# An Inheritance System for Structural & Behavioral Reuse in Component-based Software Programming

**Petr Špaček**
Christophe Dony, Chouki Tibermacine and Luc Fabresse

Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier
LIRMM

cnrs advancing the frontiers

UNIVERSITÉ MONTPELLIER 2 SCIENCES ET TECHNIQUES

LIRMM, CNRS and University of Montpellier 2
petr.spacek@lirmm.fr

26 of September, 2012

# OUTLINE

1. Motivation
   - Domain
   - Reuse
   - Problems
   - Approach
2. COMPO
   - Features
   - Component & Descriptors
   - Communication
3. Inheritance system
   - Provisions and services
   - Requirements and internal architectures
4. Conclusion
5. Questions

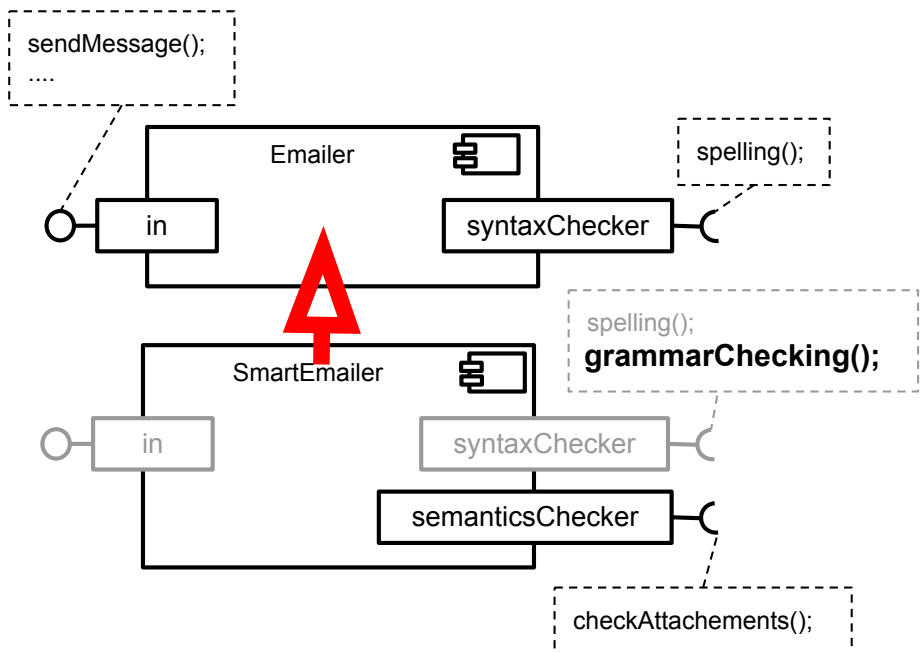
# MOTIVATION - DOMAIN

we are working in the programming languages design domain
and
we use component concept to organize program text

# MOTIVATION - REUSE

- **reuse** = ability to make a new program text on the base of an existing program text
- **differential description in components** = design a new component by stating how it differs from an existing one

```
1  component descriptor Emailer
2  {
3   provides {
4    in : { sendMessage(); ... }
5   }
6   requires {
7    syntaxChecker : { spelling(); }
8   }
9   ...
10 }
11
12 component descriptor SmartEmailer
13   extends Emailer
14 {
15  requires {
16   syntaxChecker : { grammarChecking(); }
17   semanticsChecker : {checkAttachements();}
18  }
19  ...
20 }
```



# MOTIVATION - DIFFERENTIAL DESCRIPTION

Differential description exists in other component models

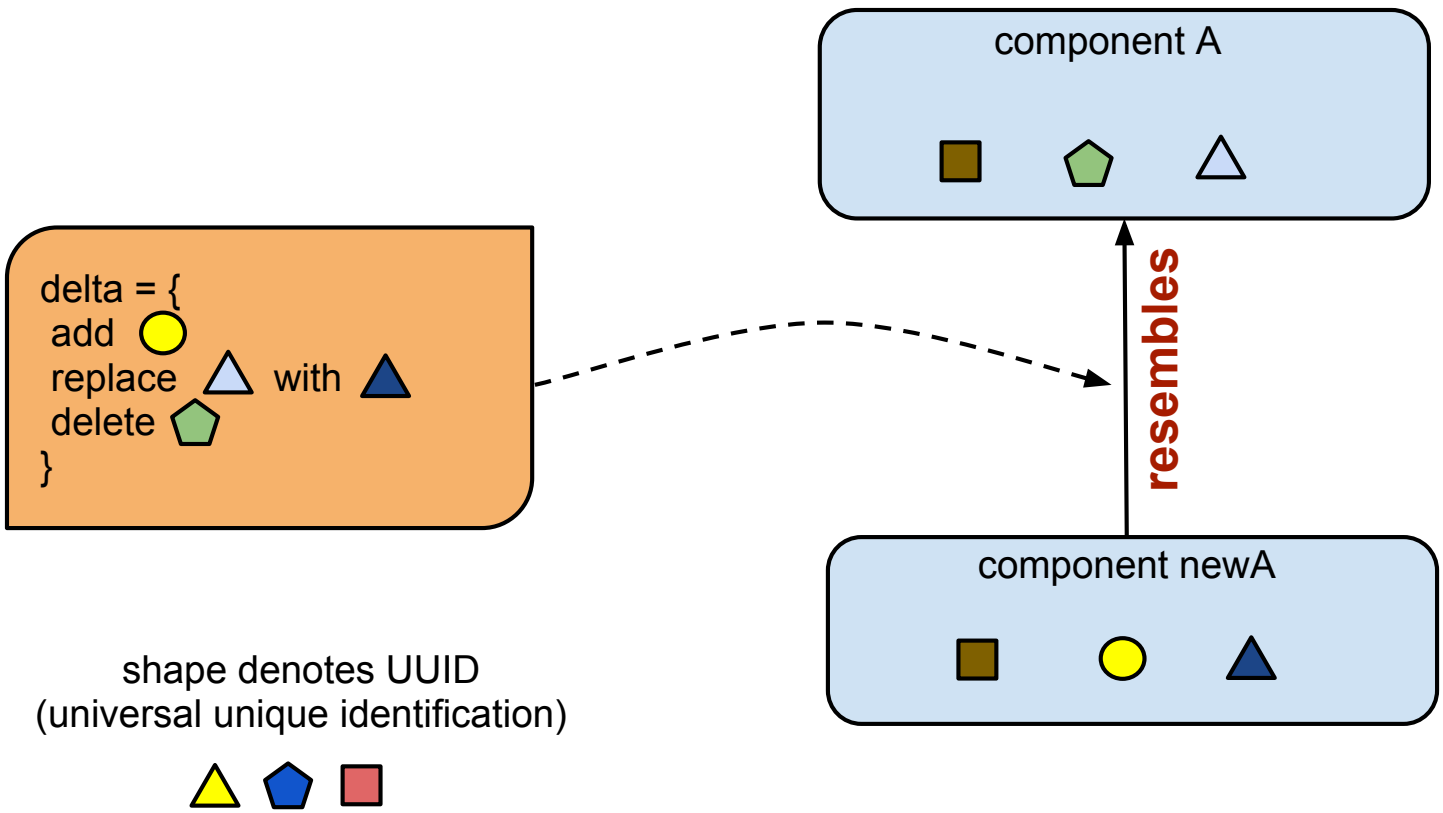
- Backbone (a Darwin-like ADL) introduce **resemblance** - an ad-hoc differential description
- in SOFA one can say *a frame* **inherits** from another frame
- Fractal ADL enables *a definition* being an **extension** of an existing definition
- *component classes* in ArchJava may **inherit** fields, methods, ports and connections
- ....

but there are limitations

# MOTIVATION - REUSE - EXAMPLES 1/2

## Backbone (a Darwin-like ADL)

define new components as a delta from the structure of one or more existing components (i.e. reuse)



# MOTIVATION - REUSE - EXAMPLES 2/2

## SOFA component model

```
1  frame ScannerType {
2    provides: ITokenStream tokens;
3    requires: ICharacterStream source;
4  }
5  frame RomanScannerType inherits ScannerType {
6    requires: IValidator romanValidator;
7  }
8  architecture Compiler implements CompilerType
9    inherits AbstractCompiler implementing CompilerType
10 { ... }
```

## FRACTAL component model and its ADL

```
1 <definition name="Compiler.ScannerType">
2   <interface role="server" signature="Compiler.ITokenStream" name="tokens" />
3   <interface role="client" signature="Compiler.ICharacterStream" name="source" />
4 </definition>
5 <definition name="Compiler.RomanScannerType" extends="Compiler.ScannerType">
6   <interface role="client" signature="Compiler.IValidator" name="romanValidator" />
7 </definition>
```

# MOTIVATION - PROBLEMS

these models have various limitations:

- limited extension and specialization of component parts
- no behavior reuse in ADLs, they focus on the structural part
- limited child-parent substitutions

# MOTIVATION - APPROACH

we want to explore another way

- way where both structure and behavior of a component can be reused
- way where declarations of component parts are inherited and can be extended or specialized
  - i.e. adding or modification of services, ports, connections, internal components
- way which favor modeling power
  - no limitations considering modeling and substitutions

# COMPO- FEATURES

We present our inheritance system in a CBPL named COMPO.

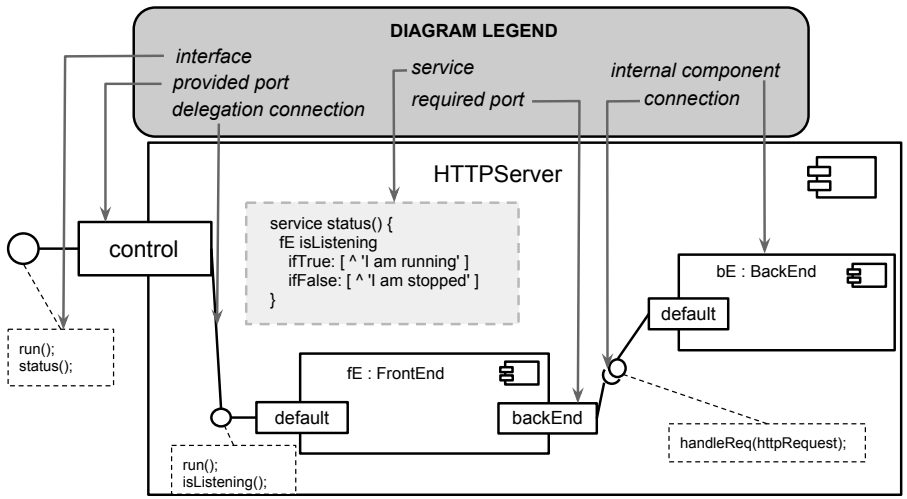
Features

- design and program components and component-based applications
  - no design-implementation projection needed
  - declare and define component services
- clear descriptor-instance dichotomy
- explicit requirements and architectures
- communication uniquely through ports

# COMPO- COMPONENTS & DESCRIPTORS

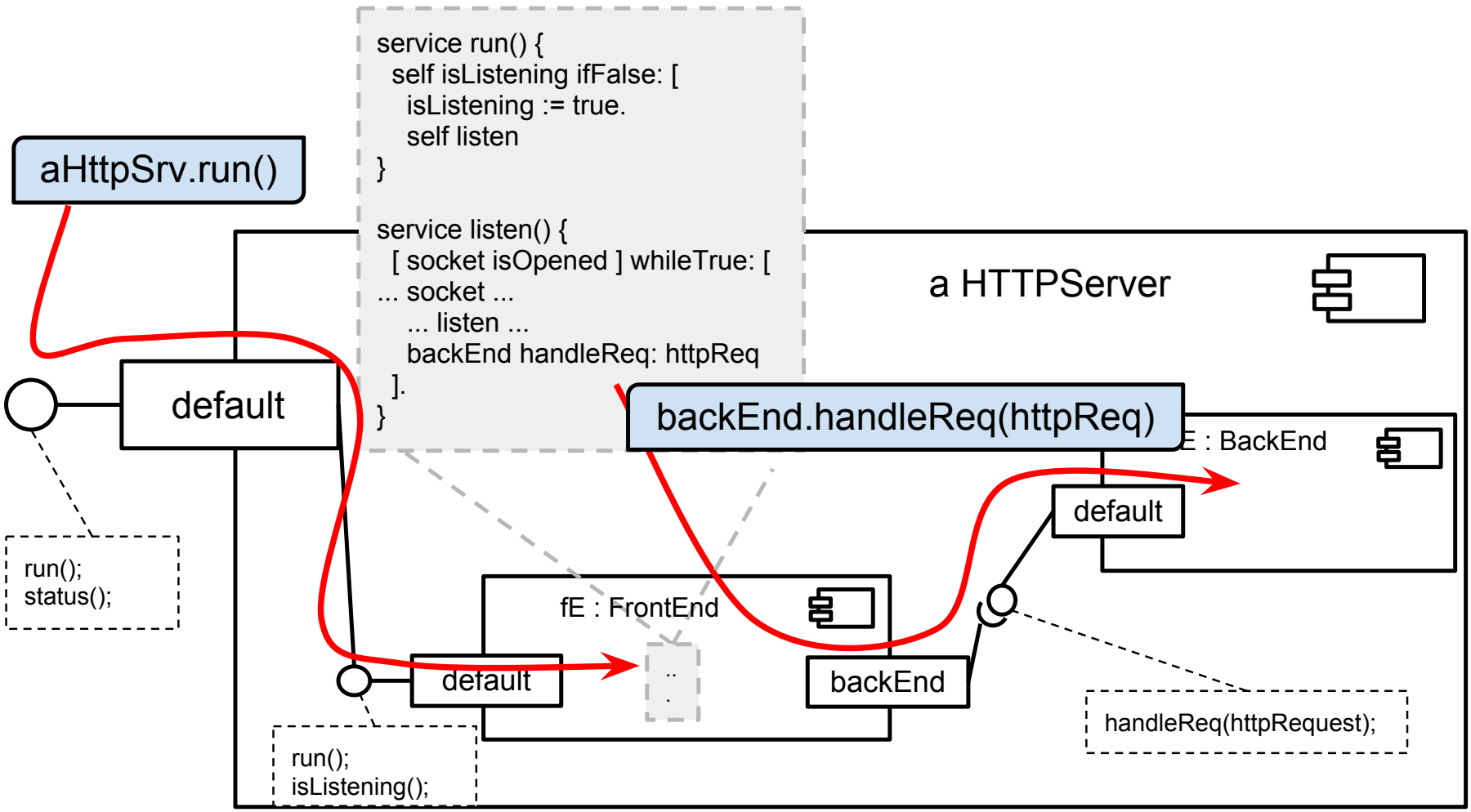
## components are instances of descriptors

```
1  component descriptor HTTPServer {
2    provides {
3      control : { run(); status() }
4    }
5    internally requires {
6      fE : FrontEnd;
7      bE <: BackEnd;
8      name <: String;
9      }
10   architecture {
11     connect fE to (FrontEnd new).default;
12     delegate default to fE@.default;
13     connect fE@.backEnd to bE@.default;
14   }
15   service status() {
16     fE isListening
17       ifTrue: [ ... ]
18       ifFalse: [ ... ].
19   }
20 }
```




ports, internal components, connections, services

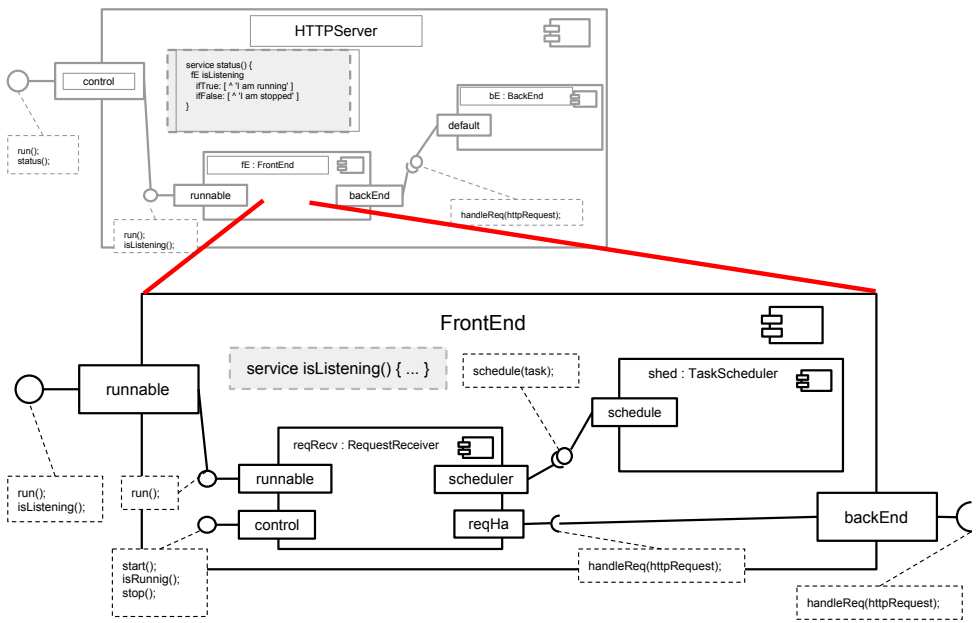
# COMPO- COMMUNICATION




service invocations, delegation & regular connections

# INHERITANCE SYSTEM - PROVISIONS & SERVICES 1/2

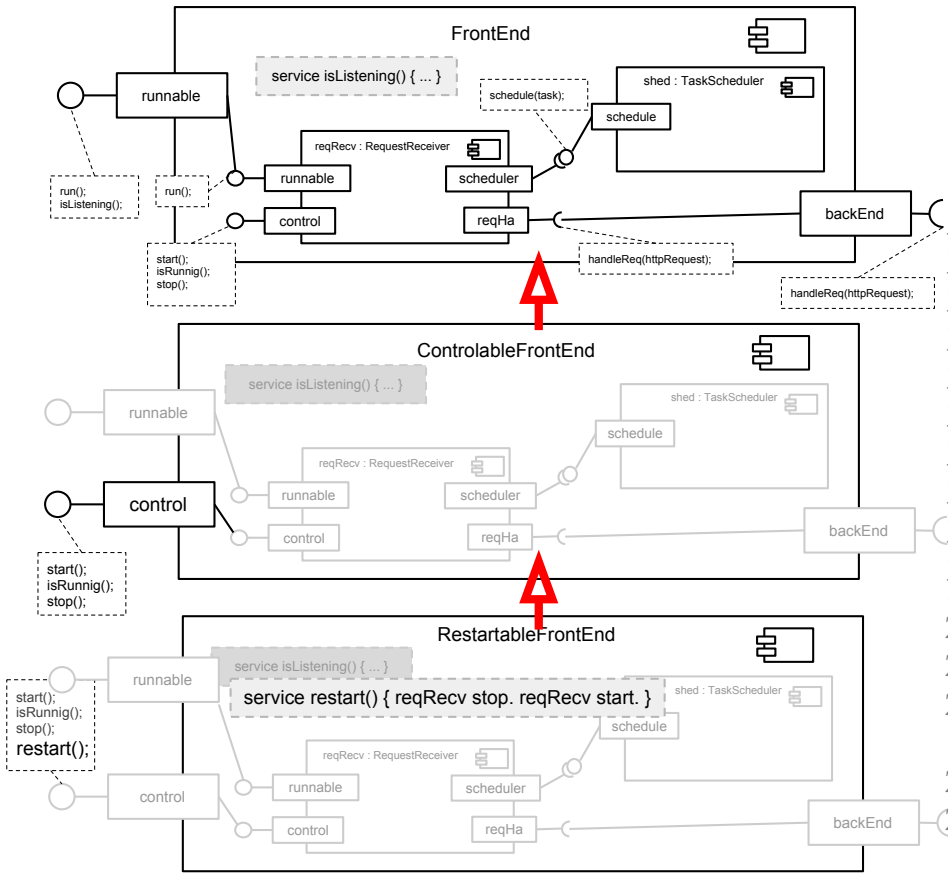
```
1  component descriptor FrontEnd {
2    provides {
3      default : { run(); isListening(); }
4    }
5    requires {
6      backEnd : { handleRequest(r) }
7    }
8    internally requires {
9      rR <: RequestReceiver;
10     s : TaskScheduler;
11   }
12   architecture {
13     delegate default to rR@.default;
14     connect s to (TaskScheduler new).
           default;
15     connect rR@.scheduler to s@.schedule;
16     delegate rR@.handler to backEnd;
17   }
18   service isListening() {
19       ^ rR isRunning.
20     }
21 }
```




what if we would like to export the control services of **reqRecv** internal component ?

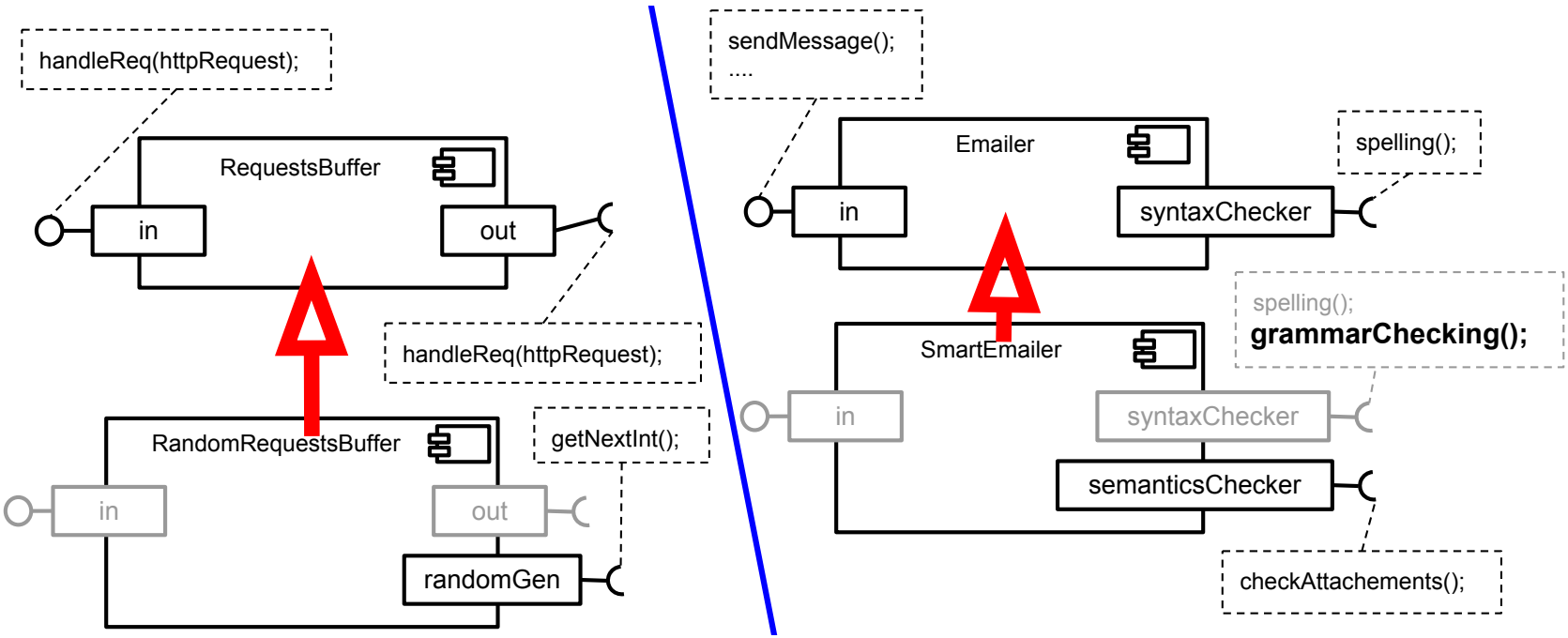
# INHERITANCE SYSTEM - PROVISIONS & SERVICES 2/2

In our inheritance system we can add or specialize provided ports and services




```
1  component descriptor ControlableFrontEnd
2    extends FrontEnd
3  {
4    provides {
5      control : {start(); isRunning(); stop()
                 }
6    }
7    architecture {
8      delegate control to rR@.control;
9    }
10 }
11 ------------------------------------------
12 component descriptor RestartableFrontEnd
13   extends ControlableFrontEnd
14 {
15   provides {
16     control : { restart(); }
17   }
18   service restart() {
19     rR stop. rR start
20   }
21   service isListening() {
22     super isListening ifTrue: [^0] ifFalse:
                 [^1]
23   }
24 }
```

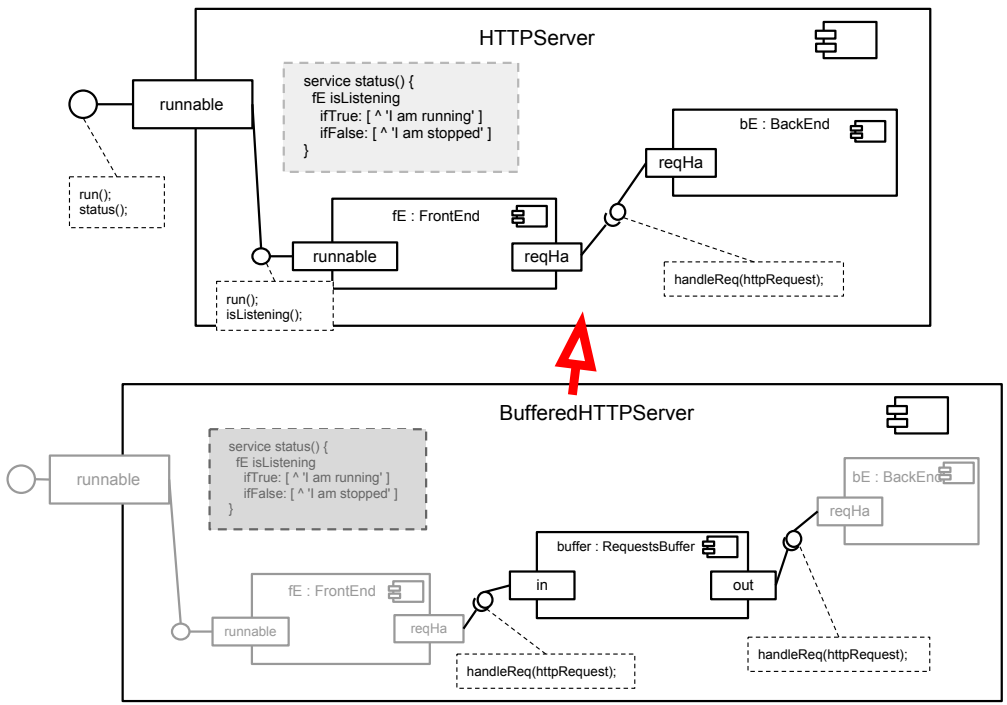
# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 1/6




extension, specialization of required ports

# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 2/6

```
1  component descriptor HTTPServer
2  {
3      ....
4  }
5
6  component descriptor QueuedHTTPServer
7    extends HTTPServer
8  {
9   internally requires {
10   buffer <: RequestsBuffer
11  }
12  architecture {
13   disconnect fE@.backEnd from bE@.default;
14   connect fE@.backEnd to buffer@.in;
15   connect buffer@.out to bE@.default;
16  }
17 }
```

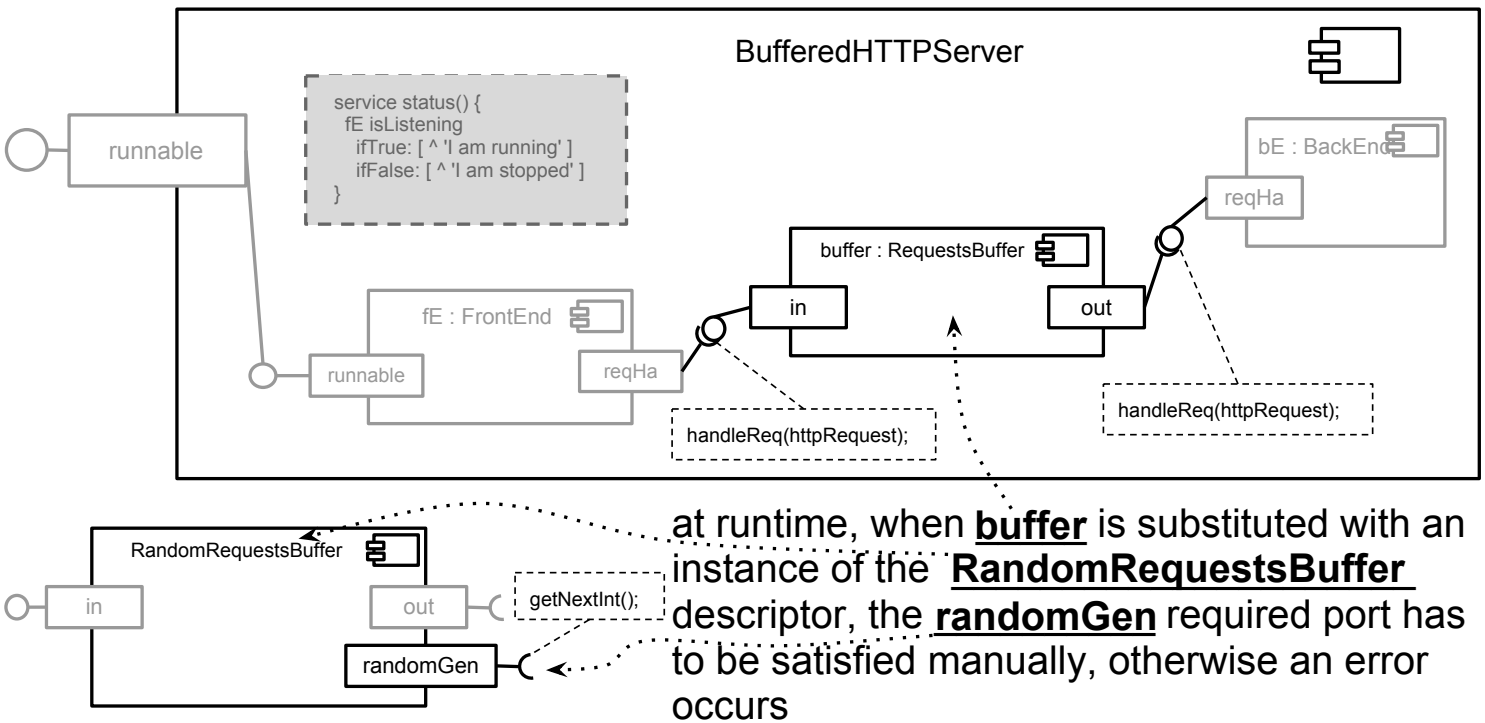



extension, specialization of internal architectures

# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 3/6

a sub-descriptor with additional requirements **raise new question**

- what about child-parent compatibility, in terms of substitutions?



# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 4/6

what about child-parent compatibility, in terms of substitutions?

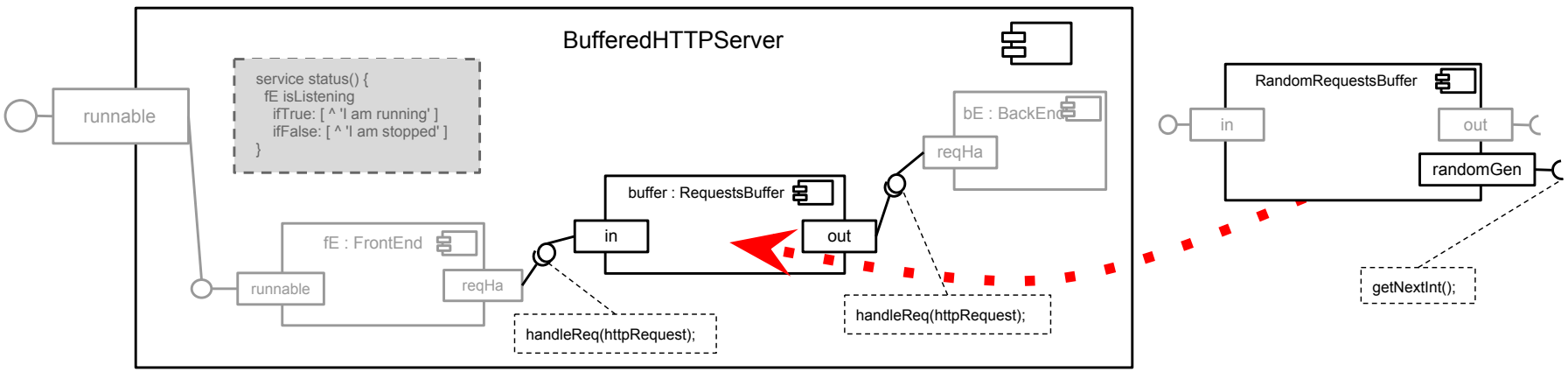
1. forbid declaration of new req. ports in sub-descriptors
   - + preserve child-parent compatibility (provide the same or more, require the same or less)
   - - restricts modeling power
2. forbid substitutions with additional requirements
   - + modeling power
   - - wrong when all additional requirements are satisfied
3. allow substitutions with additional
   - + modeling power
   - + not limited substitutions (core of many framework technologies)
   - - provide support for performing substitution safely

# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 5/6

## support methods for performing substitution safely

- *newCompatible(arrayOfPairs)*
  - used when a sub-descriptor has at least one additional required port
  - returns a component compatible with super-descriptor
  - has a unique parameter, an array of pairs *port-component*
  - additional requirements are satisfied by connections to components given in the array argument
- *isCompatibleWith* for substitutions validity checking
- *reconnect(portName,newComponent)*
  - has two arguments: (1) the name of an internal required port referencing the component which should be replaced; (2) the replacing component
  - works in two phases
  - first, it checks for compatibility between the original and the new component descriptor
  - second, if everything is ok, it performs the replacement and does all necessary reconnect actions

# INHERITANCE SYSTEM - REQUIREMENTS AND INTERNAL ARCHITECTURES 6/6




```
1  component descriptor Test {
2   internally requires {
3    server <: BufferedHTTPServer;
4   }
5   service testOne() {
6    server reconnect: 'buffer'
7           to: (RandomRequestsBuffer newCompatible:
8                 Array with: (Pair key: 'randomGen' value: (Randomize new))
9                 ).
10  }
11 }
```

# CONCLUSION - SUMMARY

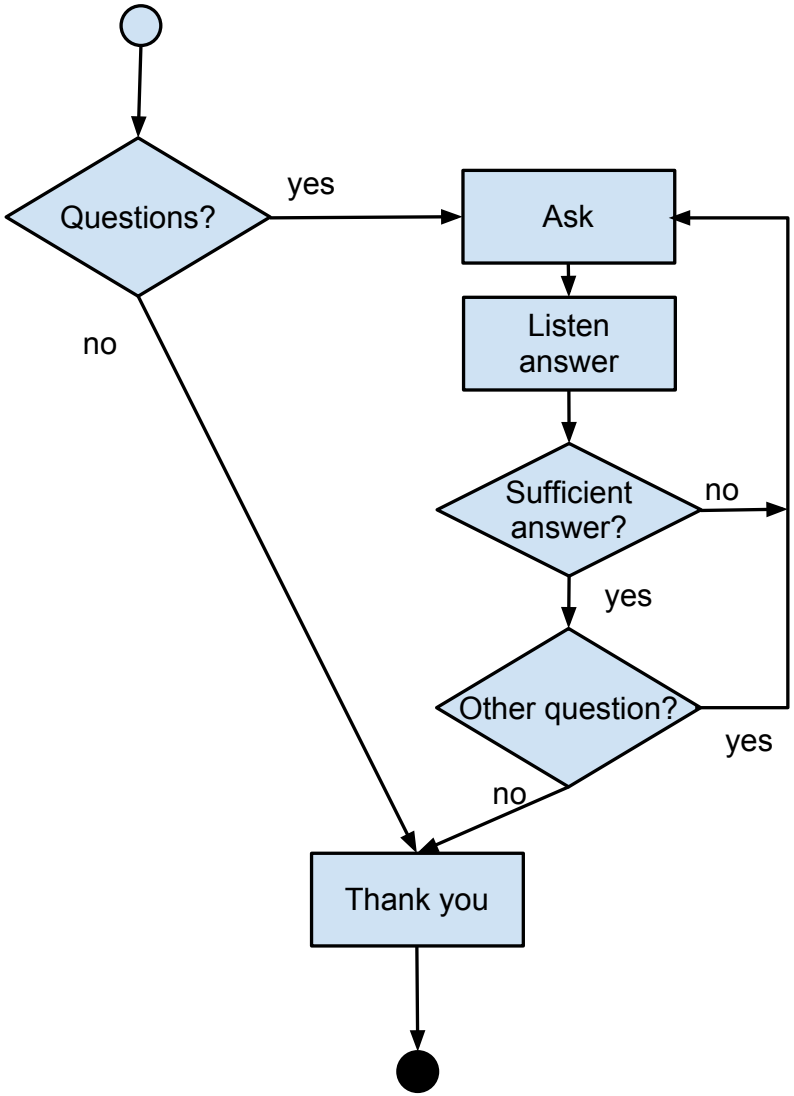
- inheritance in CBPL is a fully-fledged mechanism which lets us reuse behavior and structure of component descriptors
- proposal highlights
  - it takes into account in the same context the architecture modeling and coding aspects of components
  - adding and specialization of all parts of descriptors
  - covariant specializations support - to promote expressive power and reuse
- current version of Compo can be downloaded here:

  http://www2.lirmm.fr/~spacek/compo

# CONCLUSION - PERSPECTIVES

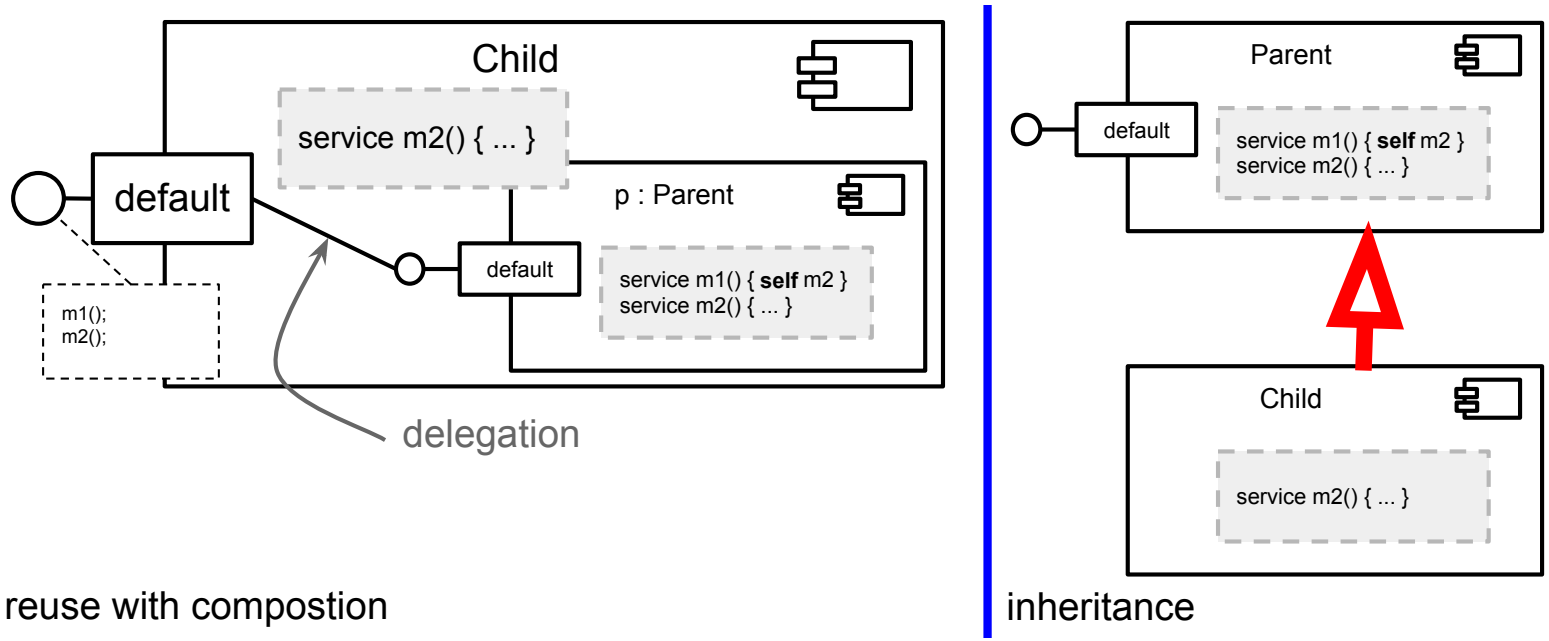
- reflective version of COMPO, based on meta-model
- design and write hierarchies of architecture constraints in COMPO
- implement model transformations of COMPO application in COMPO

# CONCLUSION - THE END



# QUESTIONS - COMPOSITION 1/2

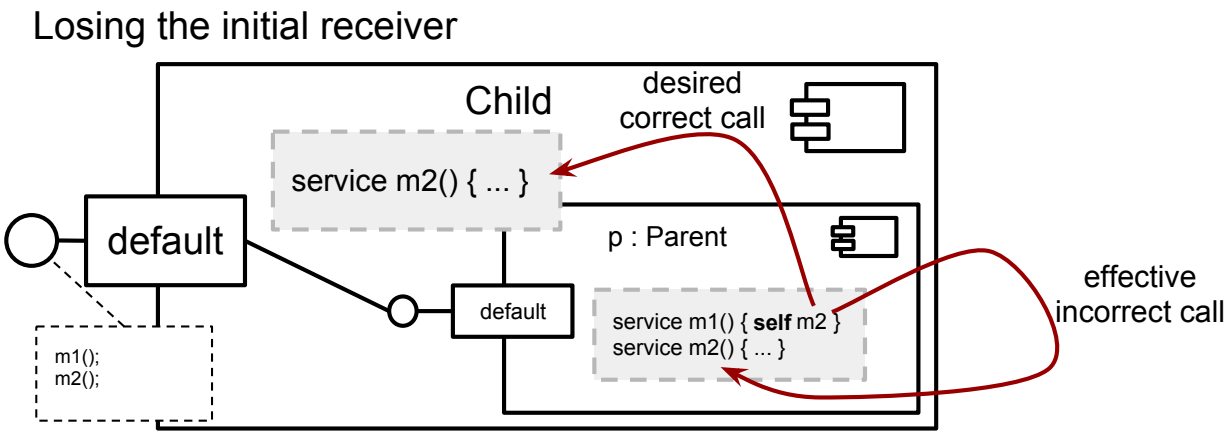
Why not use composition only?

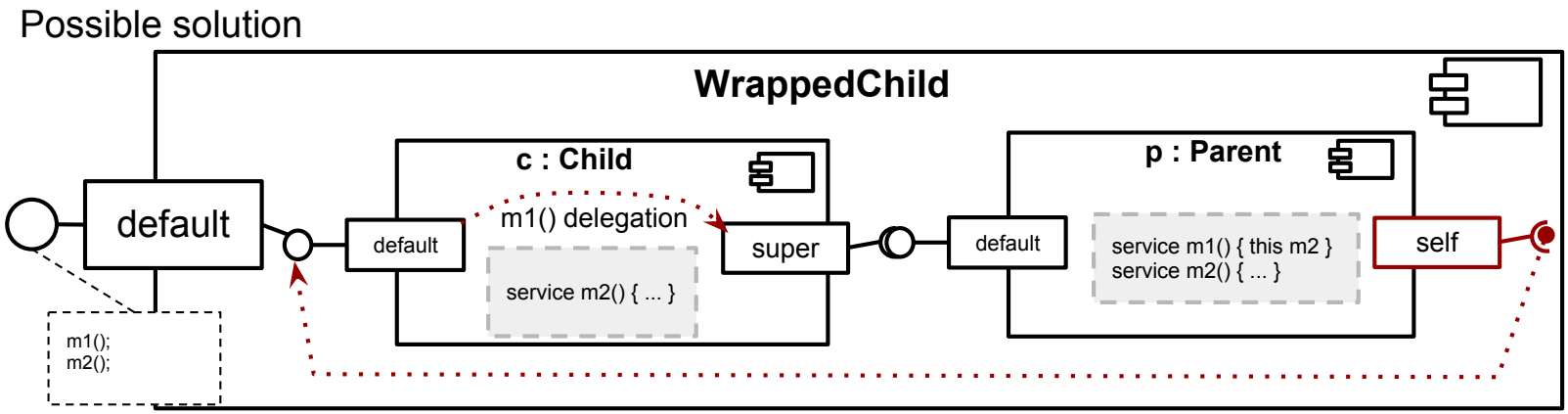



if we execute `(new Child()).m1()`, whose `m2()` will be executed?

# QUESTIONS - COMPOSITION 2/3

- what about receiver environment i.e. `self` (`this` in Java, ...)?




- how to reuse internal architecture design?

# QUESTIONS - COMPOSITION 3/3

Conclusion

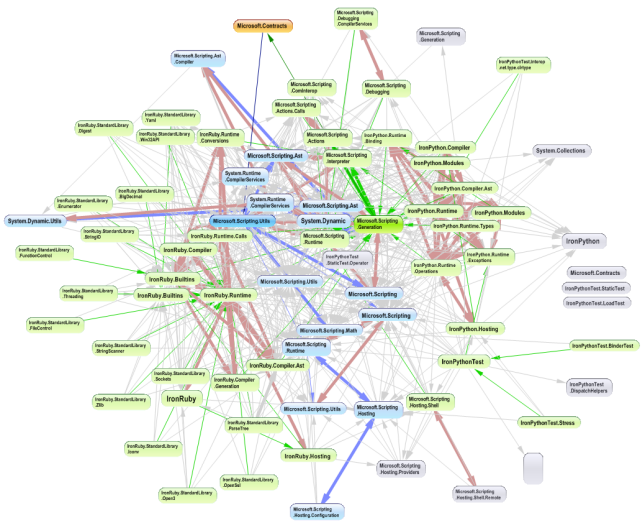
- it is not possible to reuse internal architecture design
- implementation of services specialization is complicated
- clumsy when modeling hierarchies or for conceptual classification

# QUESTIONS - OUR VISION OF COMPONENT

How do we see components?

components beyond objects

- explicit requirements - object dependencies are spread out
- explicit architecture - scattered internal composition in object



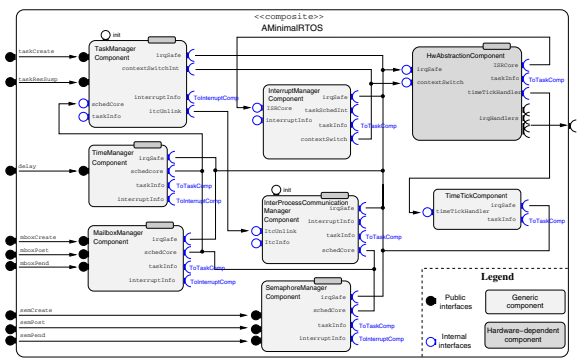



**Fig. 4.** Architecture's Example of a $\mu$C RTOS