

Software Composition

M.R.V. Chaudron

www.win.tue.nl/~mchaudro

Dept. of Mathematics and Computing Science
Eindhoven University of Technology

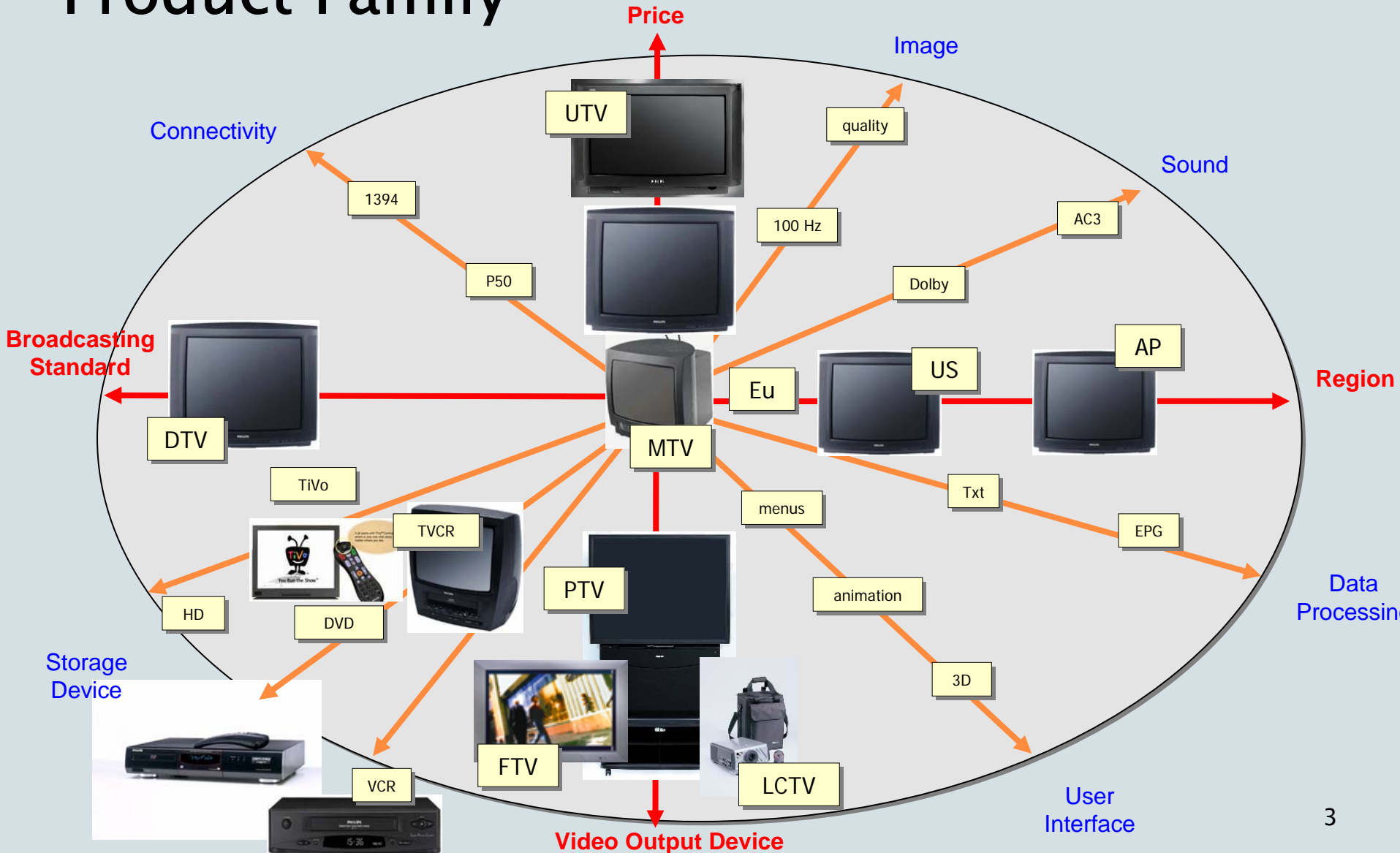
With contributions from Prof. Ivica Crnkovic, Malardalen, Sweden

Agenda

Composition is Pervasive in Software

- Bit on industrial component model & experience
- Composition in different programming languages
- Composition of components in distributed systems
 - CORBA: request–response middleware
- Predictable Assembly
 - Predicting properties of assemblies of components
- Composition of Systems

Product Family



Construct by composition

TV + VCR = TVCR



TV + DVD = TV-DVD



TV + HD = Tivo



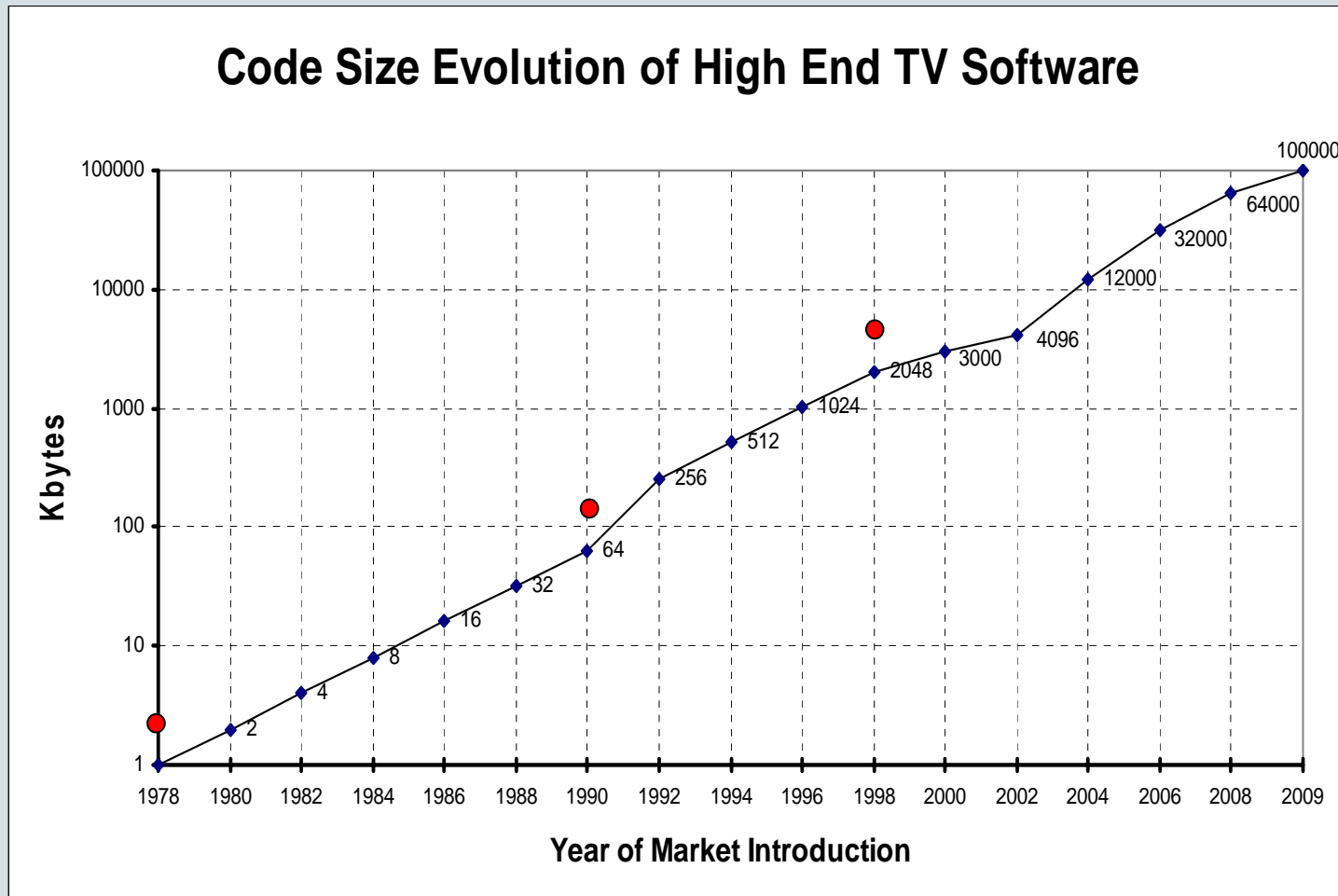
TV + STB = Digital TV



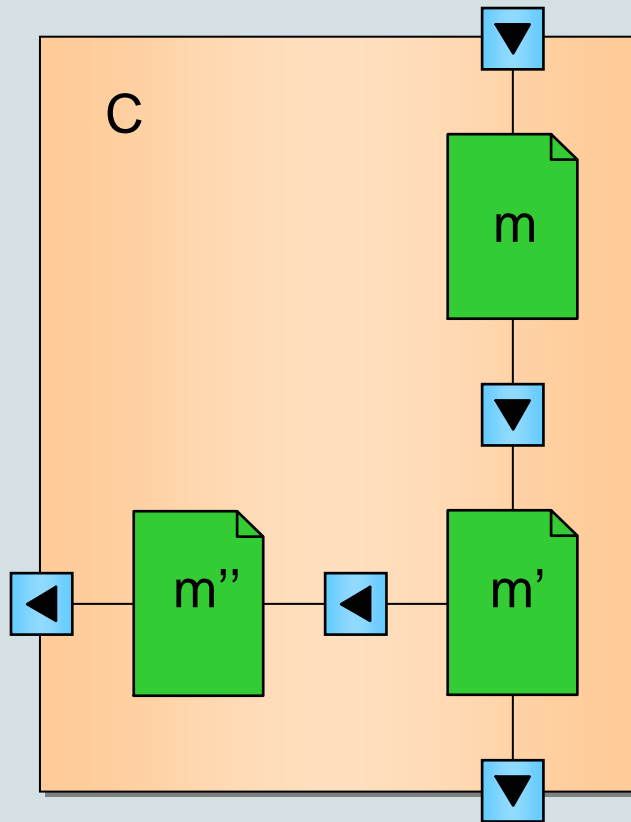
TV + Audio = Home Theater



(1) Complexity



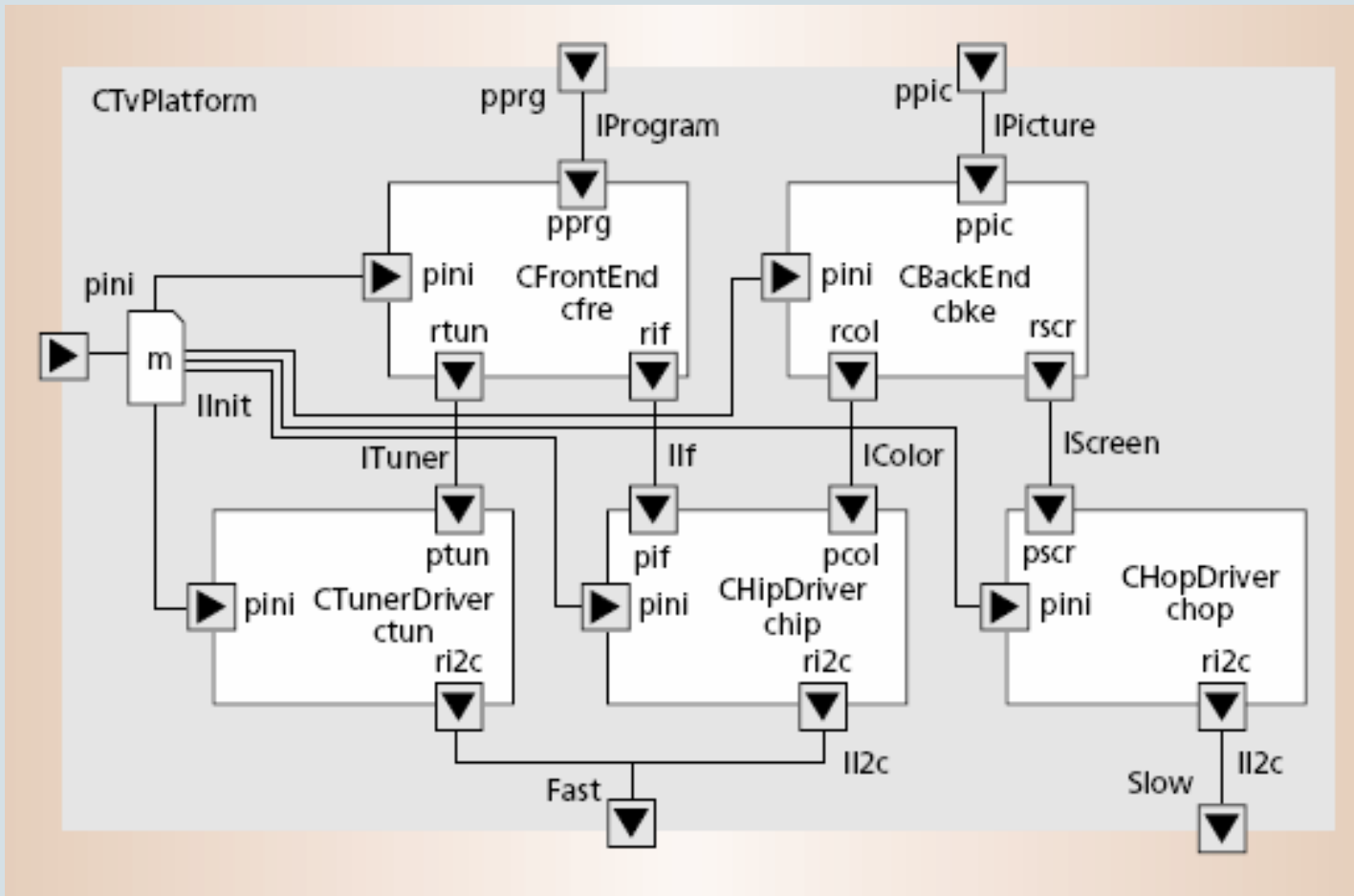
Component



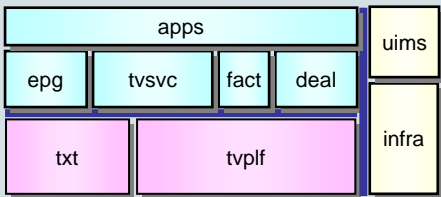
A Koala component is:

- a **reusable** ...
- ... **unit of encapsulation** ...
- ... still with **variation points**
- providing stuff

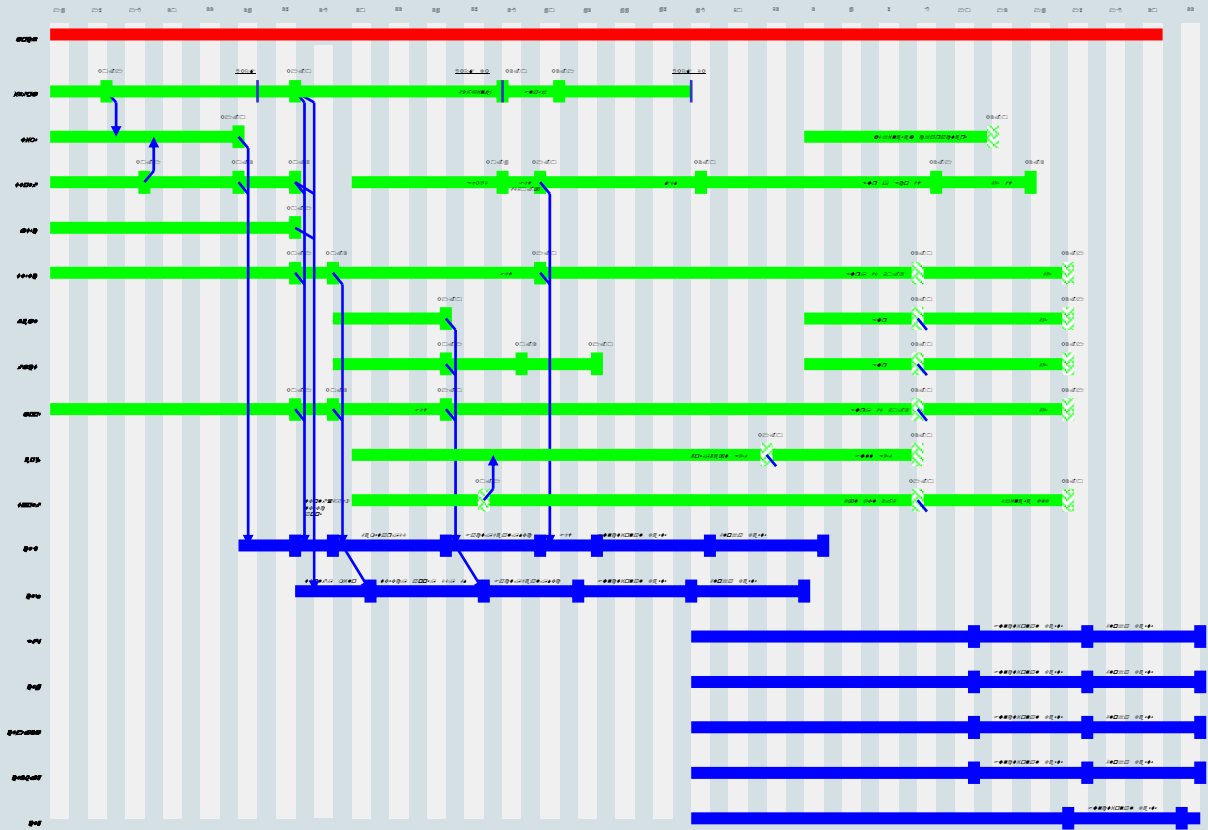
Koala - Example



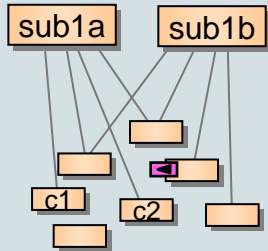
Development Processes



Architecture Project



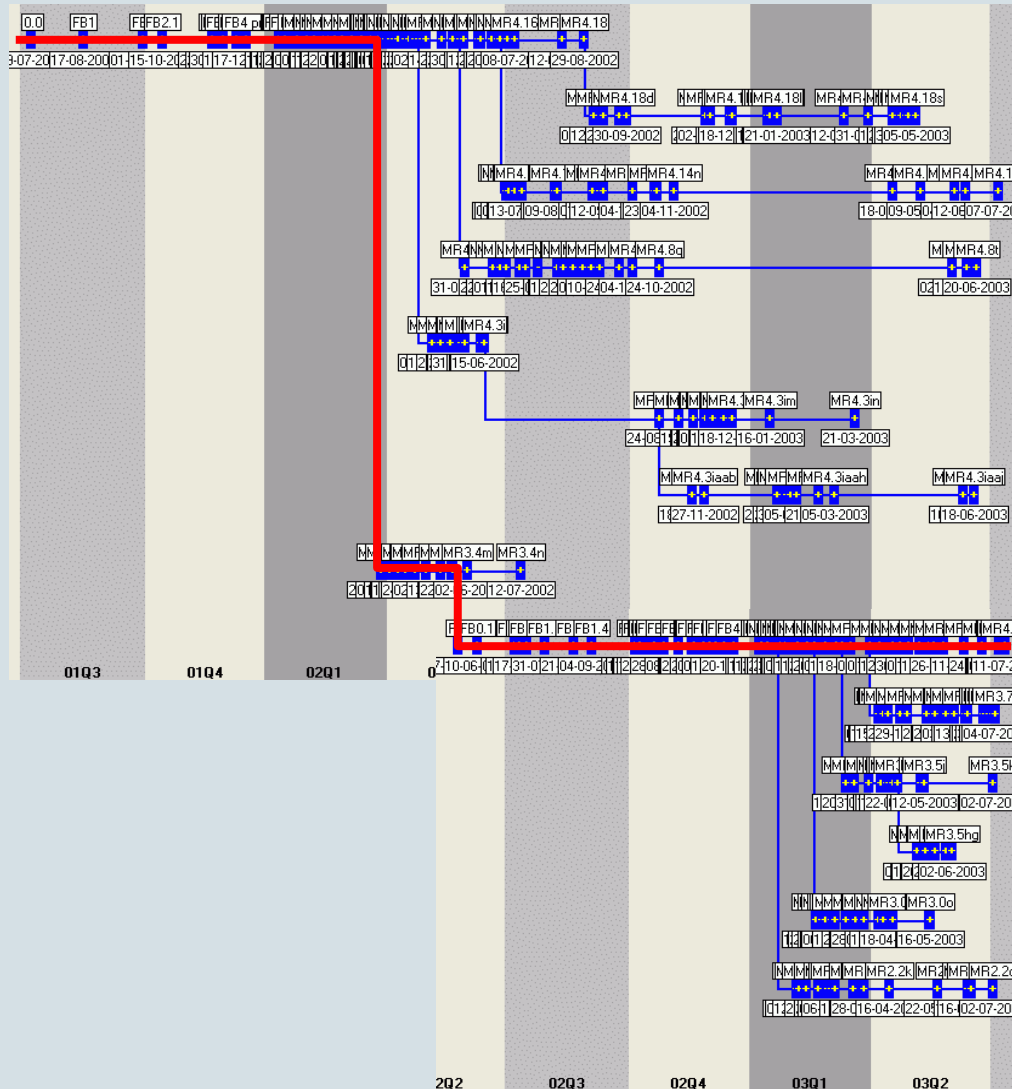
Subsystem Evolution Projects



Product Realization Projects



Branching in practice...



There are a large number (>10) of long-living (> 1y) branches.

Advantage:
Safeguard products from 'improvements' in other products.

Disadvantage:
Multiple maintenance.

Documentation: Interface Data Sheet

The family

MG-R Software Components

INTERFACE DATA SHEET

Long name, short name and description

<Interface> (<shortname>)

<Short description of the interface>

<status> Specification

Approved by: **Status**

March 15, 1999

Version: <version> (15)

<author>

Date, version, build, author

Philips Consumer Electronics

Philips Consumer Electronics <Interface>

Interface <Interface> (<short>)

The interface <Interface> ...

Interface <Interface> (<short>)

Very short description

Concepts

Types

Constants

Functions

Interface <InterfaceNotify>

Table of contents

Notification Functions

Behaviour

Application Notes

Remarks

Concepts Things you have to read first...

Types

Constants

Functions Notification interface in same data sheet!

Interface <InterfaceNotify>

Notification Functions

Behaviour State transition diagrams or sequence diagrams

Application Notes

Remarks Hints on how to use the interface

ACXAC 0, 0000 <name> - Specification Page 2 of 1

Documentation: Component Data Sheet

The family

MG-R Software Components

COMPONENT DATA SHEET

Long name, short name and description

<Component> (<shortname>)

one line description

Draft Specification

Approved by: **Status**

March 15, 1999

Version: <version> (13)

<author>

Date, version, build, author

Philips Consumer Electronics

Philips Consumer Electronics - Component -

Component <Component>

The component <Component> provides facilities for ... The component has the following pin-out:

External diagram

Functionality **Bullet list of selling points...**

Concepts **Things you have to read first...**

Interfaces **Provides and requires**

Diversity

Execution & Synchronization

Resources

Application Notes **Typical usage**

Remarks

No implementation notes!!!

December 1, 1998

Draft Specification

Page 2 of 1

Composition

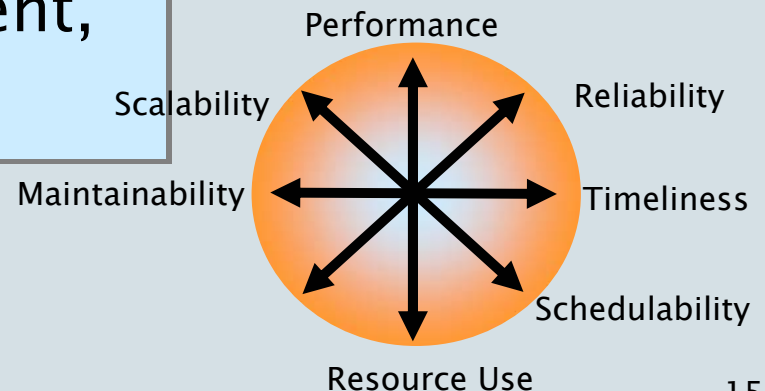
Substitution versus Composition

| Substitution | Composition |
|---|--|
| Replace one ' <i>piece of software</i> ' with another that has the same observable properties – in particular the same interface. | Compose ' <i>pieces of software</i> ' such that they may interact. |
| This is 'do-able' and often sufficient for reuse of software. | |
| Context is static, Semantic context is known | Context may be dynamic, Semantic context is not known |
| Typically done before compilation | Typically done after compilation: run-time |

What should be the aim of composing software ?

- Interoperation?
- Functional composition?
- Synchronisation?
- ... Suggestions from the audience ...

Composition of functions & features
under invariance, or improvement,
of quality properties



What is the desired unit of composition?

- What units make sensible software building blocks?
- Keep in mind issues related to components
 - State(less)
 - Hierarchy
 - Role of the infrastructure

What do you want to compose?

Aspect of Scale / Granularity:

- Instructions
 - Methods
 - Modules / Classes
 - Frameworks / Libraries
 - Services
 - Components / Aspects
 - Systems
- } intra-language
- } intra-architecture
- } across systems

Composition: the programming language level

- Functional Programming Languages
- Logic Programming Languages
- Imperative Programming Languages
- Object Oriented Programming Languages

Composition in Functional Prog. Lang's

Functional Programming: Haskell, ML, LISP, ...

Philosophy:

Software is for computing functions.

The composition of functions is again a function.

- $(f \circ g) x$ or $f(g(x))$

-The output of function g is input to function f

-Requires compatible types

• 'o' is of type $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$

-Has nice algebraic properties (distr, assoc, comm, idem)

-Function is also unit of abstraction/structuring

-Has elegant semantics

• for the functionality, not for the behaviour!

• not good at handling state

Composition in Logical Prog. Lang's

Logical Programming: Prolog

Philosophy: Computing = Theorem Proving

A program is a set of deduction rules (in predicate logic)

Composition is the union of deduction rules.

- $B(x) \Leftarrow A(x) \wedge C(y) \Leftarrow B(y)$
- The result of ded-rules are the facts that can be derived from them
- Semantics is elegant for small programs only
 - The interaction of deduction rules becomes incomprehensible
 - There is no intuitive relation between a program and its behaviour

Composition in Imperative Prog. Lang's

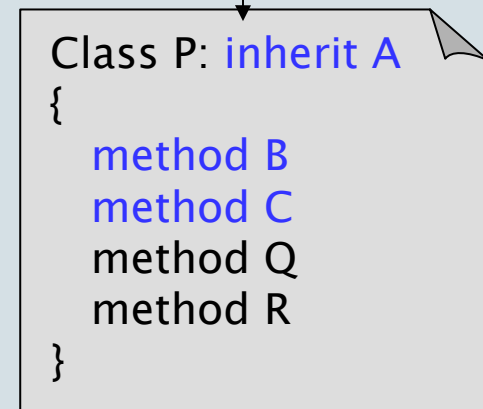
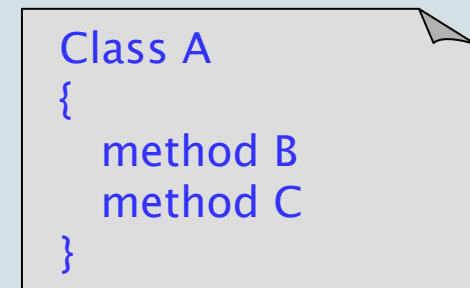
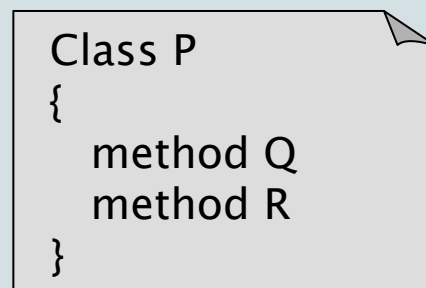
Programming Languages: Fortran, Pascal, C, ADA, ...

Philosophy:

- A program is a sequence of elementary instructions for a processor.
- Composition is the appending/merging of the sequence of instructions.
- Elementary instructions are: assignment & expression evaluation
 - $x := y * y$
 - instructions can be composed using control-flow statements:
; , if .. then .. else, while .. do, repeat ,
- Semantics is elegant for small programs only; reasoning is difficult
 - There is a direct relation between a program and its behaviour
 - This lack of abstraction complicates reasoning about large programs

Composition in OO Prog. Lang's

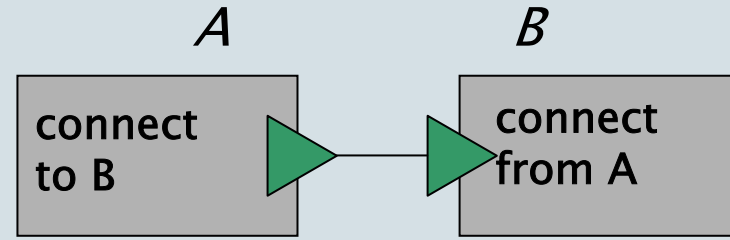
- OO Prog. Lang's: Simula, C++, Java, ...
- Objects are essentially a structuring/abstraction mechanism
- Main composition mechanism:
 - Inheritance
 - Mixin's (Bracha)



How does a system know where to 'return' the result of a function call?

Endogenous versus Exogenous composition

Endogenous composition:
composition is built into the items that are composed

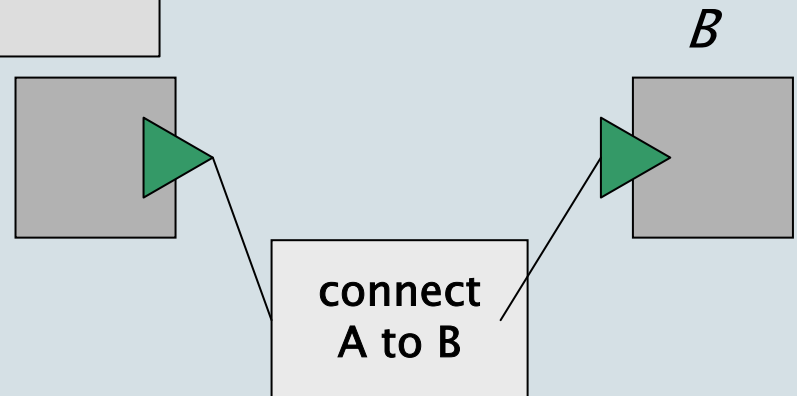


e.g. method calls in OO languages

```
Class A
{
  method Q
  {
    B.m()
  }
}
```

Exogenous composition:
composition is defined outside of the items that are composed.

- localizes changes in binding
- reduces dependencies between application components



Benefits: consider changes in A or B or in the connections.

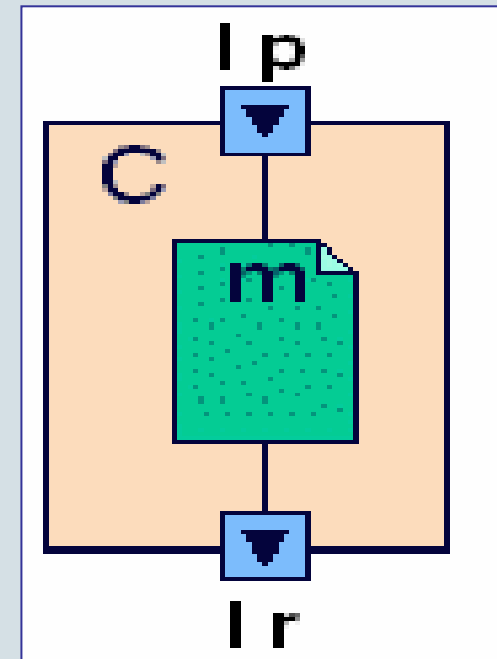
KOALA Example

- File i.id (hand written)

```
interface I{
    void f(int x);
}
```

- File c.cd (hand written)

```
component C {
    provides I p;
    requires I r;
    contains module m;
    connects p = m;
           m = r;
}
```



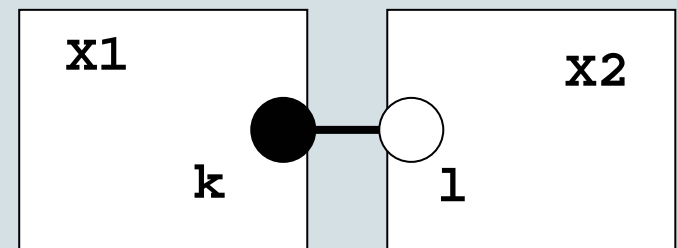
Binding of Components

```
Interface Iv  
{ ... }
```

```
Component P1  
{ provide Iv k }
```

```
Component P2  
{ require Iv l }
```

```
Inst X1:P1, X2:P2  
Bind X1.k - X2.l
```



Composition: the programming language level

- Technically we have:
 - FP: functions; LP: clauses; OO: classes
 - Issues in composition:
 - Syntax: form of the interface
 - Execution model: control flow, data flow
 - Data model: representation of data, meaning of data
 - Often incomplete information in the program text

Composition: The Component Level

Component Binding

When a component X uses/knows information about another component Y , this is a dependency of X on Y . We say that X is bound to Y .

Composition/Binding time

Binding time: the stage in development that a component X starts using information about another component Y.

Development-time

- Integrate source text files and build instructions

Compile / Link-time

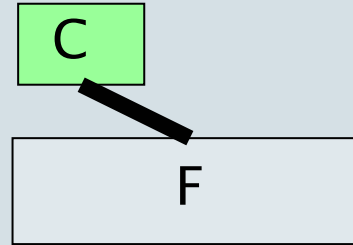
- Integrate source or object-code into executable

Run-time

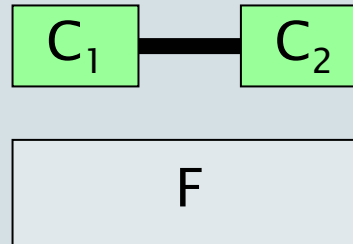
- Integrate executable components in a running system

Composition Forms

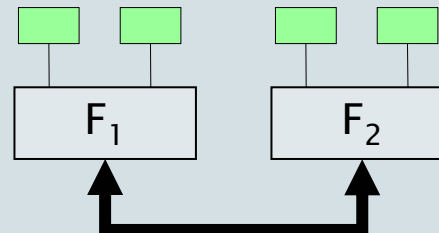
component–framework
‘docking’



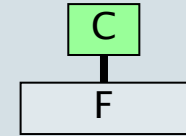
component–component



framework – framework



Component – Framework binding



C–F binding is governed by a deployment contract.

This binding enables components to:

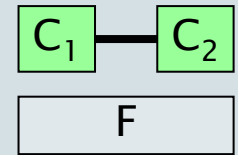
- acquire resources for instantiation and execution
- use the interaction mechanism(s) provided by the framework

This binding enables the framework to provide common facilities to all components; e.g.

- supervise component resource use; such CPU– and memory–use.
- life cycle services (installation, execution, replacement, removal)
- registration & look–up services

The relation between a Framework and a Component seems very similar to that between an OS and an application.

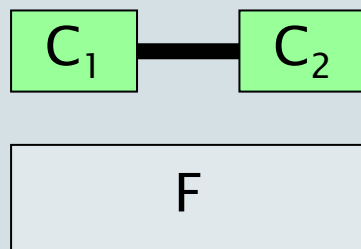
Component – Component Binding



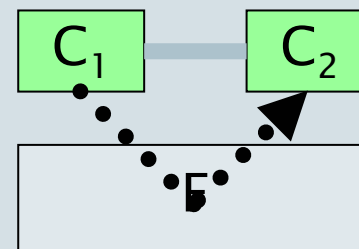
C–C bindings are governed by application contracts.

These binding enable interaction between components and hence the assembly of component services.

The interaction mechanism is provided by the framework.

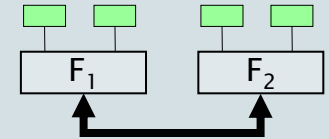


conceptual



actual

Framework–Framework binding



F–F bindings are governed by interoperation–contracts.

This binding enables interaction between components of different, possibly heterogeneous, frameworks.

Typical example is IIOP which connects two ORB's.

Let's look at an Example: OBJECT ORIENTED MIDDLEWARE (CORBA)

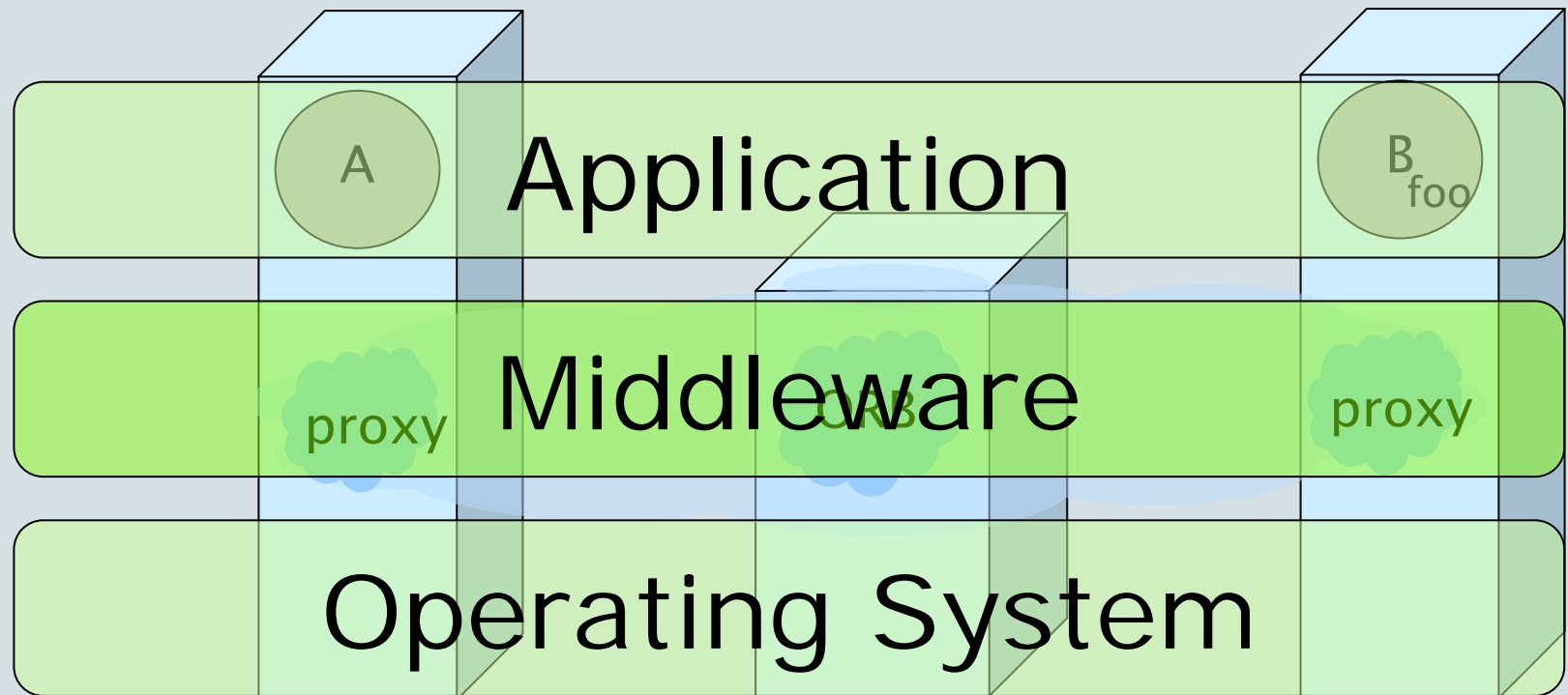
The items to be interconnected are *objects*

Objects interact via method invocation (remote procedure call)

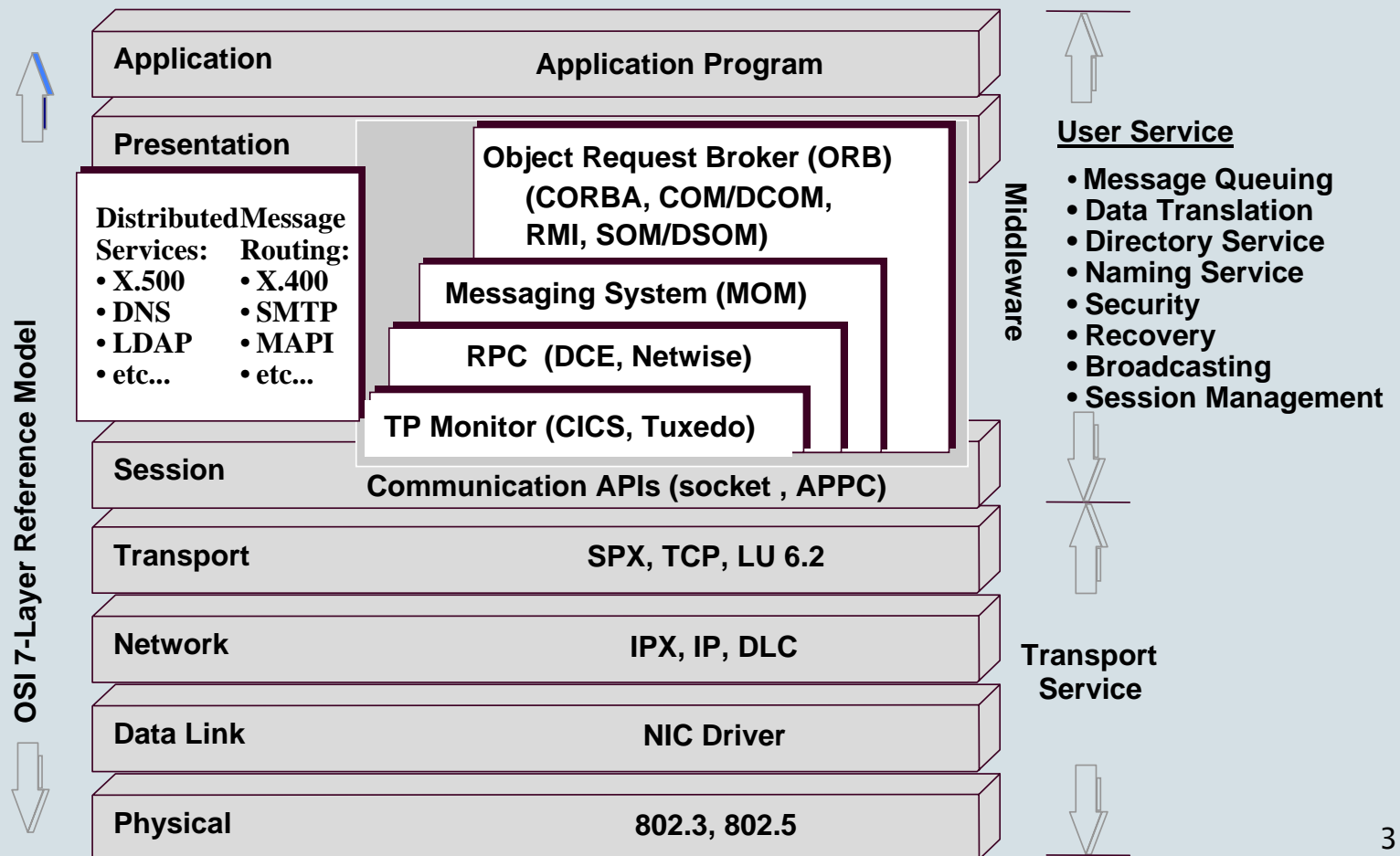
Middleware should make the distribution invisible to the objects



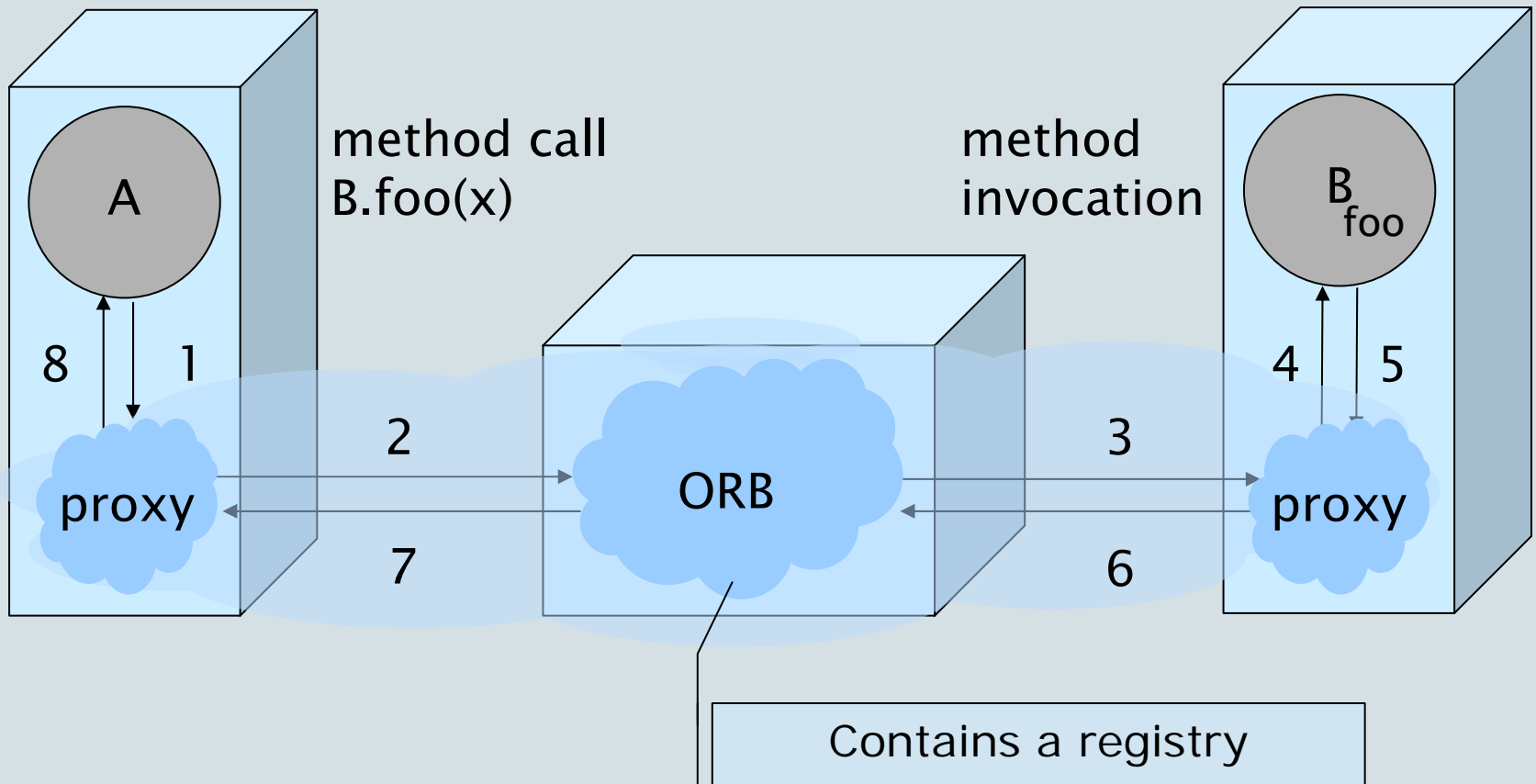
OBJECT ORIENTED MIDDLEWARE



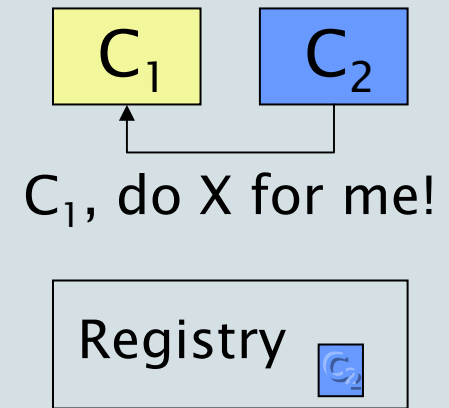
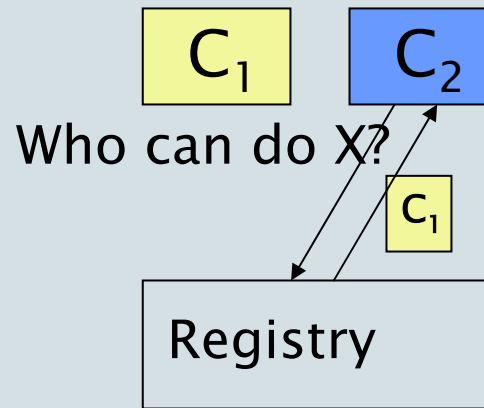
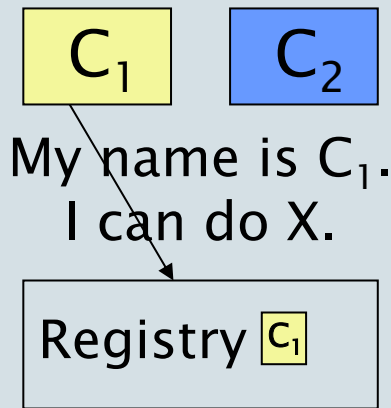
Middleware's Position within the Distributed System Architecture



Remote Procedure Call MIDDLEWARE



Binding CORBA style



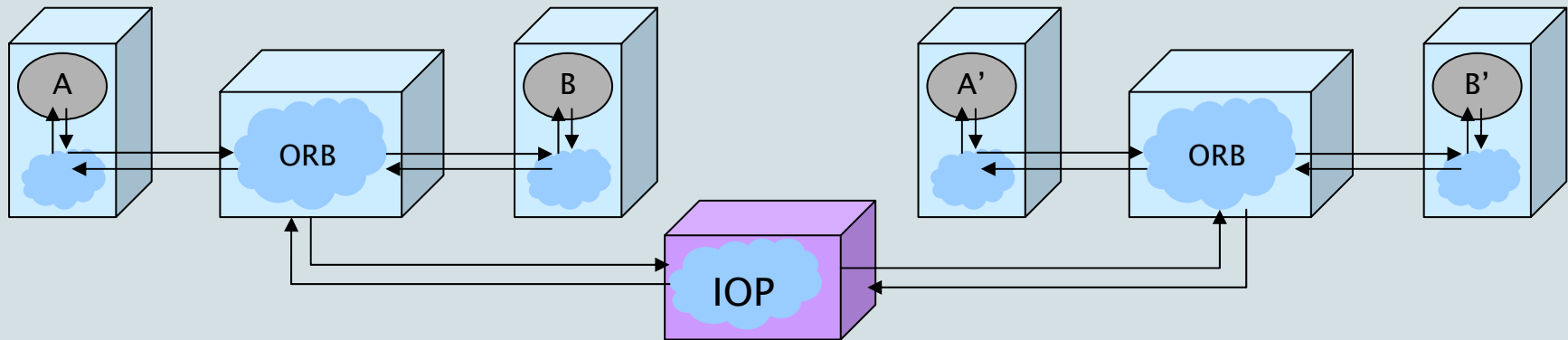
C_2 gets a reference to C_1

Registry can be filled at compile-time or at run-time

CORBA SERVICES

| | |
|---------------------|--|
| Timing | Creation of a global time base |
| Naming | Binding between names and objects |
| Event notif. | Asynchronous message comm. between objects |
| Life-cycle | Creation, deletion, copying and moving of objects |
| Persistency | Persistent storage and management of object state |
| Transaction | Support of multiple concurrent transactions Flat transaction mandatory; nested transactions optional |
| Concurrency Control | Concurrent coordinated access of objects by clients Object locksets with read, write and upgrade locks |
| Relationship | Supports the specification, creation and maintenance of relations between objects for maintenance purposes |
| Externalization | Externalization of objects across processes and ORB's |

INTER ORB PROTOCOLS (IOP)



Standard for ORB-to-ORB interoperability

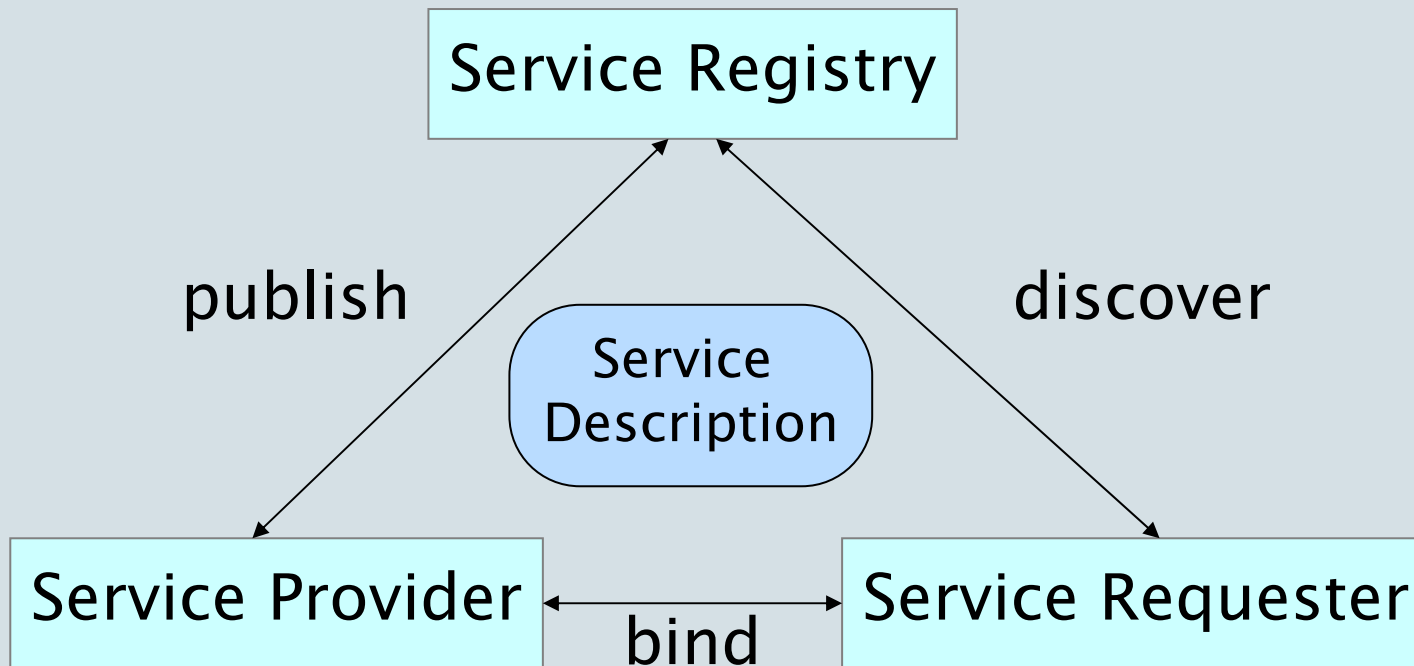
General Inter-ORB Protocol (GIOP)

- specifies transfer syntax
- standard set of message formats for connection-oriented transport

Internet Inter-Orb Protocol (IIOP)

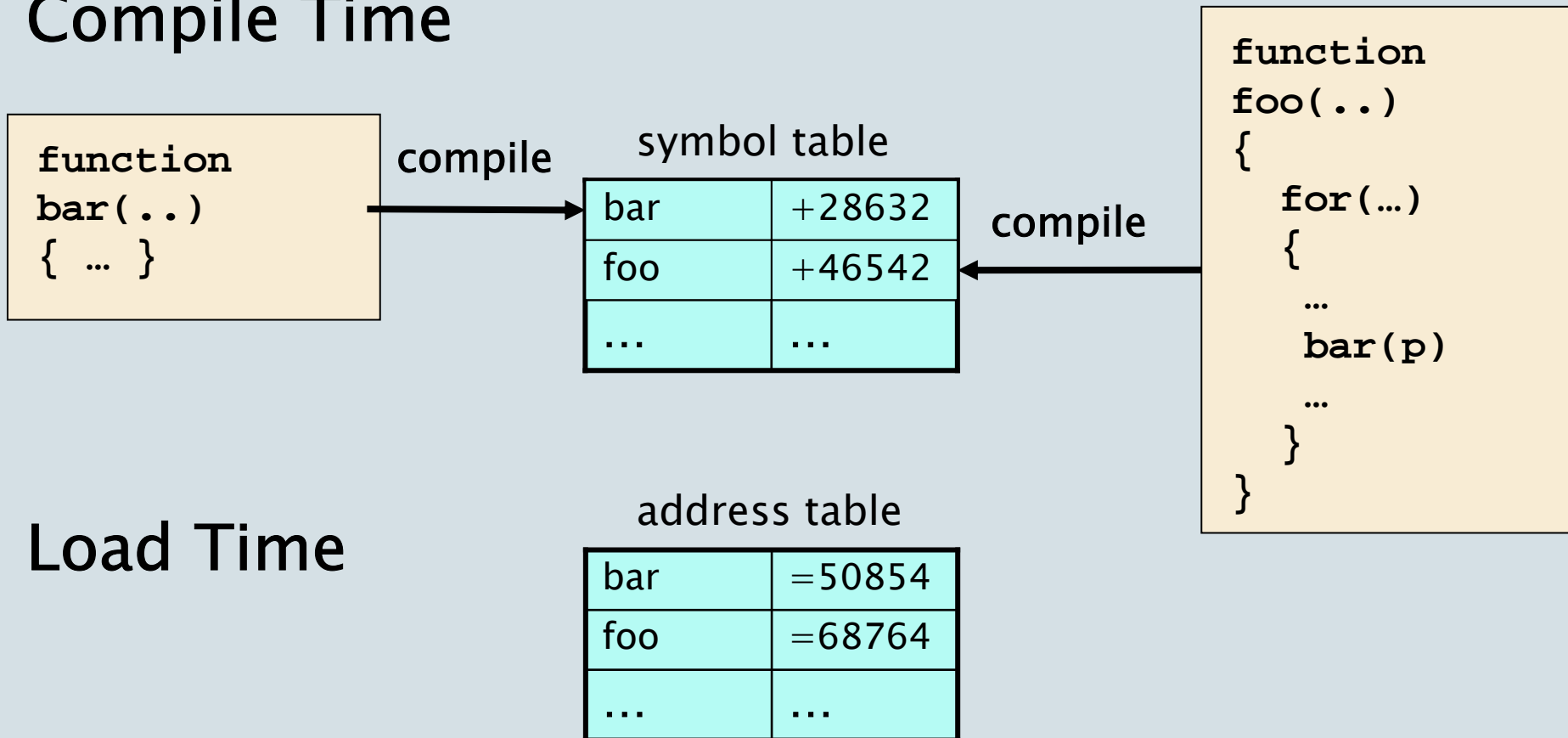
- specifies how GIOP is built on top of TCP/IP

Binding Pattern



How does a system know where to 'return' the result of a function call?

Compile Time



Load Time

How does a system know where to 'return' the result of a function call?

Run Time

| |
|-----------------------|
| function |
| <code>foo(...)</code> |
| <code>{</code> |
| <code>for(...)</code> |
| <code>{</code> |
| <code>...</code> |
| <code>bar(p)</code> |
| <code>...</code> |
| <code>}</code> |
| <code>}</code> |

stack

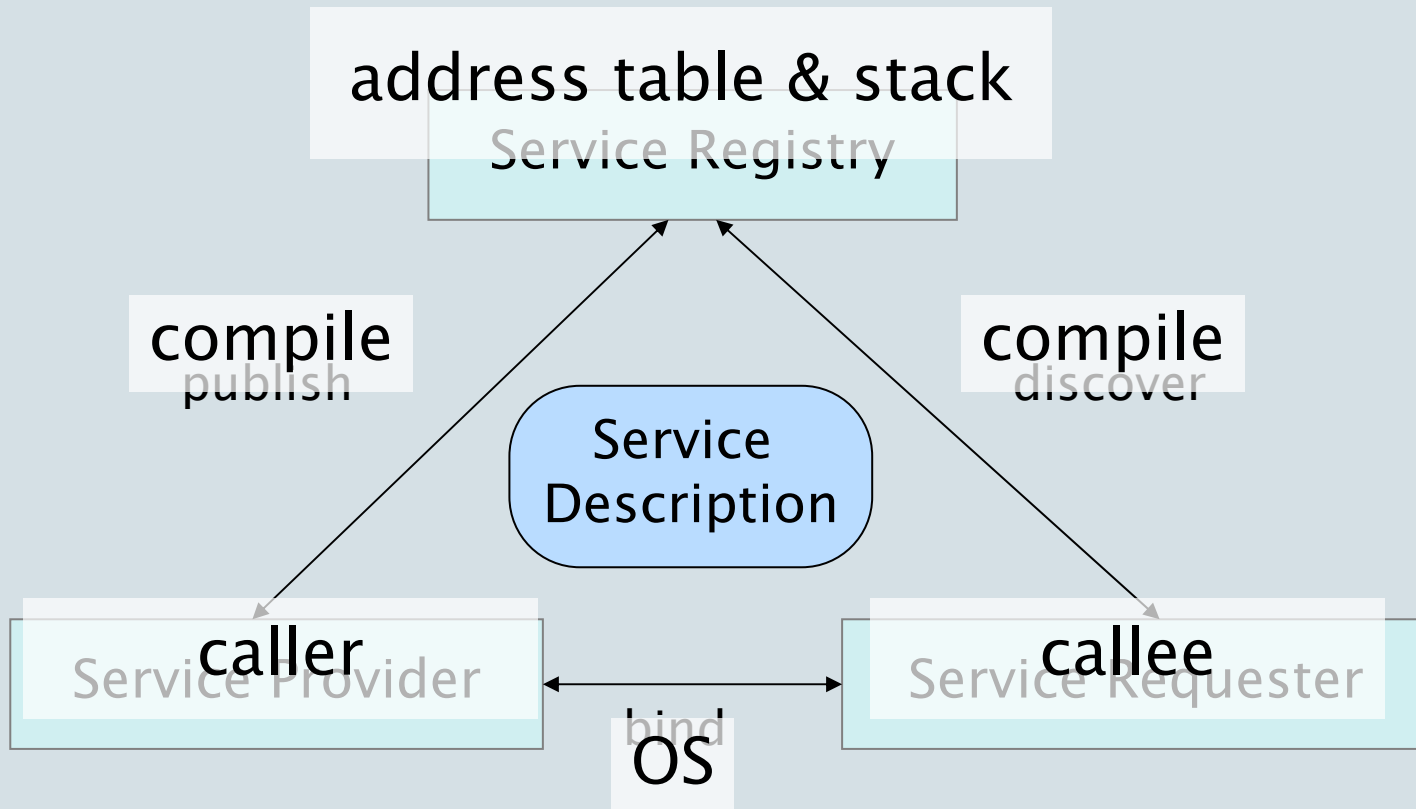
| | |
|-----|-----|
| | |
| ... | ... |
| ... | ... |

| |
|-----------------------|
| function |
| <code>bar(...)</code> |
| <code>{ ... }</code> |

address table

| | |
|------------------|---------------------|
| <code>bar</code> | <code>=50854</code> |
| <code>foo</code> | <code>=68764</code> |
| <code>...</code> | <code>...</code> |

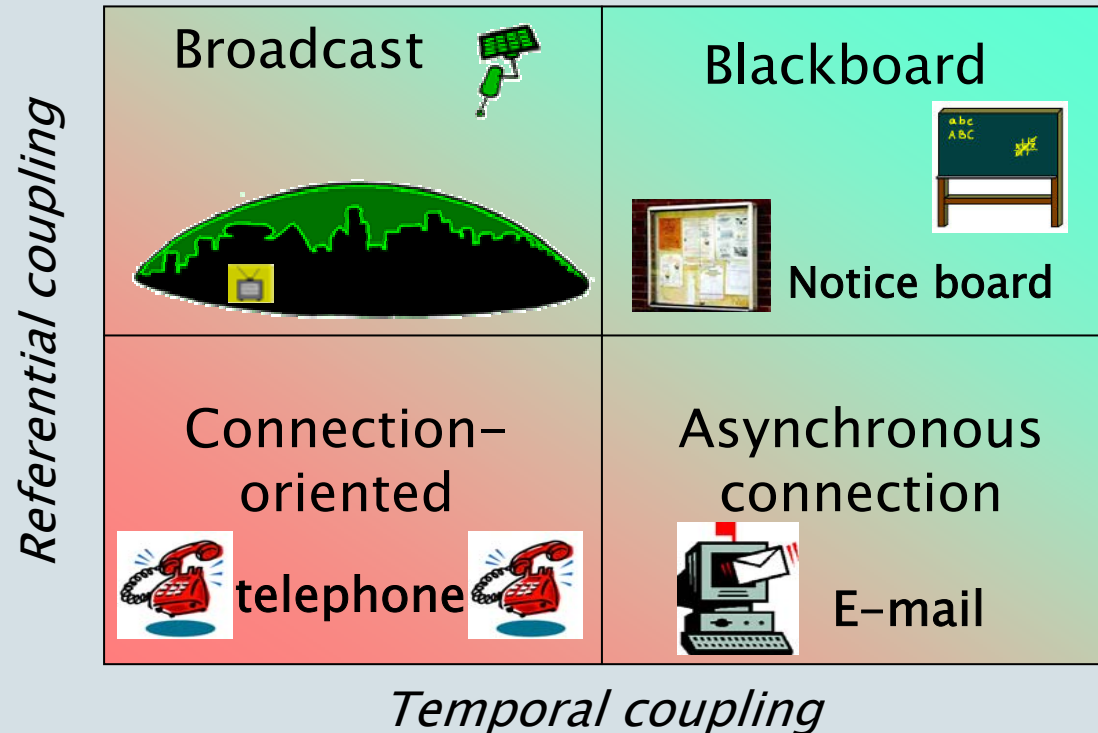
Binding Pattern



- break

Interaction Style and Binding

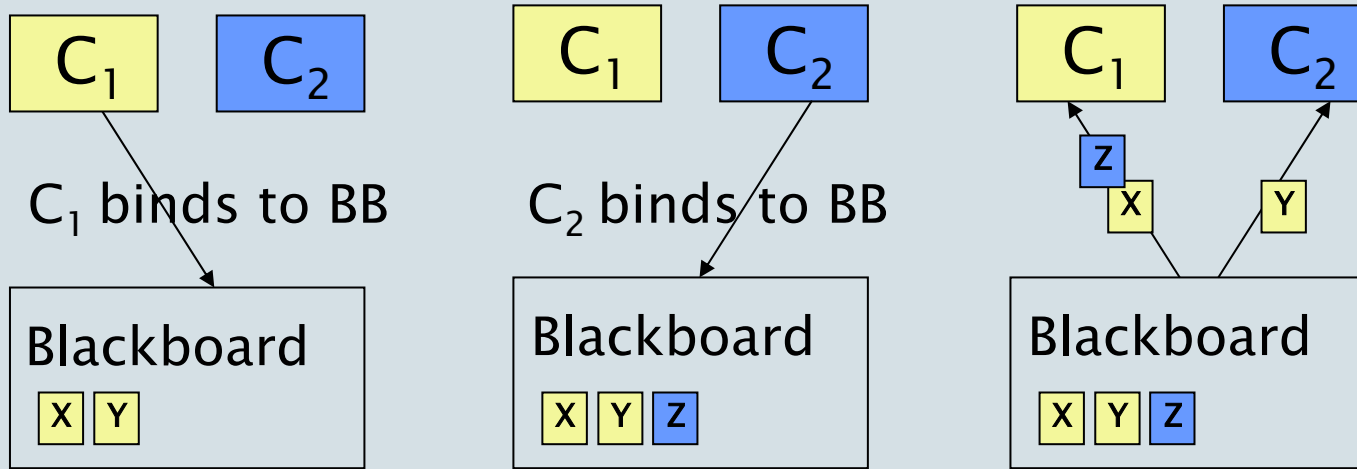
Interaction styles bring
inherent type of binding



Referential coupling : sender has reference to receiver name/address

Temporal coupling: sender and receiver synchronize in time

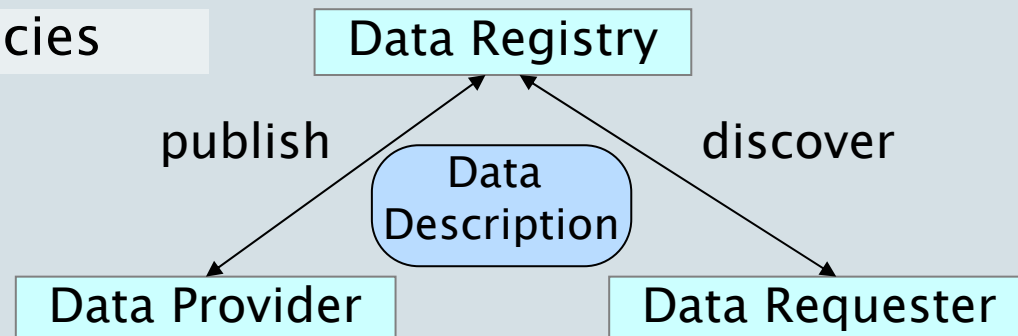
Binding Blackboard style (e.g. JavaSpaces)



C_1 and C_2 read data from the BB – independent of who put it there

C_1 and C_2 do not need references to each other !

No references → no dependencies



Composition time trade-offs

The later composition takes place, the more flexibility the component model/system provides.

Run-time composition requires

- more functionality in the framework
 - more complex framework
 - more expensive to develop
 - use more resources (important in embedded systems)
- more meta-information in the component
 - more expensive to develop
 - more expensive to download
- indirection during run-time
 - performance penalty

Run-time binding

Run-time binding needs run-time support from the component framework. Hence, this needs to be foreseen in the component model.

This design decision whether or not to support run-time binding, affects all systems that are built using the resulting component model.

Many ‘component approaches’ that try to migrate from existing technology are not able to integrate run-time binding.