeather._pressure ); // trigger it
  }
};
```

# I4WeatherMon: **AOP** Solution Space




## Generic actuator integration

A generic actuator (processes all sensors) is implemented by an aspect

- Gives advice to `Sink::process()` to execute processing pre-/post actions
- Gives generic advice to all overloads of `Sink:process_data()` to invoke each sensor (typed) in order to process its data via the generic `str_val()`

⇝ Generic actuator does not know the available / possible sensor types

# I4WeaterMon: AOP Generic Actuator Integration

```
aspect Display {
  …
  // display each element of the weather data
  advice execution("void Sink::process_data(%)") : before () {
    typedef JoinPoint::template Arg<0>::ReferredType Data;
    char val[5];
    tjp->arg<0>()->str_val( val );
    print( Data::name(), val, Data::unit() );
  }
};
```

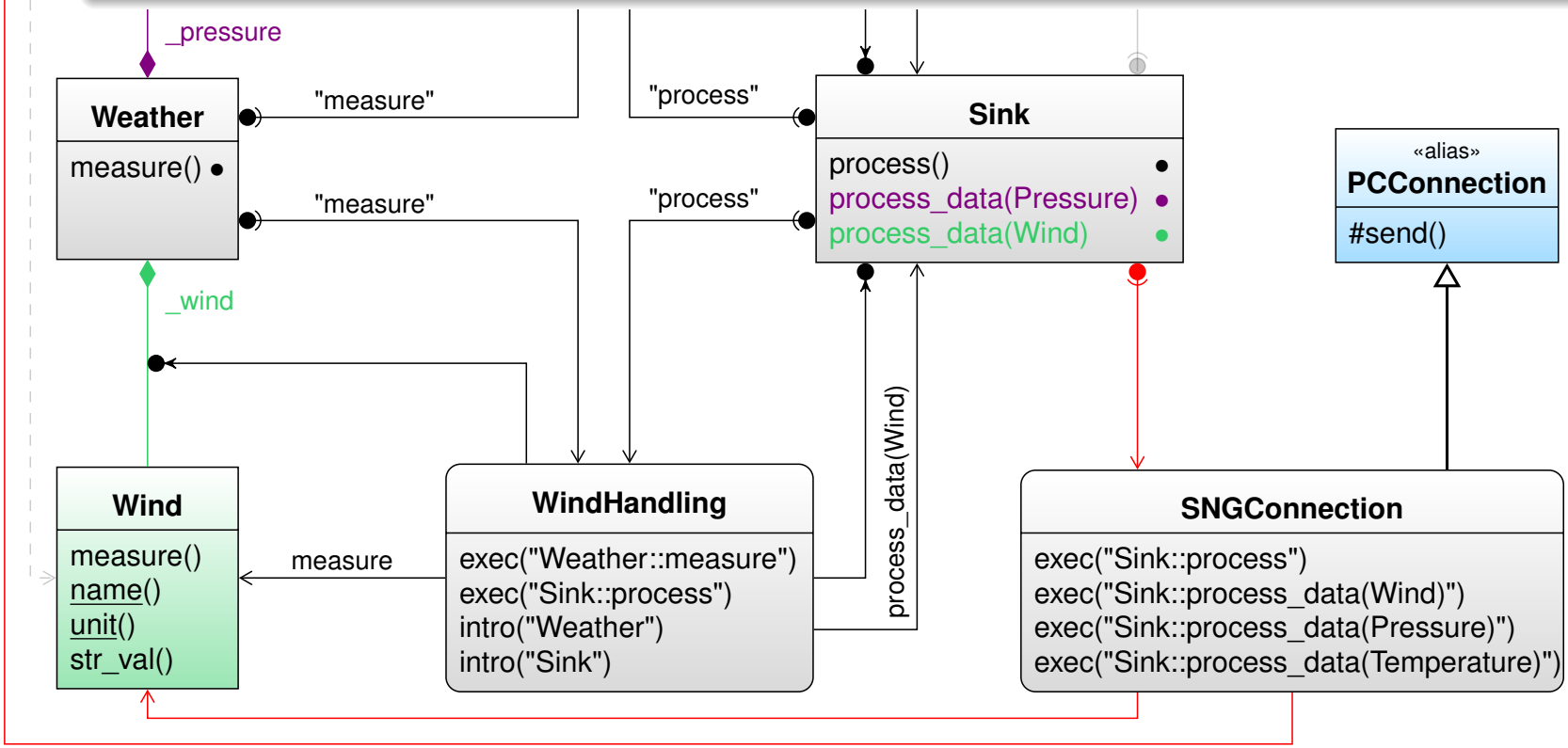
# I4WeatherMon: **AOP** Solution Space

## Nongeneric actuator integration

A nongeneric actuator (processes some sensors) is implemented by an aspect

- Gives advice to `Sink::process()` to execute processing pre-/post actions
- Gives advice to selected overloads of `Sink:process_data()` to invoke them in order to process each sensors data via a sensor-specific interface

⇝ Nongeneric actuator has to know specific sensor types



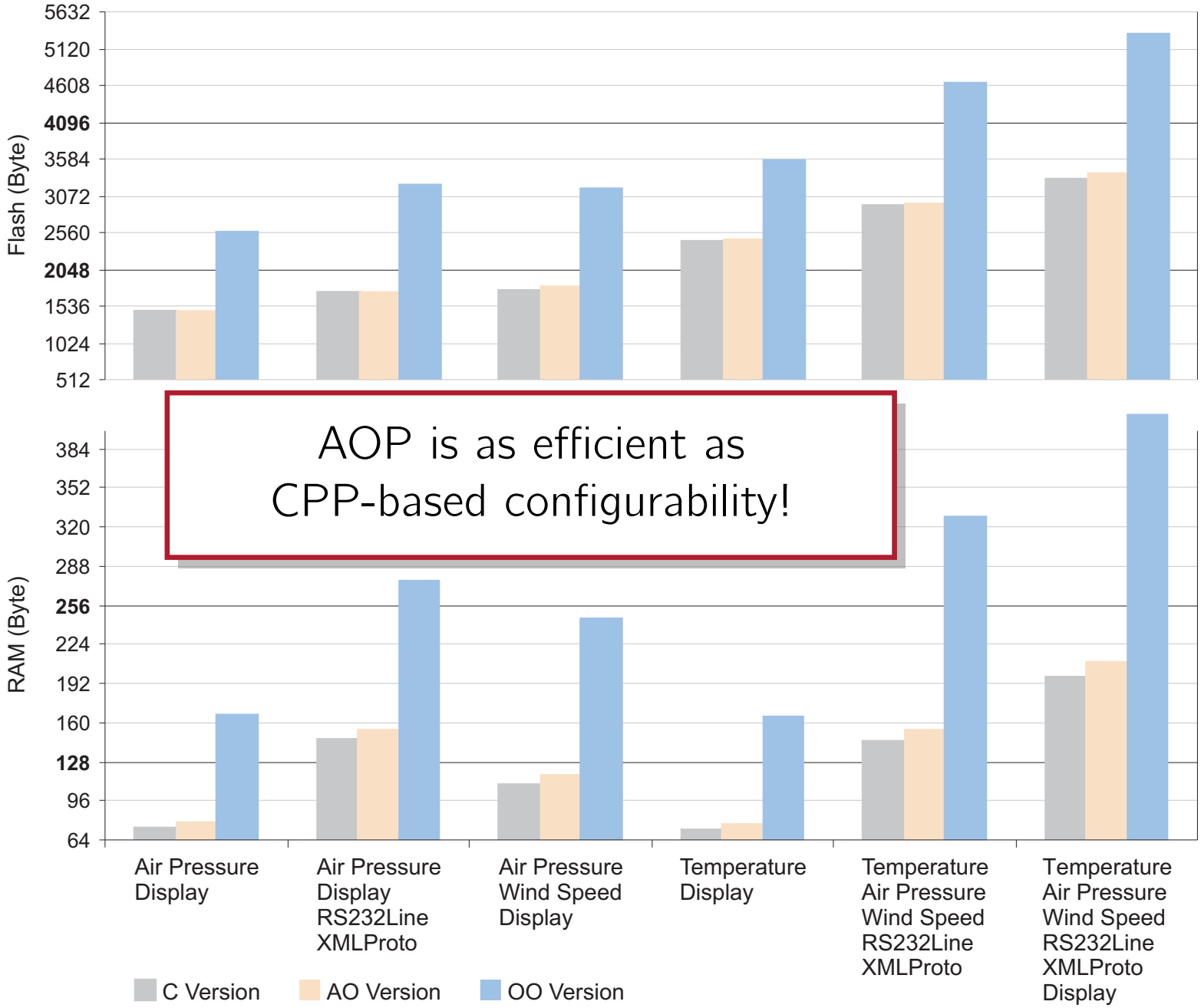
# I4WeaterMon: AOP Generic Actuator Integration

```
aspect SNGConnection : protected PCConnection {
  UInt8 _p, _w, _t1, _t2;     // weather record
  ...
  // let this aspect take a higher precedence than <Sensor>Handling
  advice process () : order ("SNGConnection", "%Handling");
  advice execution("void Sink::process(const Weather&)")
      : before () { ... /* init record */ }
  advice execution("void Sink::process(const Weather&)")
      : after () { ... /* transmit record */ }

 // collect wind, pressure, temperature data by giving specific advice
  advice execution("void Sink::process_data(...)") && args (wind)
      : before (const Wind &wind) {
    _w = wind._w;
  }
  advice execution("void Sink::process_data(...)") && args (pressure)
      : before (const Pressure &pressure) {
    _p = pressure._p - 850;
  }
  advice execution("void Sink::process_data(...)") && args (temp)
      : before (const Temperature &temp) {
    _t1 = (UInt8)temp._t1;
    _t2 = temp._t2;
  }
};
```

# I4WeaterMon: CPP vs. AOP – Footprint






# I4WeaterMon (AOP): Evaluation

## General

1. Separation of concerns (SoC) ✔
2. Resource thriftiness ✔

## Operational

3. Granularity ✔
   - Every component implements functionality of a single feature only.
4. Economy ✔
   - All control-flow bindings are established at compile time.
5. Pluggability ✔
   - Sensors and actuators integrate themselves by aspects.
6. Extensibility ✔
   - “Plug & Play” of sensor and actuator implementations.

# Agenda



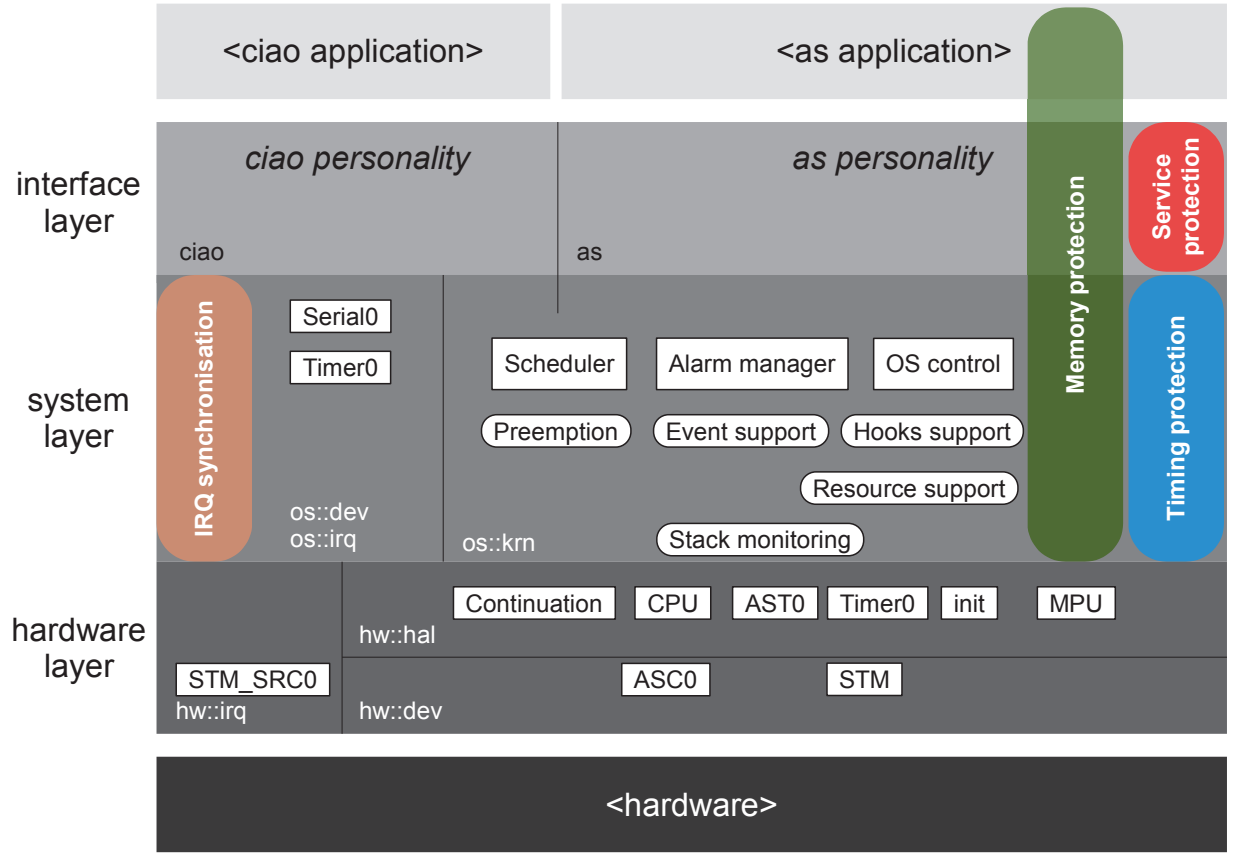
# CiAO: Motivation and Goals

> *“ Throughout the entire operating-system design cycle, we must be careful to **separate policy decisions** from implementation details (**mechanisms**). This separation allows maximum flexibility if policy decisions are to be changed later. ”*
>
> Silberschatz, Gagne, and Galvin 2005: *Operating System Concepts* [8, p. 72]

- Primary goal: architectural configurability
  - configurability of even fundamental policies
    $\mapsto$ synchronization, protection, interaction
- Secondary goal: *< the standards >*
  - efficiency, configurability in general, portability
- Approach: **aspect-aware operating system design**
  - strict decoupling of policies and mechanisms in the implementation
    $\mapsto$ using aspects as the primary composition technique

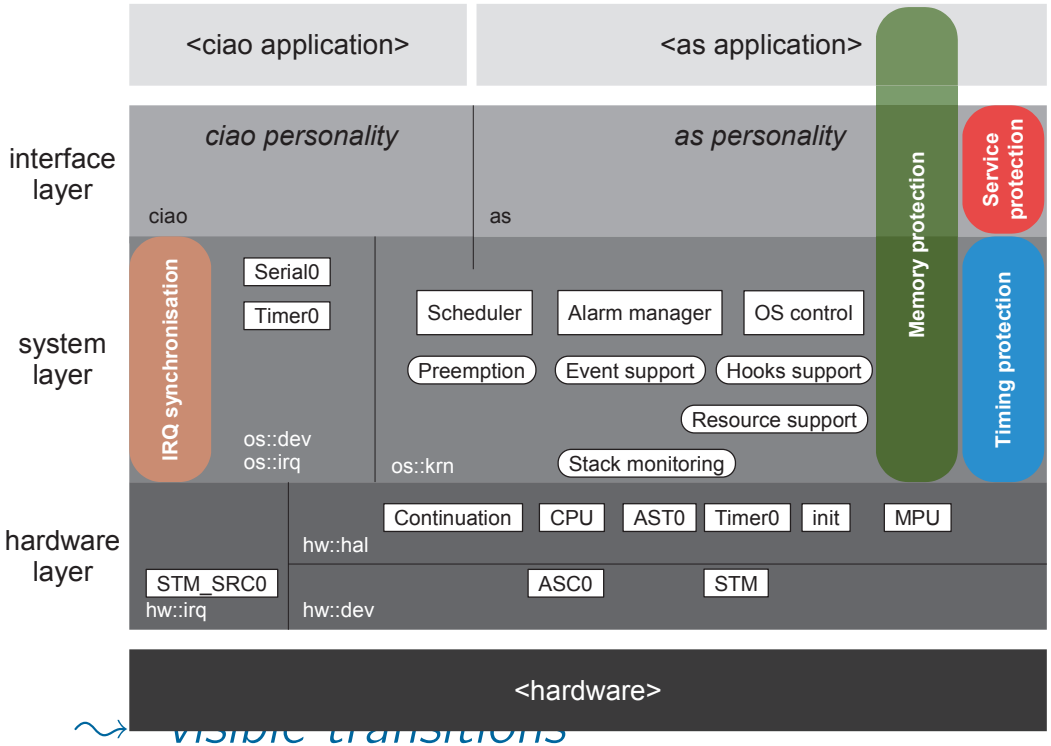
# CiAO ↦ CiAO is Aspect-Oriented

- A product line of aspect-oriented operating systems
  - Implements OSEK VDX / AUTOSAR OS [1, 7]
  - Fine-grained configurability of all system policies and abstractions
  - Developed from scratch with AOP




# CiAO: General Structure

- Layered Architecture
  - Interface layer (`as`/`ciao`)
  - System layer (`os`)
  - Hardware layer (`hw`)
- Layers $\mapsto$ C++ namespaces
  - Potential join points for cross-layer transitions
  - Further refined by sublayers (`os::krn`, `hw::irq`)
  - Layers as a means of aspect-aware development




$\rightsquigarrow$ *visible transitions*

```
// yields all hardware invocations from the system layer
pointcut OStoHW() =  call("% hw::...::%(...)")
                  && within("% os::...::%(...)");
```

# Methodology: Principles of Aspect-Aware Development

| Design Principles | $\mapsto$ | Development Idioms |
|---|---|---|
| 1. **loose coupling** | *by* | advice-based binding |
| 2. **visible transitions** | *by* | explicit join points |
| 3. **minimal extensions** | *by* | extension slices |

# Methodology: Principles of Aspect-Aware Development

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The *binding* of components, but also their *instantiation* (e.g, placement in a certain memory region) and the time and order of their *initialization* should all be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

# Methodology: Roles of Aspects and Classes

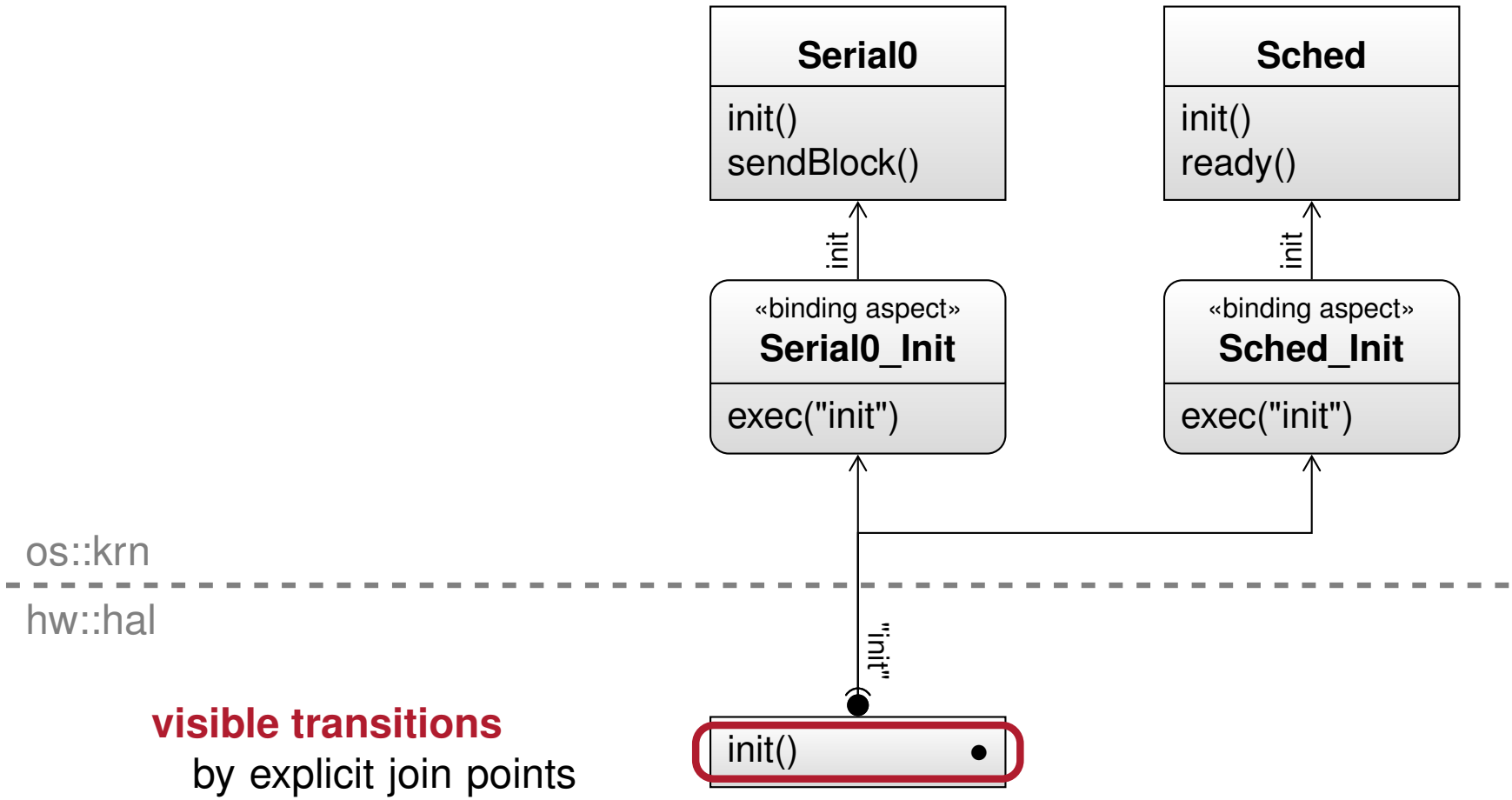
## What to model as a *class* and what as an *aspect*?

- $< \textit{Thing} >$ is a class if – and only if – it
  is a distinguishable, instantiable concept of CiAO:
  - A **system component**, instantiated internally on behalf of CiAO
    - The Scheduler, the Alarm Manager, the OS control facility, ...
    - Hold and manage kernel state, singletons by definition
  - A **system abstraction**, instantiated as objects on behalf of the user
    - Task, Event, Resource, ...
    - In AUTOSAR OS: instantiated at compile time
  - Both are sparse $\rightsquigarrow$ provide a minimal implementation only
- Otherwise $< \textit{thing} >$ is an aspect!

# Methodology: Roles of Aspects and Classes (Cont'd)

- Three idiomatic aspect roles
  - **Extension aspects:** extend some system component or system abstraction by additional functionality.
  - **Policy aspects:** "glue" otherwise unrelated system abstractions or components together to implement some CiAO kernel policy.
  - **Upcall aspects:** bind behavior defined by higher layers to events produced in lower layers of the system.

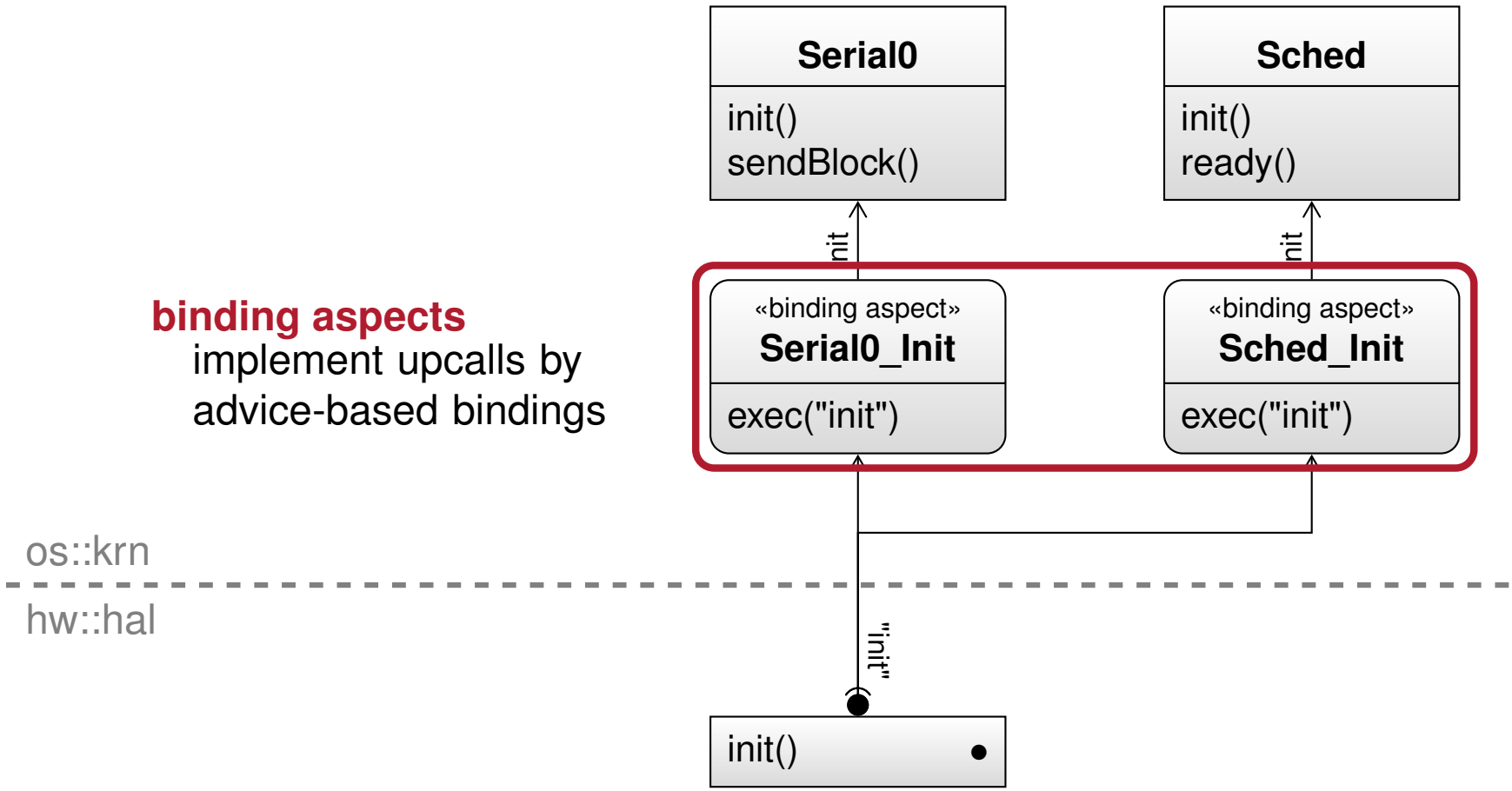
# Example: Mechanism Integration



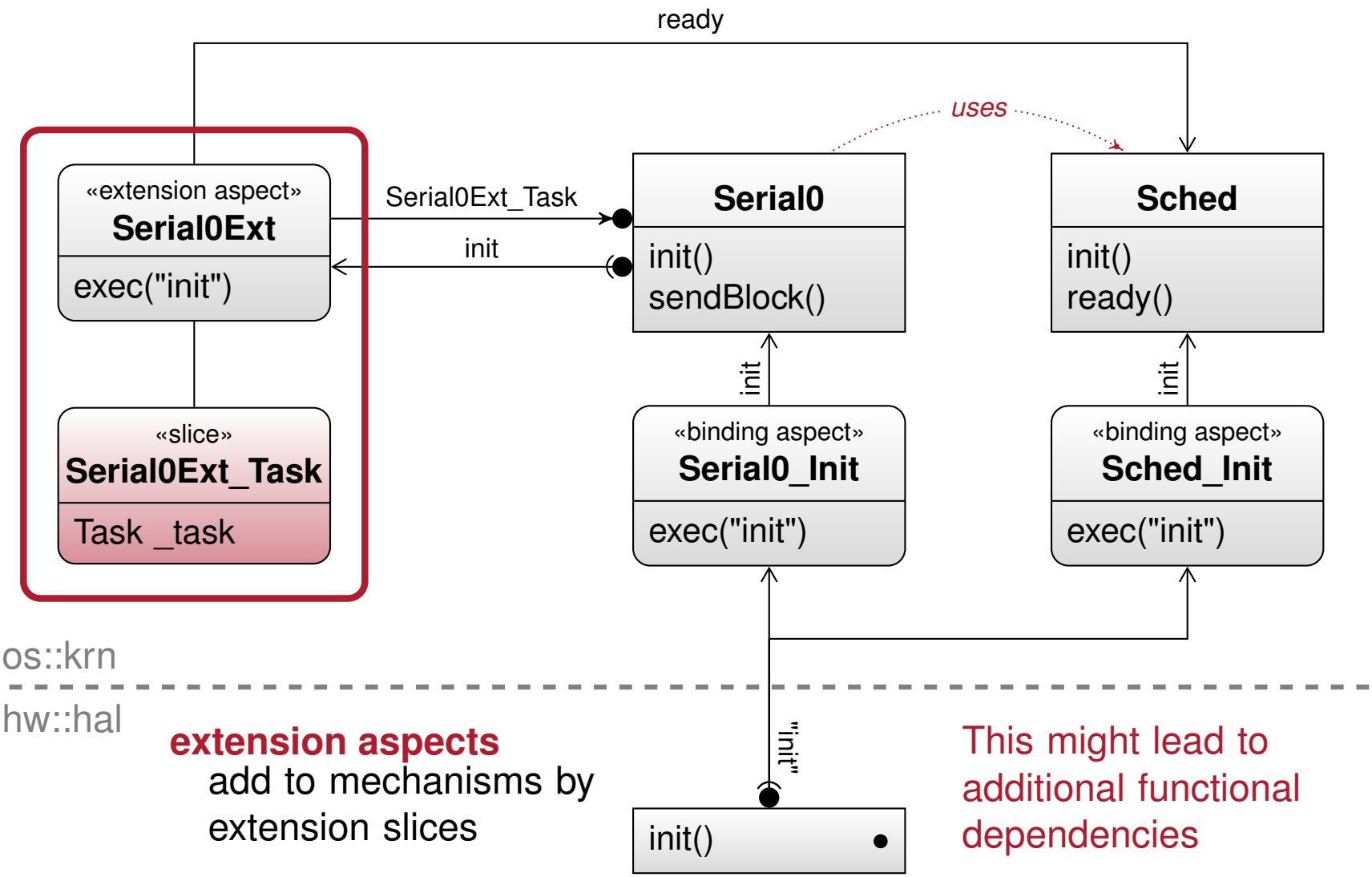

**visible transitions**
by explicit join points

# Example: Mechanism Integration

**binding aspects**
implement upcalls by
advice-based bindings



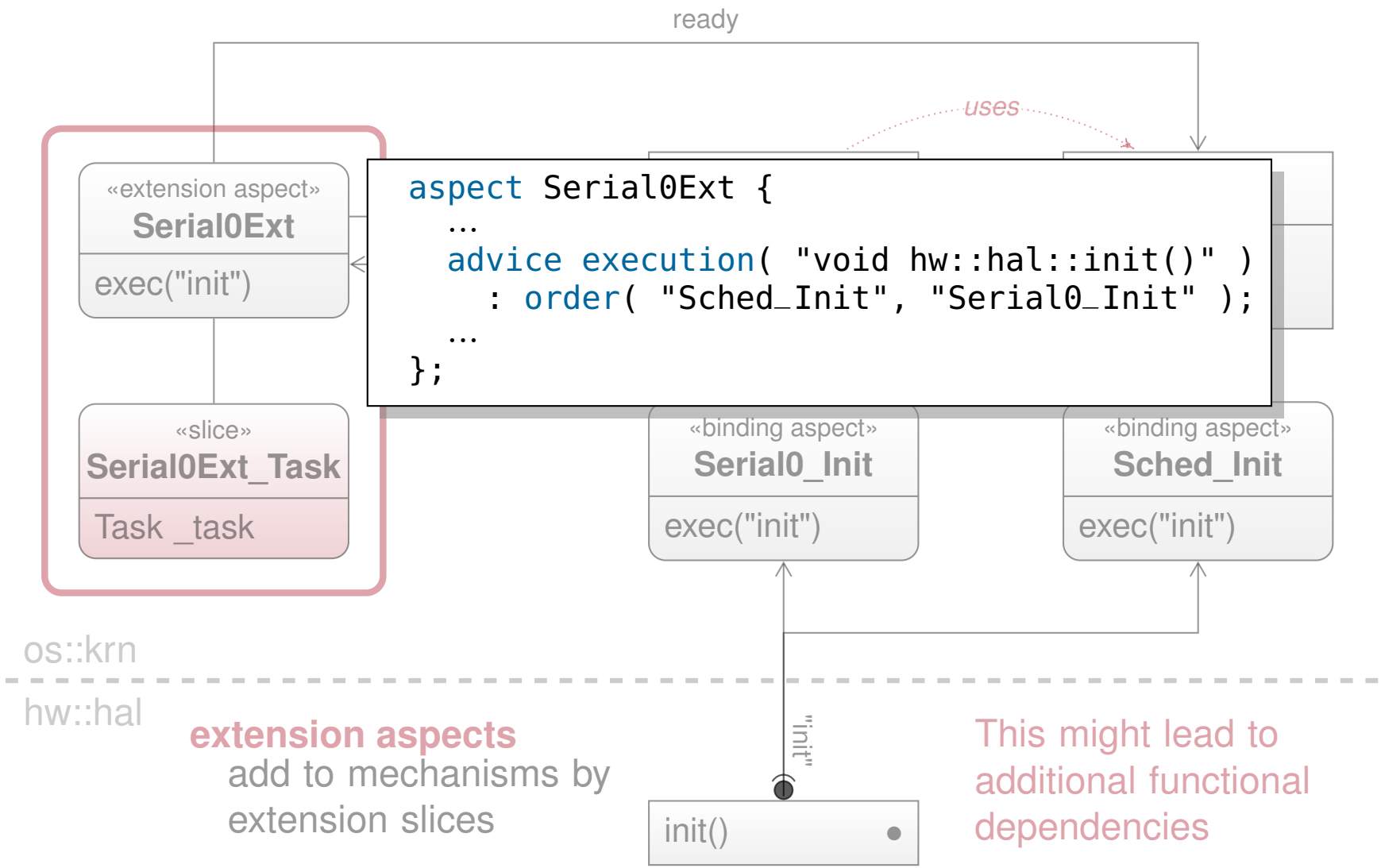

# Example: Mechanism Integration




**extension aspects**
add to mechanisms by extension slices

This might lead to additional functional dependencies

# Example: Mechanism Integration

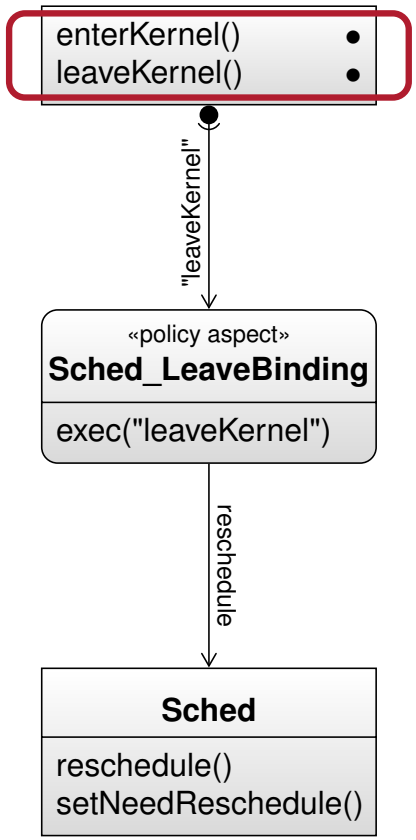



**extension aspects**
add to mechanisms by extension slices

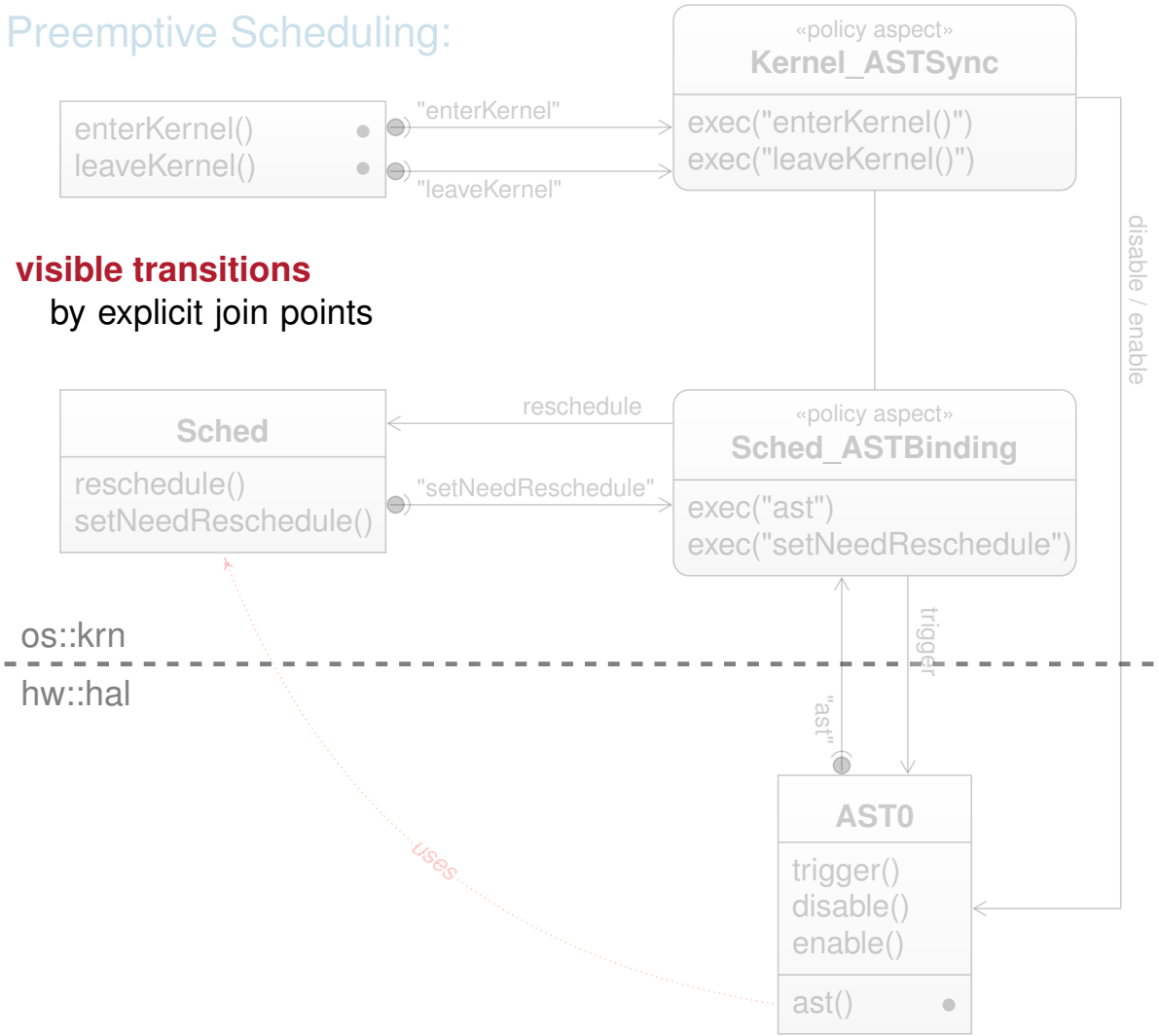
This might lead to additional functional dependencies

# Example: Policy Integration

## Cooperative Scheduling:

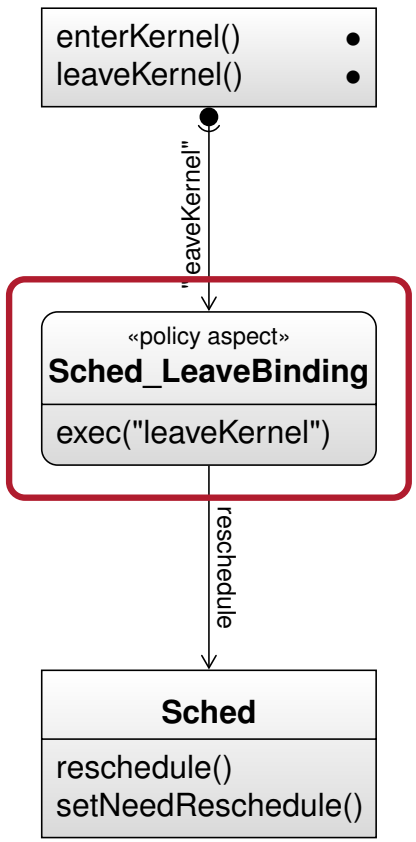



## Preemptive Scheduling:

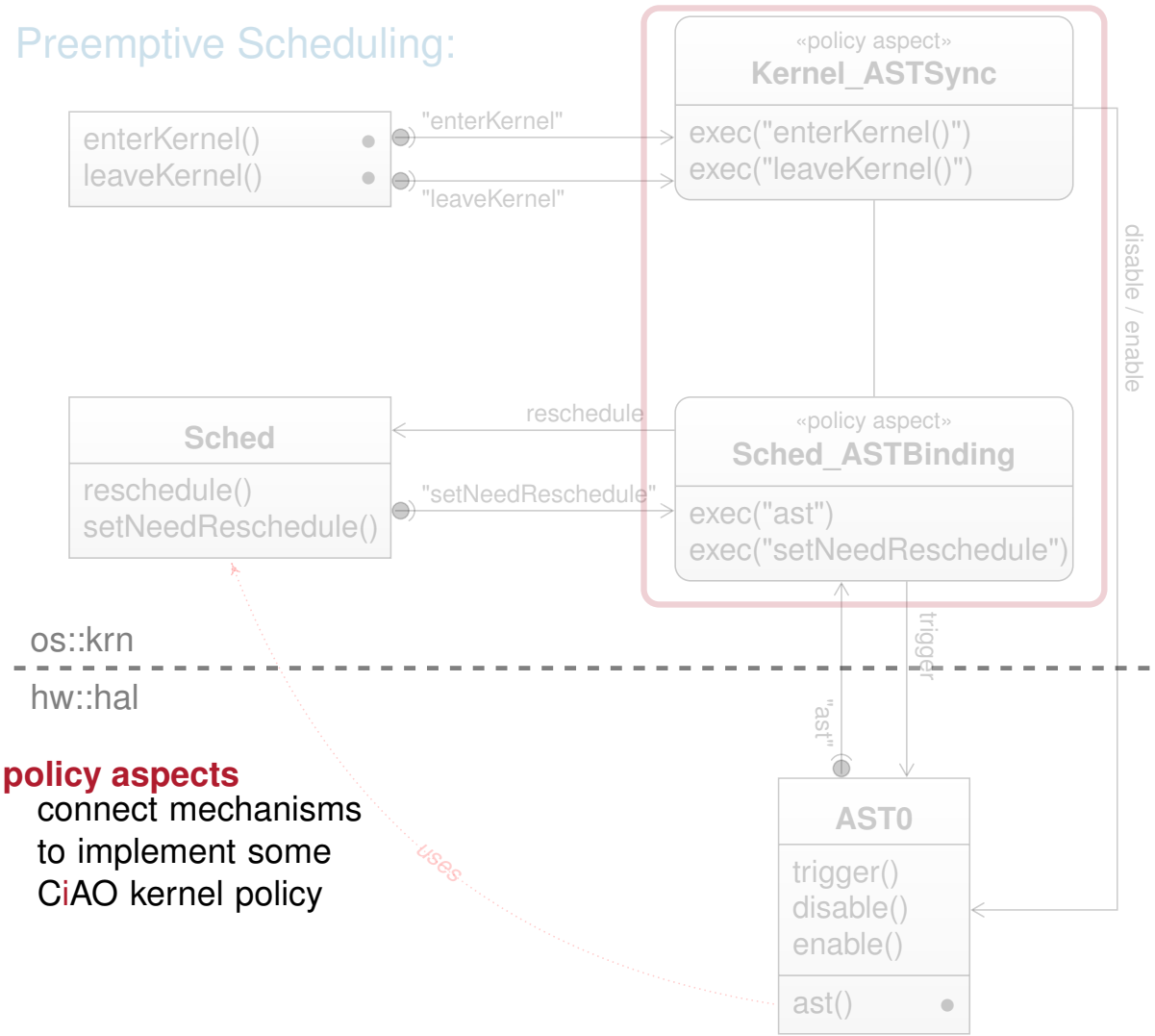



# Example: Policy Integration

## Cooperative Scheduling:



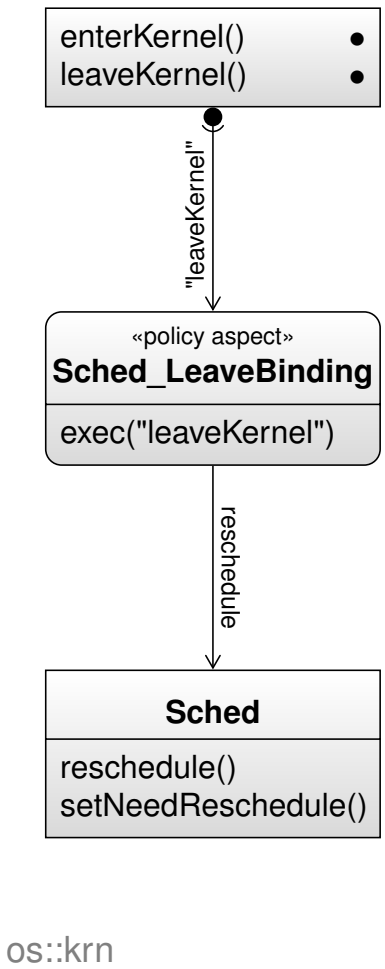

## Preemptive Scheduling:



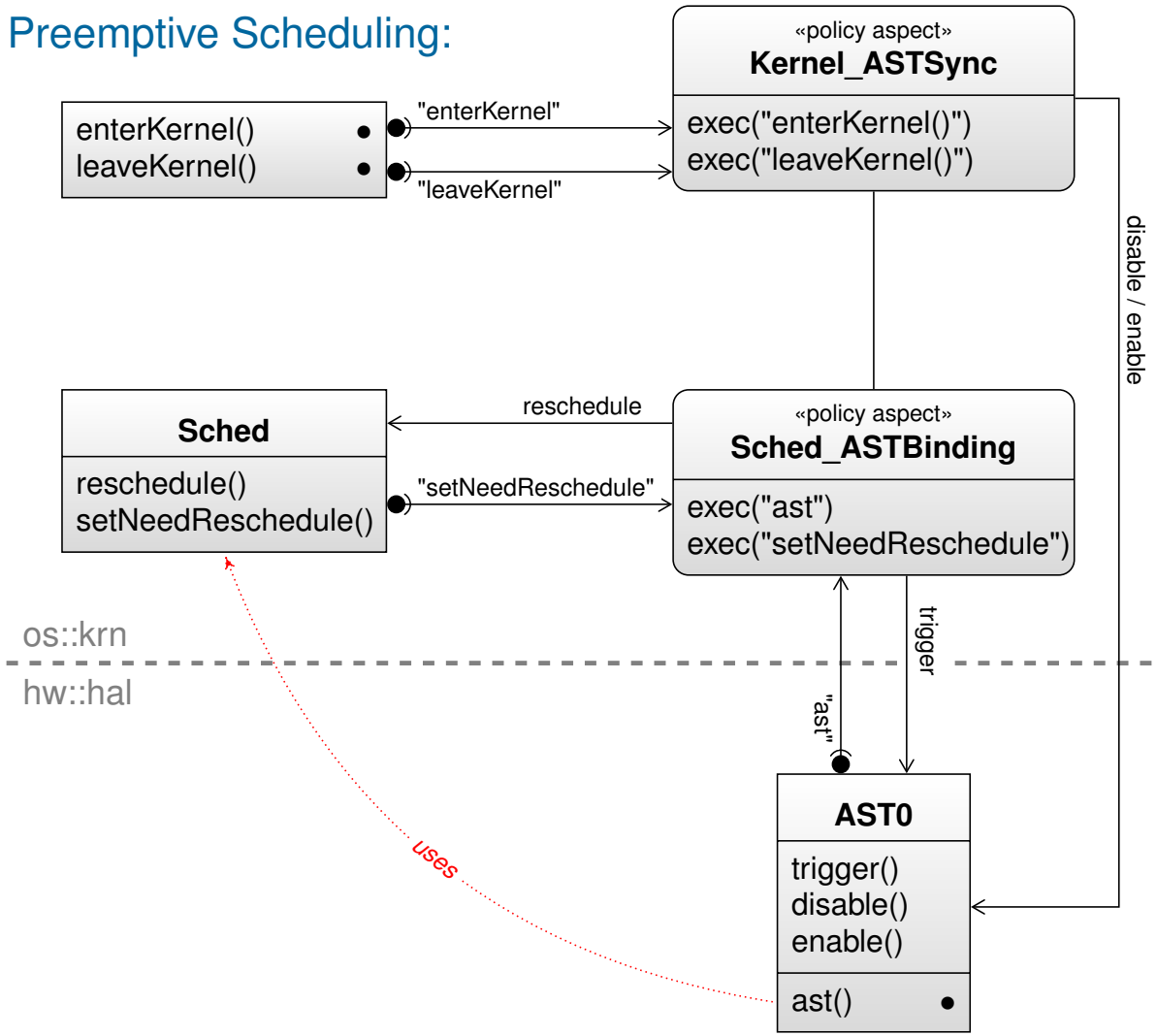

**policy aspects**
connect mechanisms to implement some CiAO kernel policy

# Example: Policy Integration

## Cooperative Scheduling:




## Preemptive Scheduling:



# Methodology: Explicit Join Points

- Advice-based binding $\mapsto$ availability of the "right" join points
  - for all semantically important transitions in the system
  - statically evaluable
- Fine-grained component structure $\leadsto$ many implicit join points, but
  - amount and precise semantics often implementation dependent
  - aspects have to "know" $\leadsto$ no obliviousness
- Important transitions not available for technical reasons as JPs
  - target code may be fragile (e.g., context switch) $\leadsto$ must not be advised
  - target code may be written in assembly $\leadsto$ transitions not visible as JPs

# Methodology: Explicit Join Points (Cont'd)

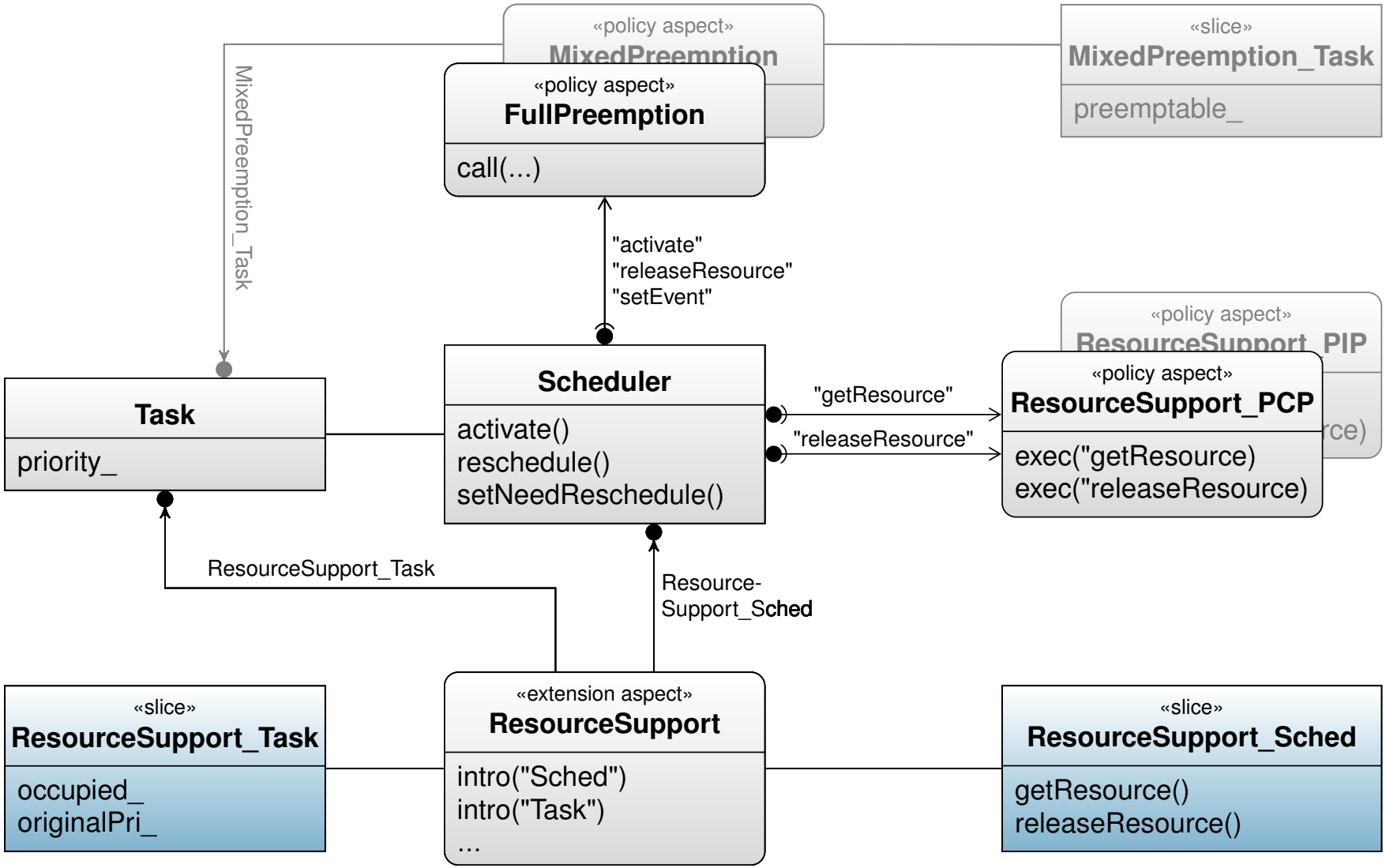
- Solution: explicit join points
  - empty inline methods for the sole purpose that aspects can bind to them
  - explicitly triggered by components or other aspects

  $\leadsto$ well defined semantics

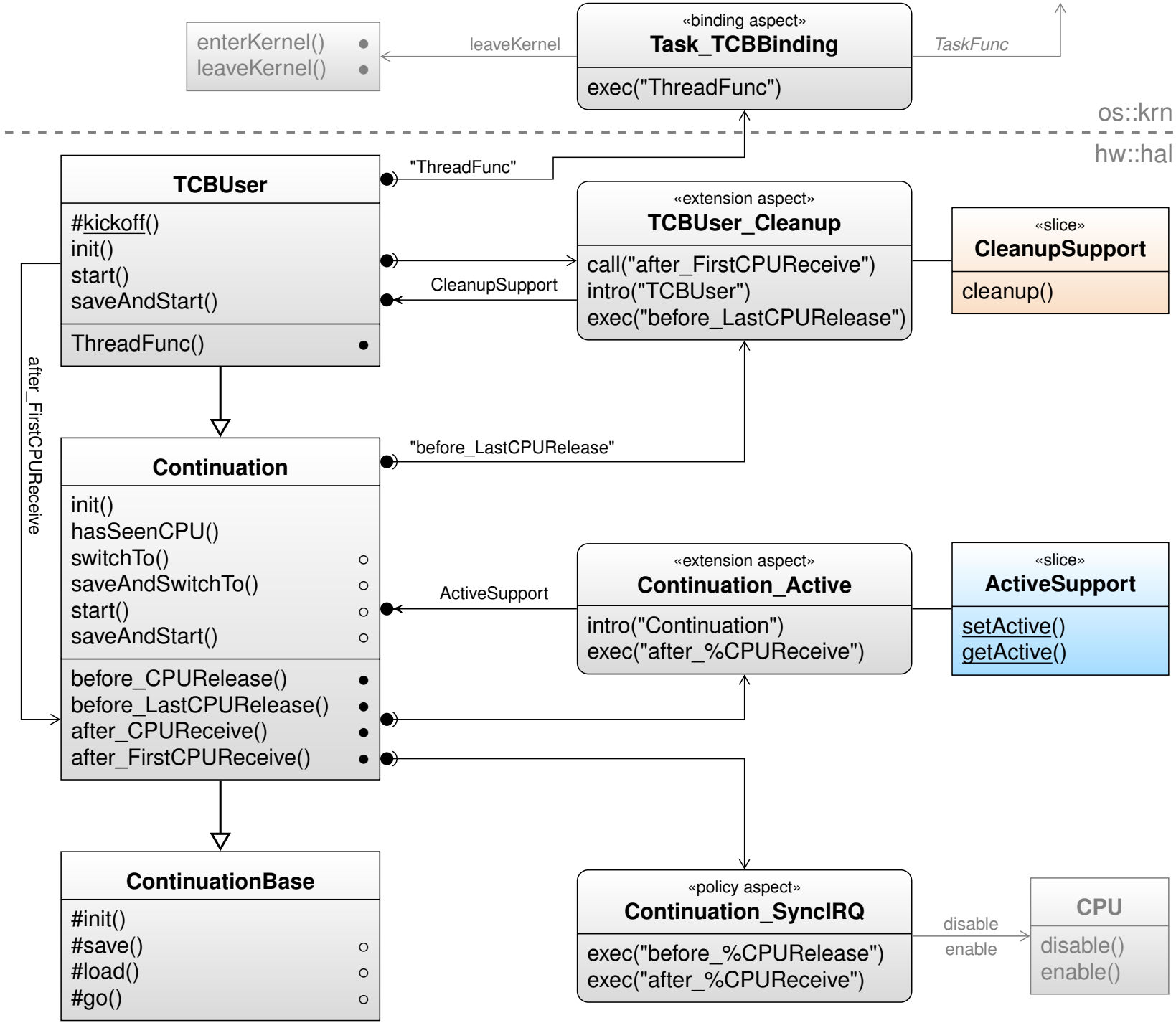
- Upcall join points (U) represent system-internal events that are to be processed on a higher layer
  - exceptions, such as signals or interrupts
  - internal events, such as system intitalization or entering of the idle state
- Transition join points (T) represent semantically important control-flow transitions inside the kernel
  - level transitions: *user* $\rightarrow$ *kernel*, *user* $\rightarrow$ *interrupt*
  - context transitions: *threadA* $\rightarrow$ *threadB*

| type | representing function or method | description |
|---|---|---|
| os::krn | U `interalErrorHook()`<br>U `internalProtectionHook(StatusType error)`<br>U `internalPreTaskHook()`<br>U `internalPostTaskHook()`<br>U `internalStartupHook()`<br>U `internalShutdownHook()` | Explicit join points for the support and binding of OSEK OS and AUTOSAR OS user-level hook functions, as specified in [36, p. 39] and [4, p. 46]. Triggered in case of an *error*, a *protection* violation, before (*pre*) and after (*post*) at high-level task switch, and at operating-system startup and shutdown time. |
| | T `enterKernel()`<br>T `leaveKernel()`<br>... | Triggered when a control flow enters (respectively leaves) the kernel domain. |
| hw::hal | U `ThreadFunc()` | Entry point of a new thread (continuation). |
| | T `before_CPURelease(Continuation*& to)`<br>T `before_LastCPURelease(Continuation*& to)` | Triggered immediately before the running continuation is deactivated or terminated; `to` is going to become the next running continuation. |
| | T `after_CPUReceive()`<br>T `after_FirstCPUReceive()` | Triggered immediately after the (new) running continuation got reactivated or started. |
| | U `AST<#>::ast()` | Entry point of the respective AST. |
| | U `init()` | Triggered during system startup after memory busses and stack have been initialized. |
| | ... | |
| hw::irq | U `<IRQ_NAME>::handler()` | Entry point of the respective interrupt handler. (Interrupts are still disabled.) |
| | ... | |

# Example: Optional Feature Interaction [4]




# Continuation: CiAO's Thread Abstraction [5]



# Agenda



# AUTOSAR-OS Features Implemented in CiAO as Aspects (Excerpt)

| concern | extension | policy | upcall | advice | join points | extension of \| advice-based binding to |
|---|---|---|---|---|---|---|
| ISR cat. 1 support | 1 | | $m$ | $2+m$ | $2+m$ | API, OS control \| $m$ ISR bindings |
| ISR cat. 2 support | 1 | | $n$ | $5+n$ | $5+n$ | API, OS control, scheduler \| $n$ ISR bindings |
| ISR abortion support | 1 | | | 2 | $1+m+n$ | scheduler \| $m+n$ ISR functions |
| Resource support | 1 | 1 | | 3 | 5 | scheduler, API, task \| PCP policy implementation |
| Resource tracking | | 1 | | 3 | 4 | task, ISR \| monitoring of Get/ReleaseResource |
| Event support | 1 | | | 5 | 5 | scheduler, API, task, alarm \| trigger action JP |
| Alarm support | 1 | | | 1 | 1 | API |
| OS application support | 1 | | | 2 | 3 | scheduler, task, ISR |
| Full preemption | | 1 | | 2 | 6 | \| 3 points of rescheduling |
| Mixed preem | | | | | | |
| Multiple act | | | | | | |
| Stack monitoring | | 1 | | 2 | 3 | task \| CPU-release JPs |
| Context check | | 1 | | 1 | $s$ | \| $s$ service calls |
| Disabled interrupts check | | 1 | | 1 | 30 | \| all services except interrupt services |
| Enable w/o disable check | | 1 | | 3 | 3 | \| enable services |
| Missing task end check | | 1 | | 1 | $t$ | \| $t$ task functions |
| Out of range check | | 1 | | 1 | 4 | \| alarm set and schedule table start services |
| Invalid object check | | 1 | | 1 | 25 | \| services with an OS object parameter |
| Error hook | | | 1 | 2 | 30 | scheduler \| 29 services |
| Protection hook | 1 | 1 | | 2 | 2 | API \| default policy implementation |
| Startup / shutdown hook | | | 1 | 2 | 2 | \| explicit hooks |
| Pre-task / post-task hook | | | 1 | 2 | 2 | \| explicit hooks |

Plug & Play of optional features and policies!

# Kernel Latency Comparison with "OSEK" [4]

| test scenario | CiAO | | OSEK |
|---|---|---|---|
| | min | full | min |
| **(a)** voluntary task switch | *160* | *178* | *218* |
| **(b)** forced task switch | *108* | *127* | *280* |
| **(c)** preemptive task switch | *192* | *219* | *274* |
| **(d)** system startup | *194* | *194* | *399* |
| **(e)** resource acquisition | *19* | *56* | *54* |
| **(f)** resource release | *14* | *52* | *41* |
| **(g)** resource release with preemption | *240* | *326* | *294* |
| **(h)** category 2 ISR latency | *47* | *47* | *47* |
| **(i)** event blocking with task switch | *141* | *172* | *224* |
| **(j)** event setting with preemption | *194* | *232* | *201* |
| **(k)** comprehensive application | *748* | *748* | *1216* |

Execution time [clock ticks] on TC1796@50 MHz

(ac++1.0pre3 with tricore-g++3.4.3 -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections)

# Kernel Latency Comparison with "OSEK" [4]

| test scenario | CiAO min | CiAO full | OSEK min |
|---|---|---|---|
| **(a)** voluntary task switch | *160* | *178* | *218* |
| **(b)** forced task switch | *108* | *127* | *280* |
| **(c)** pr | | | |
| **(d)** sy | | | |
| **(e)** re | | | |
| **(f)** re | | | |
| **(g)** resource release with preemption | *240* | *328* | *294* |
| **(h)** category 2 ISR latency | *47* | *47* | *47* |
| **(i)** event blocking with task switch | *141* | *172* | *224* |
| **(j)** event setting with preemption | *194* | *232* | *201* |
| **(k)** comprehensive application | *748* | *748* | *1216* |

CiAO outperforms the marked leader in 11 out of 12 cases by up to **260 percent**.

Execution time [clock ticks] on TC1796@50 MHz

(ac++1.0pre3 with tricore-g++3.4.3 -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections)

# Feature Granularity of CiAO
## (static memory demands per feature)

| feature | with feature or instance | text | data | bss |
|---|---|---|---|---|
| *Base system (OS control and tasks)* | | | | |
| | per task | *+ func* | *+ 20* | *+ 16 + stack* |
| | per application mode | *0* | *+ 4* | *0* |
| ISR cat. 1 support | | *0* | *0* | *0* |
| | per ISR | *+func* | *0* | *0* |
| | per disable–enable | *+ 4* | *0* | *0* |
| Resource support | | *+ 128* | *0* | *0* |
| | per resource | *0* | *+ 4* | *0* |
| | per task | *0* | *+ 8* | *0* |
| | per alarm | *0* | *+ 12* | *0* |
| Full preemption | | *0* | *0* | *0* |
| | per join point | *+ 12* | *0* | *0* |
| Mixed preemption | | *0* | *0* | *0* |
| | per join point | *+ 44* | *0* | *0* |
| | per task | *0* | *+ 4* | *0* |
| Wrong context check | | *0* | *0* | *0* |
| | per void join point | *0* | *0* | *0* |
| | per StatusType join point | *+ 8* | *0* | *0* |
| Interrupts disabled check | | *0* | *0* | *0* |
| | per join point | *+ 64* | *0* | *0* |
| Invalid parameters check | | *0* | *0* | *0* |
| | per join point | *+ 36* | *0* | *0* |
| Error hook | | *0* | *0* | *+ 4* |
| | per join point | *+ 54* | *0* | *0* |
| Startup hook or shutdown hook | | *0* | *0* | *0* |
| Pre-task hook or post-task hook | | *0* | *0* | *0* |

CiAO achieves excellent granularity!

# Discussion: Aspect-Aware Development

- By AAD CiAO achieves excellent properties [3–5]
  - configurability and granularity even for fundamental kernel policies
  - complete separation of concerns in the implementation
- The approach has also been applied to other system software
  - PUMA, the C/C++ transformation framework behind `ac++` [9]
  - $\text{CiAO}_{\text{IP}}$, an aspect-oriented IP stack for embedded systems [2]
- Issues: comprehensibility & tool support
  - CiAO: aspect code/base code = 1/2.4
    $\rightsquigarrow$ *where the heck xyz is implemented?*
  - calls for additional tool support
  - `ac++` weaver implementation is stable, but not as mature as `gcc`
    - missing or confusing error messages
    - no support for weaving in template code
    - no C++0x support

| | Base code | | Aspect code | |
|---|---|---|---|---|
| | Files | LOC | Files | LOC |
| CiAO kernel only | *423* | *21,086* | *333* | *5,923* |
| CiAO COM | *112* | *8,689* | *297* | *5,552* |
| CiAO IP stack | *45* | *5,038* | *96* | *3,230* |
| CiAO overall | *580* | *34,813* | *726* | *14,705* |

This is *research*, after all :-)

# Agenda



# Summary

- Aspect-Aware Development exploits AOP mechanisms to achieve separation of concerns in configurable system software
  - Advice inverses the direction in which control-flow relationships are established: C *calls* A $\Longrightarrow$ A *advises* C
    - $\rightsquigarrow$ advice-based binding as a means to integrate (optional) features
  - Pointcuts provide for an implicit quantifiction of this integration
    - $\rightsquigarrow$ advice-based binding as a means to integrate interacting features
- CiAO applies these concepts from the very beginning
  - loose coupling by advice-based binding
  - visible transitions by explicit join points
  - minimal extensions by extension slices
- The results are compelling
  - configurability of even all fundamental system policies
  - excellent granularity and footprint

# Referenzen


[1] AUTOSAR. *Specification of Operating System (Version 2.0.1)*. Tech. rep. Automotive Open System Architecture GbR, June 2006.

[2] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack". In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. (Low Wood Bay, Lake District, UK). New York, NY, USA: ACM Press, June 2012, pp. 435–448. ISBN: 978-1-4503-1301-8. DOI: `10.1145/2307636.2307676`.

[3] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, et al. "Aspect-Aware Operating-System Development". In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*. (Porto de Galinhas, Brazil). Ed. by Shigeru Chiba. New York, NY, USA: ACM Press, 2011, pp. 69–80. ISBN: 978-1-4503-0605-8. DOI: `10.1145/1960275.1960285`.

[4] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, et al. "CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems". In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2009, pp. 215–228. ISBN: 978-1-931971-68-3. URL: `http://www.usenix.org/event/usenix09/tech/full_papers/lohmann/lohmann.pdf`.


# Referenzen (Cont'd)


[5] Daniel Lohmann, Olaf Spinczyk, Wanja Hofer, et al. "The Aspect-Aware Design and Implementation of the CiAO Operating-System Family". In: *Transactions on AOSD IX*. Ed. by Gary T. Leavens and Shigeru Chiba. Lecture Notes in Computer Science 7271. Springer-Verlag, 2012, pp. 168–215. DOI: 10.1007/978-3-642-35551-6_5.

[6] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "Lean and Efficient System Software Product Lines: Where Aspects Beat Objects". In: *Transactions on AOSD II*. Ed. by Awais Rashid and Mehmet Aksit. Lecture Notes in Computer Science 4242. Springer-Verlag, 2006, pp. 227–255. DOI: 10.1007/11922827_8.

[7] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17. OSEK/VDX Group, Feb. 2005.

[8] Abraham Silberschatz, Greg Gagne, and Peter Bear Galvin. *Operating System Concepts*. Seventh. John Wiley & Sons, Inc., 2005. ISBN: 0-471-69466-5.

[9] Matthias Urban, Daniel Lohmann, and Olaf Spinczyk. "The aspect-oriented design of the PUMA C/C++ parser framework". In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10)*. (Rennes and Saint-Malo, France). New York, NY, USA: ACM Press, 2010, pp. 217–221. ISBN: 978-1-60558-958-9. DOI: 10.1145/1739230.1739256.