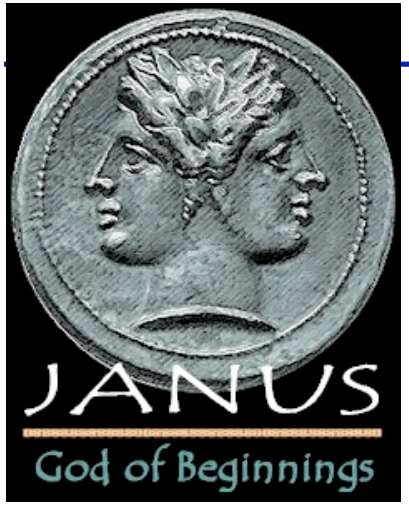


From Objects

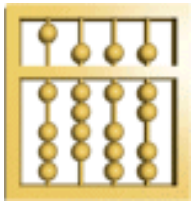
to Components

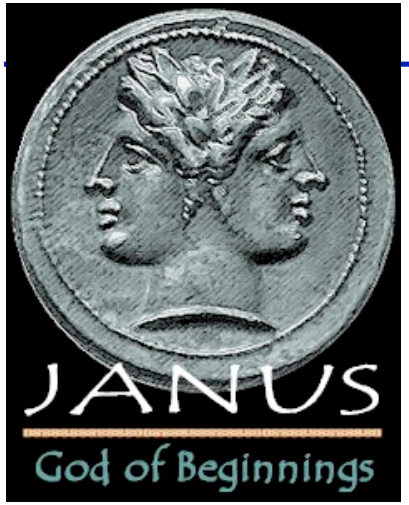
to Services

Formal models for service-oriented interfaces and layered architectures

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 München, Germany





From Objects

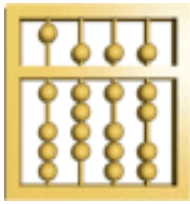
to Components

to Services

4.Lecture: Refinement

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 München, Germany



Notation

Projection:

Let I and I' be two sets of typed channels and $I \subseteq I'$ hold;
for input histories $x \in IH[I']$ we denote by

$x|I \in IH[I]$

the history with the property (let $(c, T) \in I$): $(x|I).c = x.c$

Restriction:

Given service interfaces $IF[I_1 \blacktriangleright O_1]$ $IF[I_2 \blacktriangleright O_2]$ where

$I_1 \subseteq I_2$

$O_1 \subseteq O_2$

we define for $F_2 \in IF[I_2 \blacktriangleright O_2]$

$F_2\dagger(I_1 \blacktriangleright O_1).x = \{y|O_1: \exists\, x': x = x'|I_1 \wedge y \in F_2.x'\}$

Service Refinement

Given two service interfaces $F1 \in IF[I1 \blacktriangleright O1]$

$F2 \in IF[I2 \blacktriangleright O2]$

where $I1 \subseteq I2$

$O1 \subseteq O2$

we call service $F2$ a service refinement of $F1$
if for all input histories $x \in dom(F1)$

$$F2\dagger(I1 \blacktriangleright O1).x \subseteq F1.x$$
$$\wedge\ Dom(F1) \subseteq Dom(F2\dagger(I1 \blacktriangleright O1))$$

Then we write $F1 \approx\!> F2$

Example of a service: Queue

We specify the simple service of a queue

type QIn = {req} ∪ Data

type QOut = Data

Queue

in x: QIn **out** y: QOut
{req}#x = Data#y ∧ $\overline{y} \sqsubseteq$ Data©$\overline{x}$

The input assumption here is:

AsumQ = ∀ x': x' $\sqsubseteq \overline{x} \Rightarrow$ {req}#x ≤ Data#x

Example of a service: Total Queue

We specify the simple service of a queue

type QIn = {req} ∪ Data

type QOut = Data

TotalQueue

in x: QIn **out** y: QOut
min({req}#x, Data#x) = Data#y ∧ $\overline{y} \sqsubseteq Data^{\copyright}\overline{x}$

The input assumption here is: true

Using Queues for Boxes

- We can use the service TotalQueue wherever we need a component Queue since:

  Queue $\approx>$ TotalQueue

  since:

$$
\begin{array}{ll}
 & \min(\{req\}\#x, Data\#x) = Data\#y \\
\wedge & \overline{y} \sqsubseteq Data \copyright \overline{x} \\
\wedge & AsumQ(x) \\
\Rightarrow & req\#x = Data\#y \wedge \overline{y} \sqsubseteq Data \copyright \overline{x}
\end{array}
$$

  The proof this implication proves the service refinement relation.

Remarks

- Service refinement

  $F_1 \approx\!> F_2$

  is proved by proving

  $P_2 \wedge Asume \Rightarrow P_1$

  for the specifying assertions P_1 and P_2 of F_1 and F_2 resp. and the input assumption $Asume$ of F_1
- Service refinement is appropriate for relating services with components (in property refinement the domain is always decreased)
- Service refinement is not compositional
- There is a close relationship between property refinement, chaos completion CC and service refinement

$$F_1 \approx\!> F_2 \quad \Leftrightarrow \quad CC(F_1) \equiv\!> CC(F_2)$$

Example of a service: ResponsiveQueue

We specify the simple service of a queue

type QIn = {req} ∪ Data

type QOut = Data

ResponsiveQueue

in x: QIn **out** y: QOut ∪ {rej}
$\overline{y} = f(\overline{x})$ **where** ∀ d : Data, b : Seq Data, a : StreamQIn: $f(\langle d\rangle \hat{} b \hat{} \langle req\rangle \hat{} a) = \langle d\rangle \hat{} f(b \hat{} a)$ $f(\langle req\rangle \hat{} a) = \langle rej\rangle \hat{} f(a)$

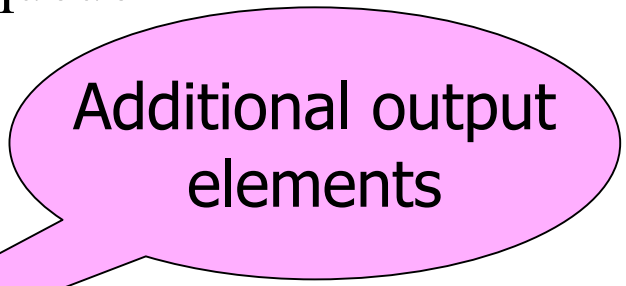
The input assumption here is: true

Example of a service: ResponsiveQueue

We specify the simple service of a queue

type QIn = {req} ∪ Data

type QOut = Data



ResponsiveQueue

in x: QIn
out y: QOut ∪ {rej}

$\bar{y} = f(\bar{x})$

where ∀ d : Data, b : Seq Data, a : StreamQIn:

$f(\langle d\rangle \hat{} b \hat{} \langle req\rangle \hat{} a) = \langle d\rangle \hat{} f(b \hat{} a)$

$f(\langle req\rangle \hat{} a) = \langle rej\rangle \hat{} f(a)$

The input assumption here is: true

Service refinement

- We do not get

  Queue ≈> ResponsiveQueue

  since the types of the channels do not match

Solution: Generalize service refinement

Generalized Service Refinement

Given two service interfaces $F1 \in IF[I1 \blacktriangleright O1]$

$F2 \in IF[I2 \blacktriangleright O2]$

where $I1 \subseteq I2$

$O1 \subseteq O2$

we call service $F2$ a service refinement of $F1$
if for all input histories $x \in dom(F1)$

$$F2\dagger(I1 \blacktriangleright O1).x \subseteq F1.x$$
$$\wedge\ Dom(F1) \subseteq Dom(F2\dagger(I1 \blacktriangleright O1))$$

Then we write $F1 \approx> F2$

Generalized Service Refinement

Given two service interfaces $F1 \in IF[I1 \blacktriangleright O1]$

$F2 \in IF[I2 \blacktriangleright O2]$

where $I1$ subtype $I2$

$O1$ subtype $O2$

we call service $F2$ a service refinement of $F1$
if for all input histories $x \in dom(F1)$

$$F2\dagger(I1 \blacktriangleright O1).x \subseteq F1.x$$
$$\wedge\ Dom(F1) \subseteq Dom(F2\dagger(I1 \blacktriangleright O1))$$

Then we write $F1 \approx> F2$

Definition of subtype

Let C and C′ be two sets of typed channels:

C subtype C′ holds if

$$\forall\, (c, T) \in C : \exists\, (c, T') \in C' : T \subseteq T'$$

In other words

- C′ contains more channels than C and
- channel identifiers c in C have a type with less messages than the type of c in C′

Notation

Projection:

Let I and I' be two sets of typed channels and I subtype I' hold; for input histories $x \in IH[I']$ we denote by

$x|I \in IH[I]$

the history with the property (let $(c, T) \in I$):

$(x|I).c = T©(x.c)$

Restriction:

Given service interfaces $IF[I_1 \blacktriangleright O_1]$ $IF[I_2 \blacktriangleright O_2]$ where

I_1 subtype I_2

O_1 subtype O_2

we define for $F_2 \in IF[I_2 \blacktriangleright O_2]$

$F_2 † (I_1 \blacktriangleright O_1).x = \{y|O_1: \exists\, x': x = x'|I_1 \wedge y \in F_2.x'\}$

Using service refinement

To describe a multi-functional system MS we may first specify many services

S1, S2, S3, S4, …

by independent service specifications

and then specify MS by

$$S1 \approx\!> MS \wedge S2 \approx\!> MS \wedge S3 \approx\!> MS \wedge S4 \approx\!> MS \wedge \ldots$$

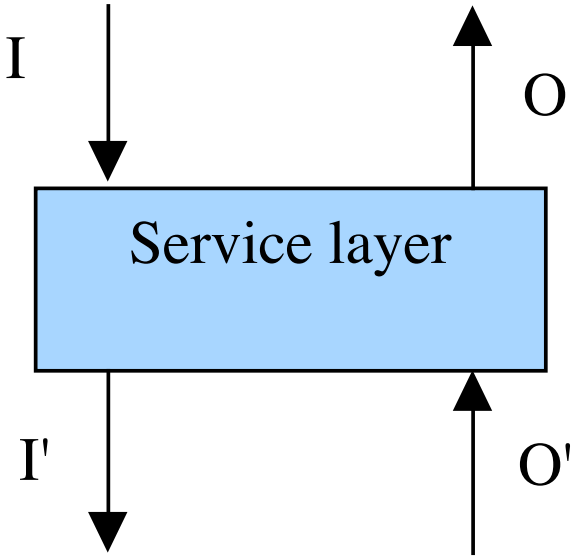
Note: There is a close relationship to multiple inheritance!

Service layers

For service interfaces

$(I \blacktriangleright O)$ export/ *upward interface*

$(O' \blacktriangleright I')$ import/ *downward interface*



where $I \cap O' = \varnothing$ and $O \cap I' = \varnothing$;

service layer L

$$L \in IF[I \cup O' \blacktriangleright O \cup I']$$

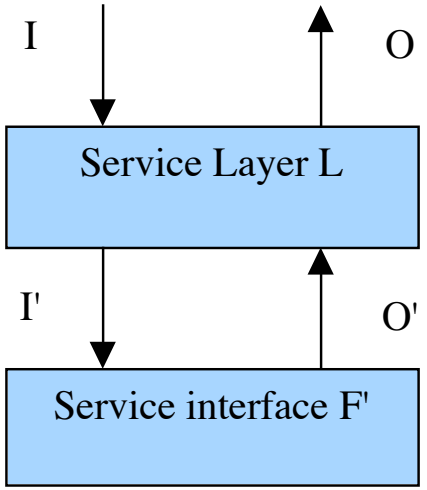
$(I \blacktriangleright O/O' \blacktriangleright I')$ syntactic service layer interface

$IL[I \blacktriangleright O/O' \blacktriangleright I']$. set of layers

Composition of Service Layers

$F' \in IF[I' \blacktriangleright O']$ *import service*

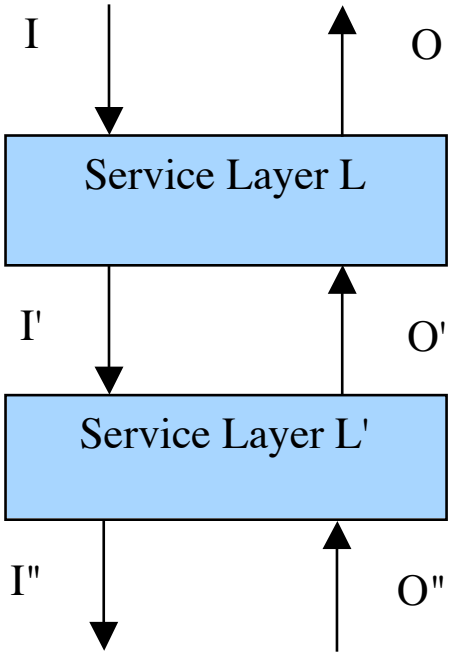
$L \in IL[I \blacktriangleright O / O' \blacktriangleright I']$ service layer



$L[I' \leftrightarrow O']F'$ composition of layer with service

$F \in IF[I \blacktriangleright O]$ *export service*

$F = L[I' \leftrightarrow O']F'$



$L \in IL[I \blacktriangleright O / O' \blacktriangleright I'$
$L' \in IL[O' \blacktriangleright I' / O" \blacktriangleright I"]$

$L[I' \leftrightarrow O']L'$ composition of layers - a layer in $IL[I \blacktriangleright O / O" \blacktriangleright I"]$.

Layered architectures

$F_j \in IF[I_j \blacktriangleright O_j]$ family of export services for $0 \leq j \leq n$.

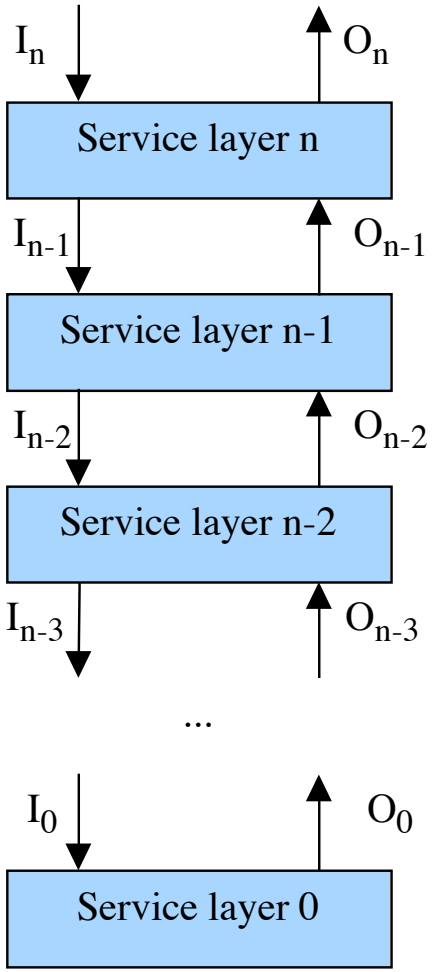
$F_{j+1} \in IF[I_{j+1} \blacktriangleright O_{j+1}]$ *export* service

$F_{j+1} = L_{j+1}[I_j \leftrightarrow O_j]F_j$

$F_j \in IF[I_j \blacktriangleright O_j]$ *import* service

$G_{j+1} \in IF[O_j \blacktriangleright I_j]$ *downward* service

$G_{j+1} = L_{j+1} \dagger (O_j \blacktriangleright I_j)$.



Specifying Services/Layered Architectures

Specify

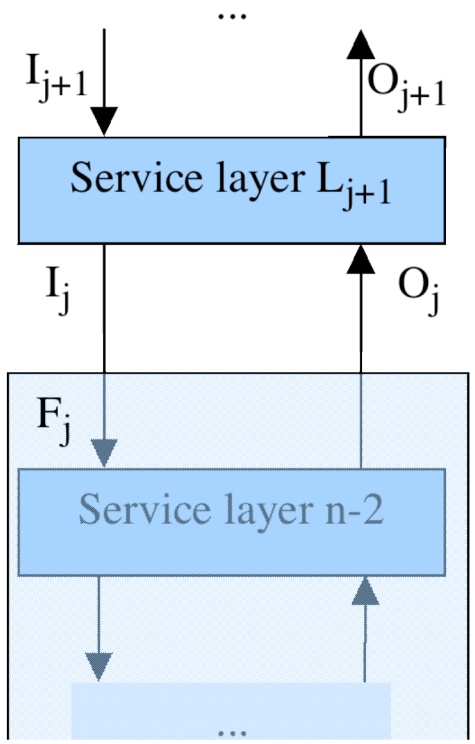
$F_j \in IF[I_j \blacktriangleright O_j]$ family of export services

$L_j \in IL[I_j \blacktriangleright O_j / O_{j+1} \blacktriangleright I_{j+1}]$ family of layers

Prove

$F_{j+1} \approx\!> L_{j+1}[I_j \leftrightarrow O_j]F_j$

L_{j+1} guarantees F_{j+1} under import of F_j



Specifying Services/Layered Architectures

Specify

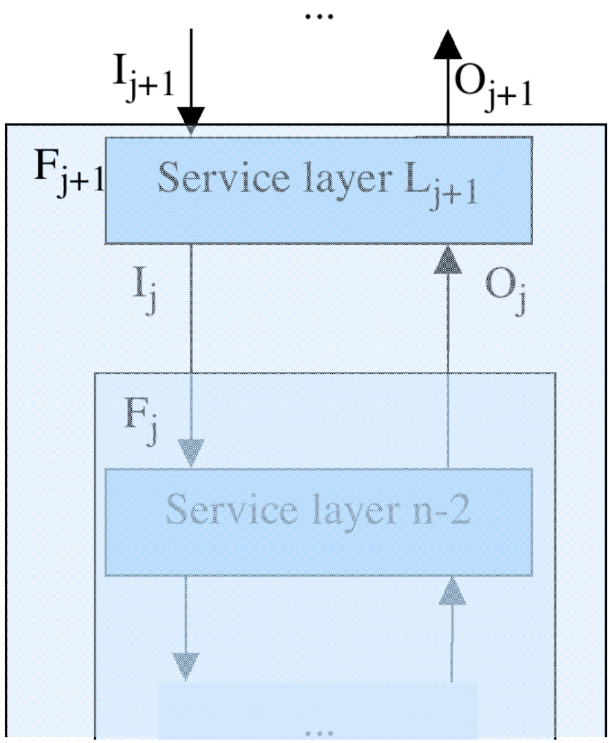
$F_j \in IF[I_j \blacktriangleright O_j]$ family of export services

$L_j \in IL[I_j \blacktriangleright O_j / O_{j+1} \blacktriangleright I_{j+1}]$ family of layers

Prove

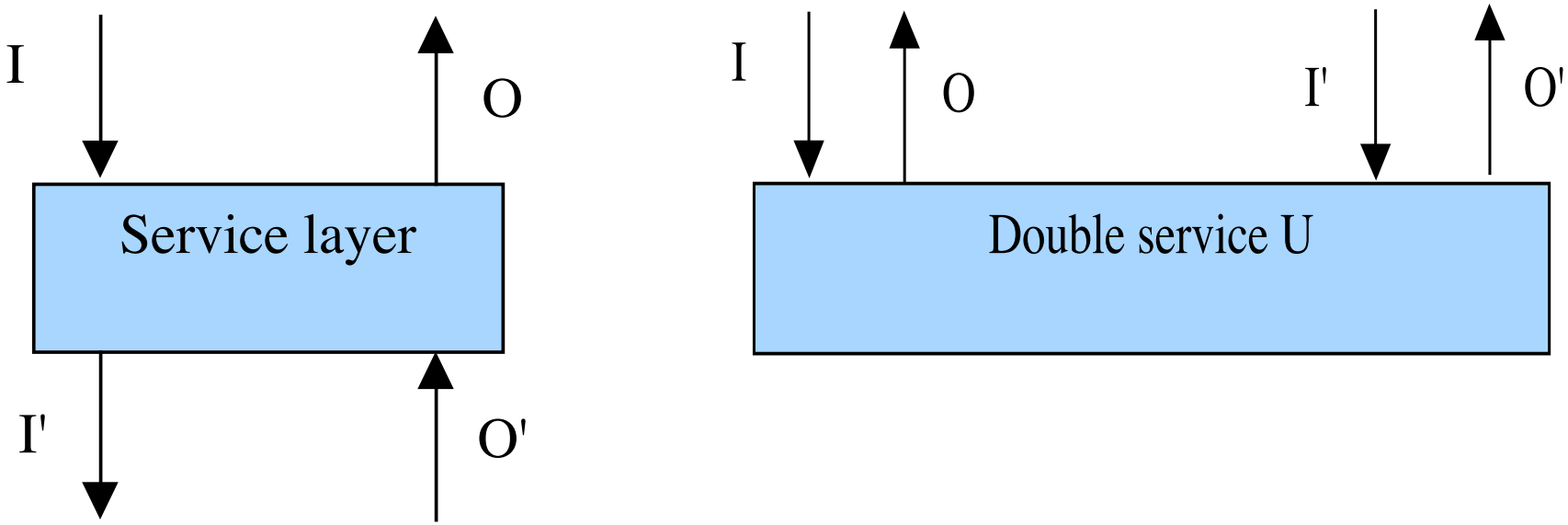
$F_{j+1} \approx\!> L_{j+1}[I_j \leftrightarrow O_j]F_j$

L_{j+1} guarantees F_{j+1} under import of F_j



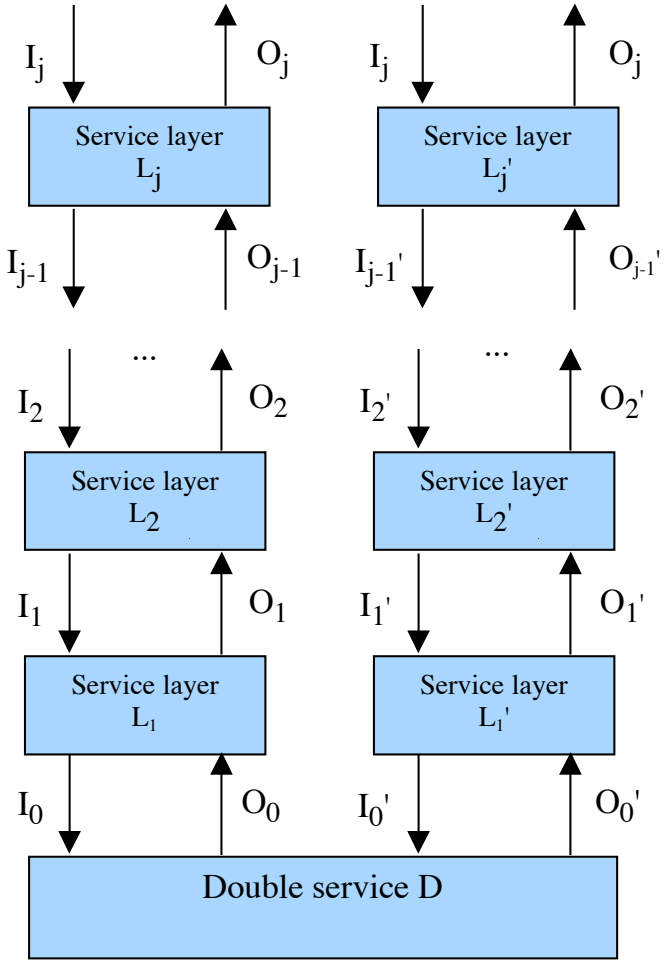
Communication layers

- A communication connection is a special case of a service layer



Communication layers

This way we can build communication layers

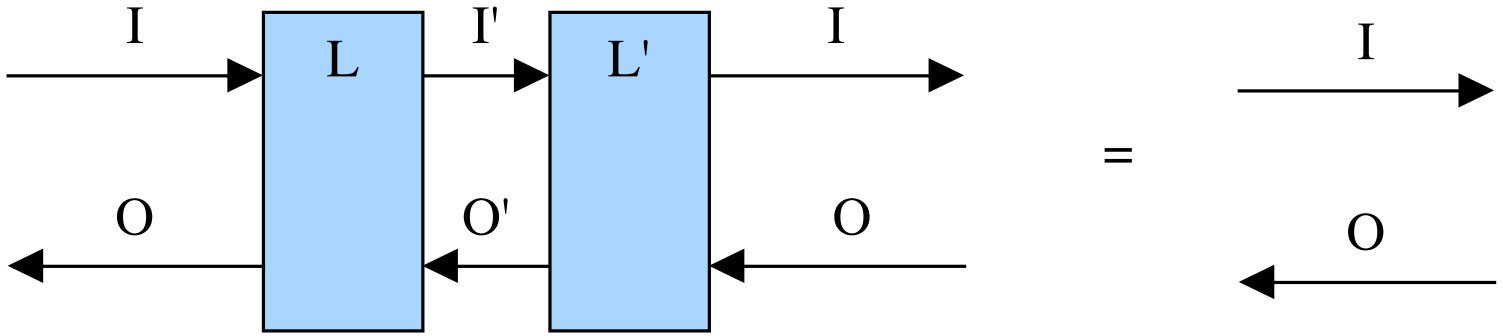


Refinement

- A layer refinement pair are two layers that form the time independent identity

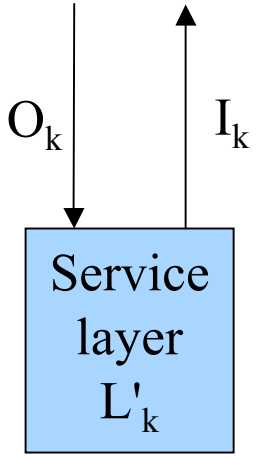
Two layers $L \in IL[I \blacktriangleright O/O' \blacktriangleright I']$ and $L' \in IL[I' \blacktriangleright O'/ \blacktriangleright I]$ are called a *refinement pair* for $(I \blacktriangleright O \,/\, O \blacktriangleright I)$ if

$$L[I' \leftrightarrow O']L' = Id(I \blacktriangleright O)$$

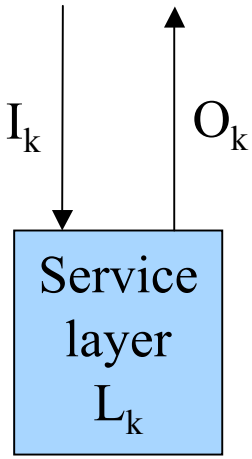


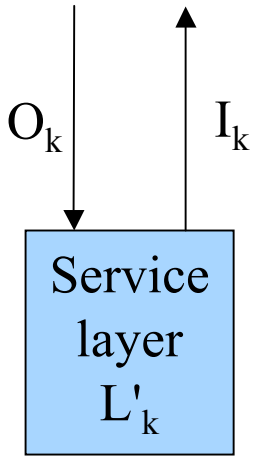
Layered protocols

Layered protocols

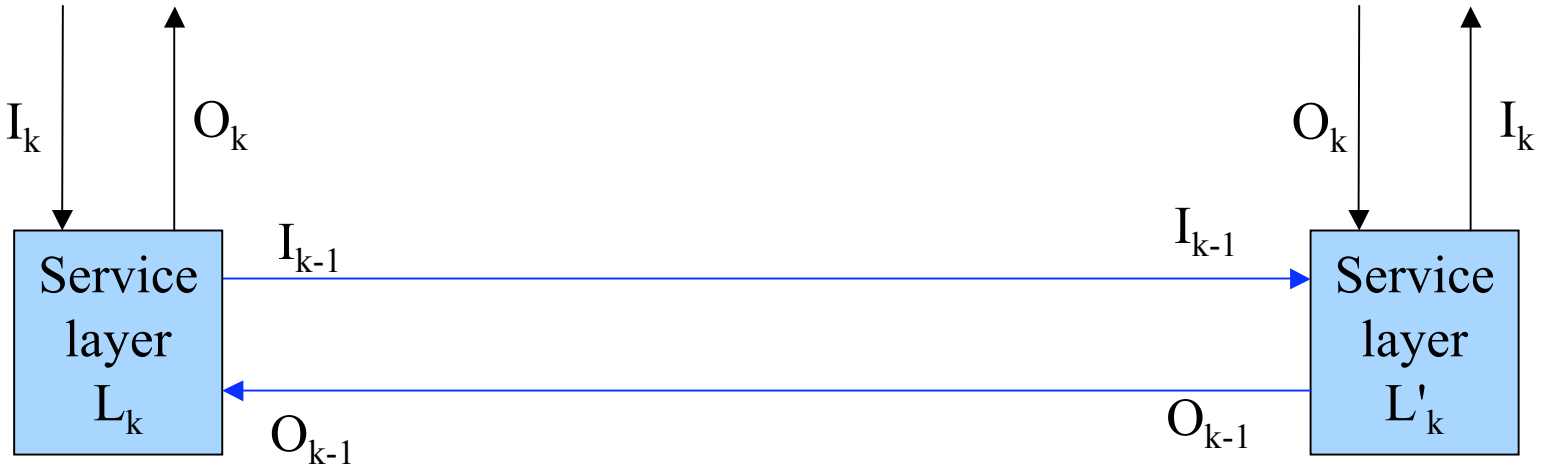


Layered protocols

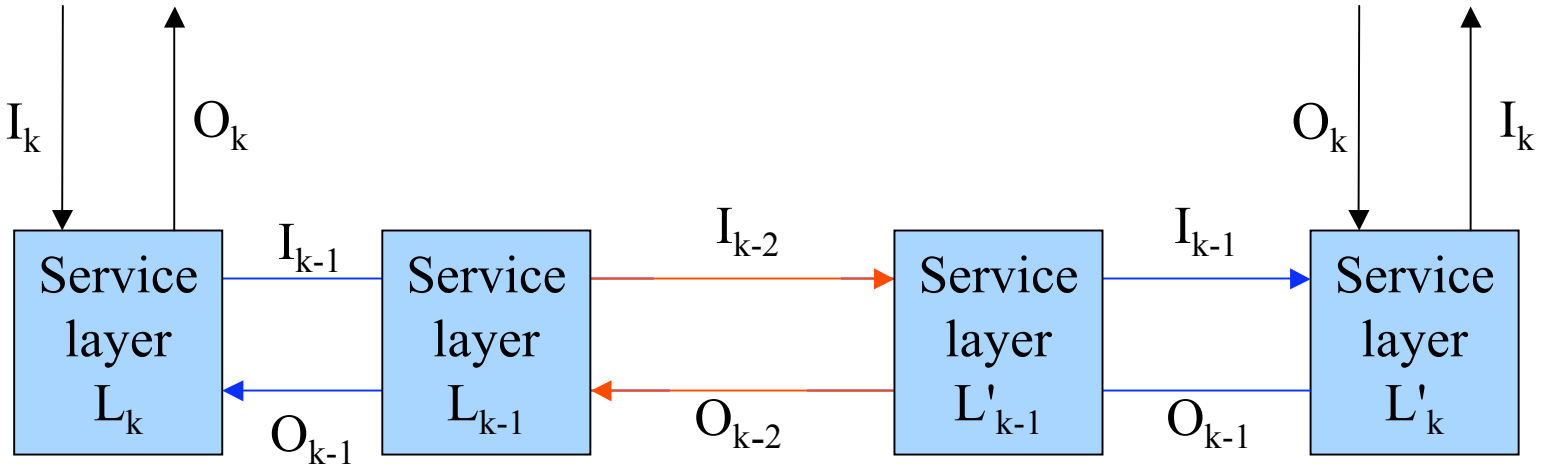




Layered protocols

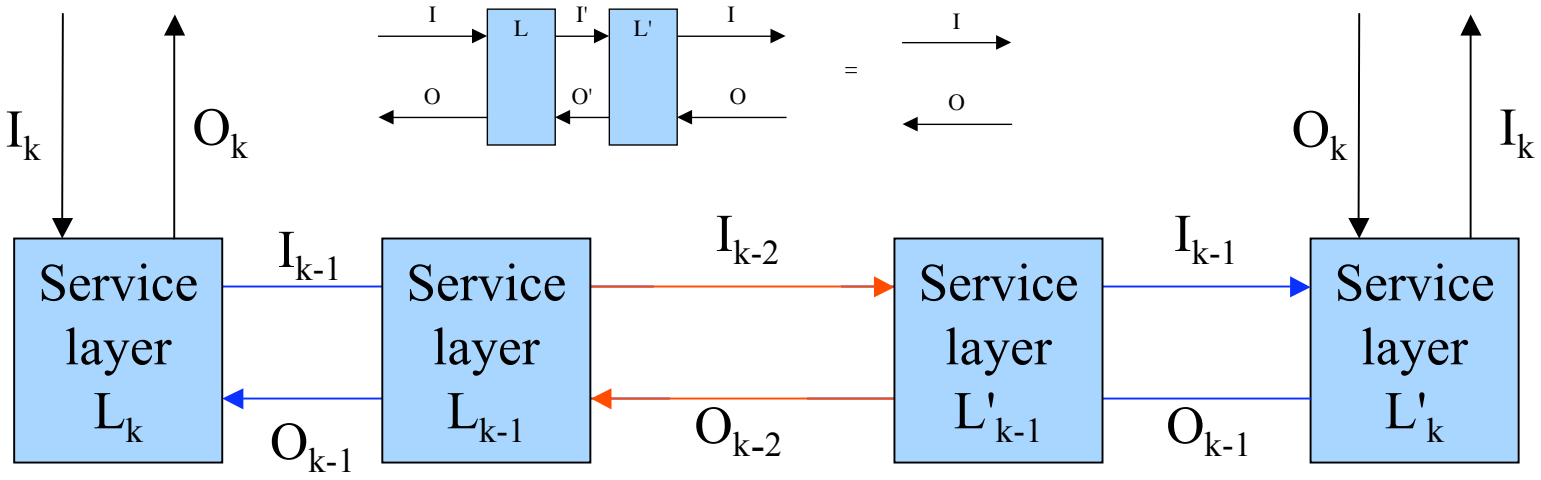


Layered protocols

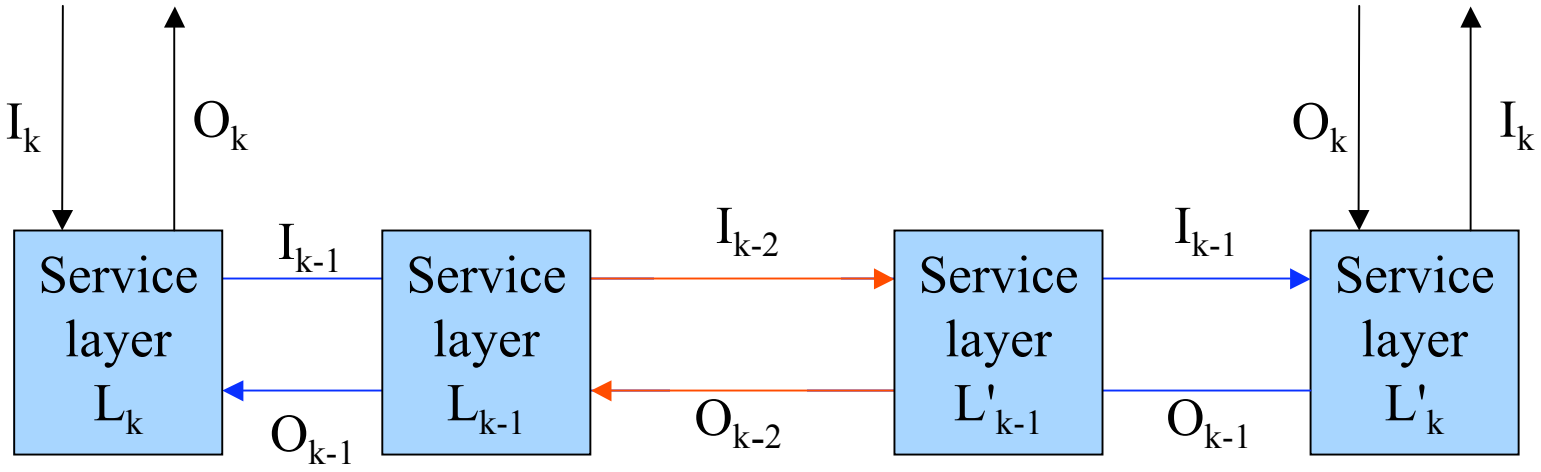


Layered protocols

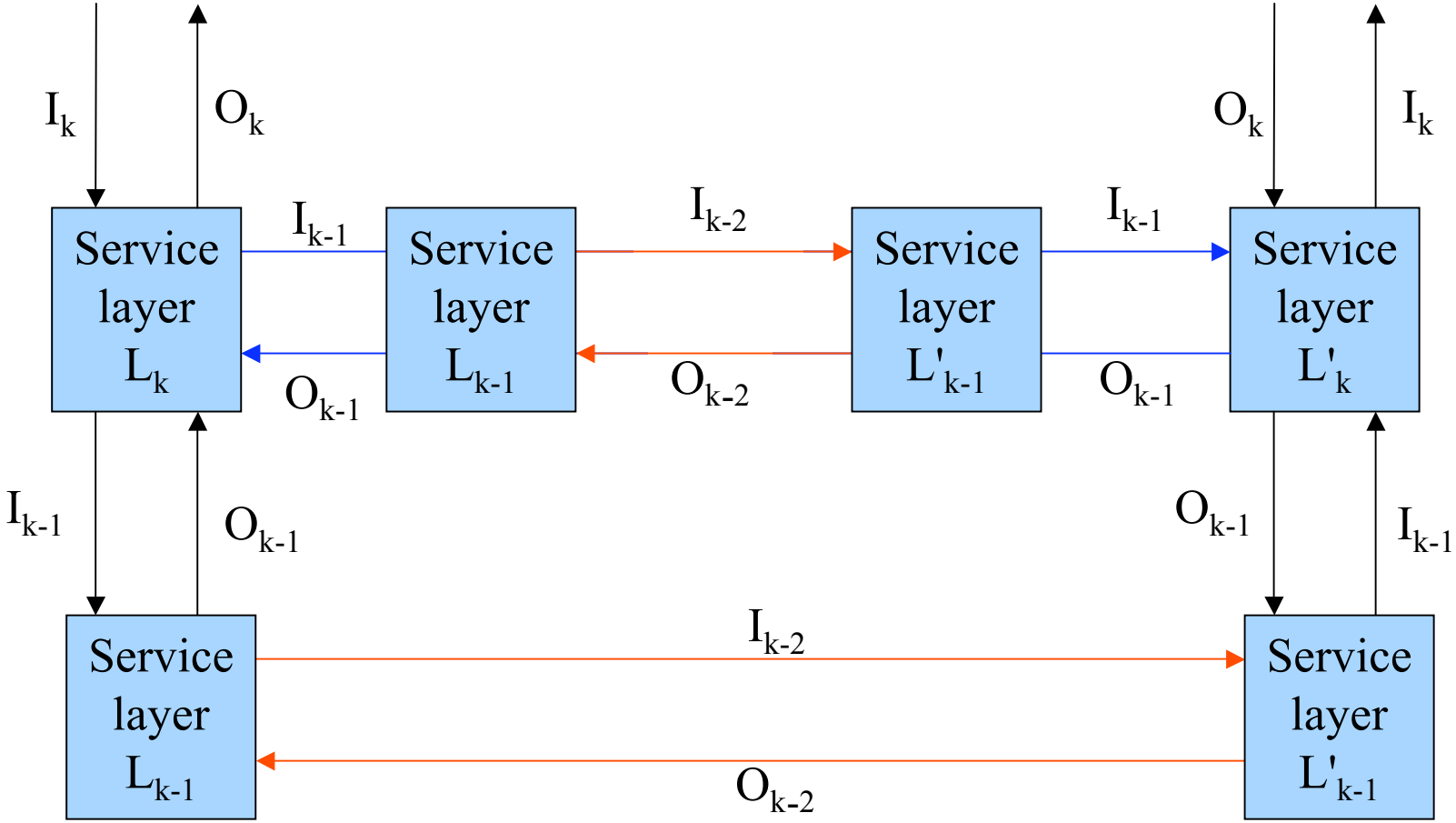
Remember



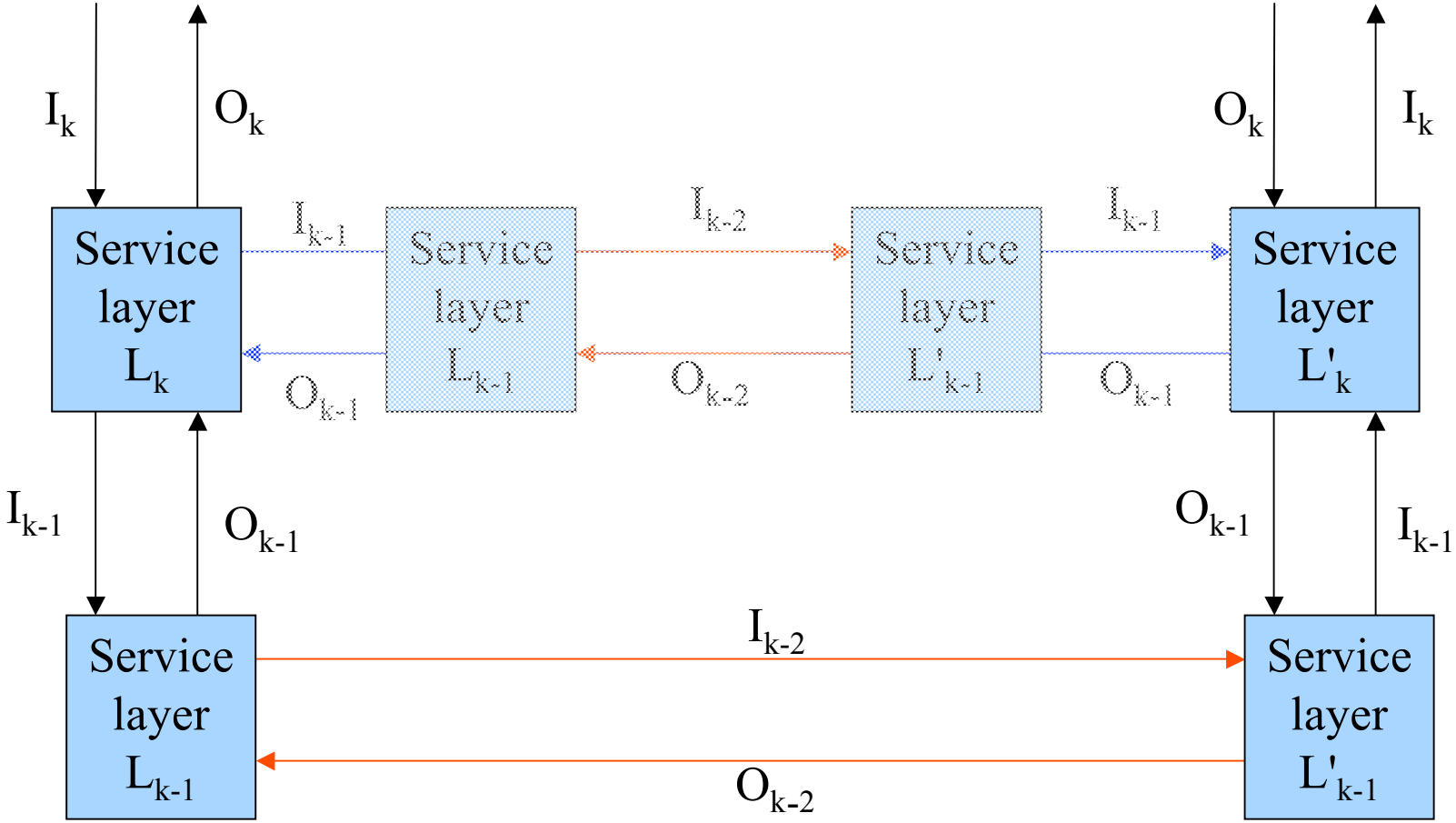
Layered protocols



Layered protocols



Layered protocols



Observations - Problem Areas in Software Development

- Requirements engineering
  - ◇ Requirements are not precisely formulated
  - ◇ Not well supported by formalisms in practice
  - ◇ Structuring concepts missing
- Architecture design
  - ◇ Not well supported by formalisms in practice
  - ◇ Interfaces are not precisely formulated
  - ◇ Interface specification concepts missing
  - ◇ Transition from requirements to architecture not well supported
- Quality assurance
  - ◇ Integration with architecture/interface specification insufficient

Two Structural Views onto Systems

There are (at least) two dimensions in modelling

- Functionality - Requirements Engineering

  A multi-functional system offers many different services; we are interested in a structured view on these services and how they are related

  Structured interface specification of the system

- Decomposition - Design

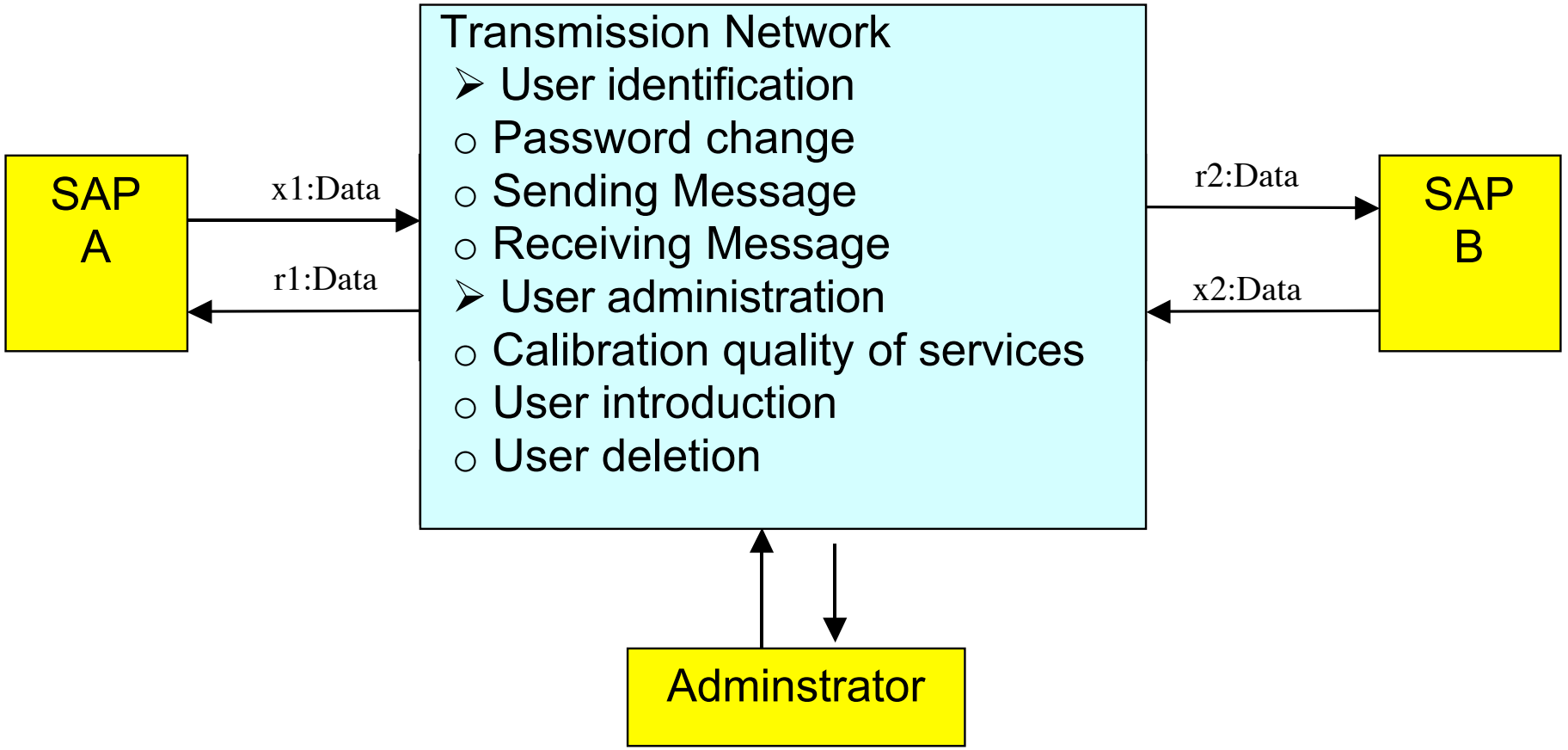
  A system is decomposed in a family of components that cooperate to generate a behaviour to implement the interface specification of the system

  Component architecture of the system

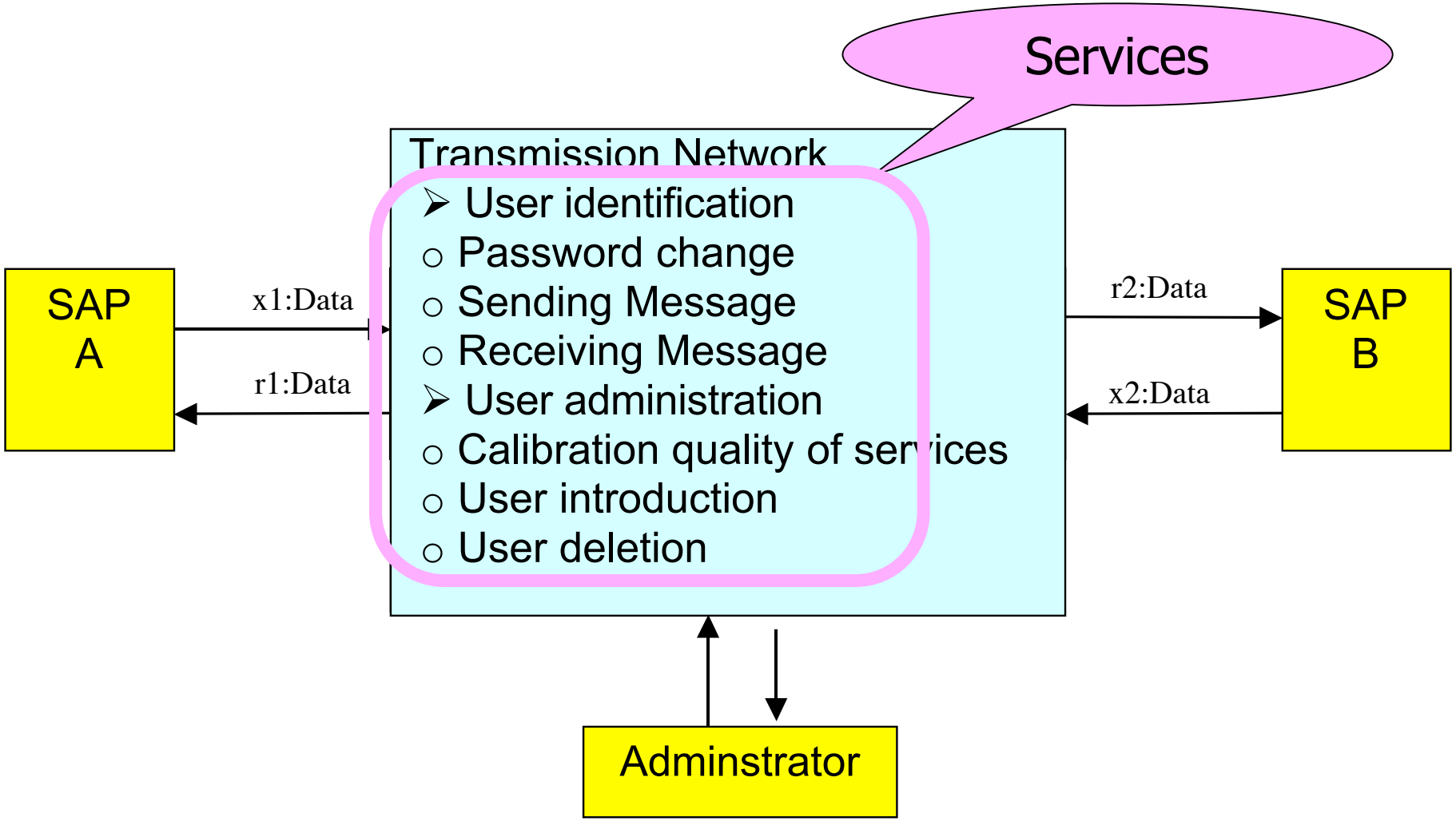
Ergo: Two tasks of structuring

- Structuring Functionality
  - ◇ Less well understood
  - ◇ Key issue: Structured interface specifications in sub-functions
  - ◇ What are the typical relations between sub-functions
- Structuring Component Architecture
  - ◇ Well understood
  - ◇ Studied for quite some time (architectural description languages)
  - ◇ Key issues:
    - Interface specification
    - Composition of interface specifications
- Both views have to be integrated

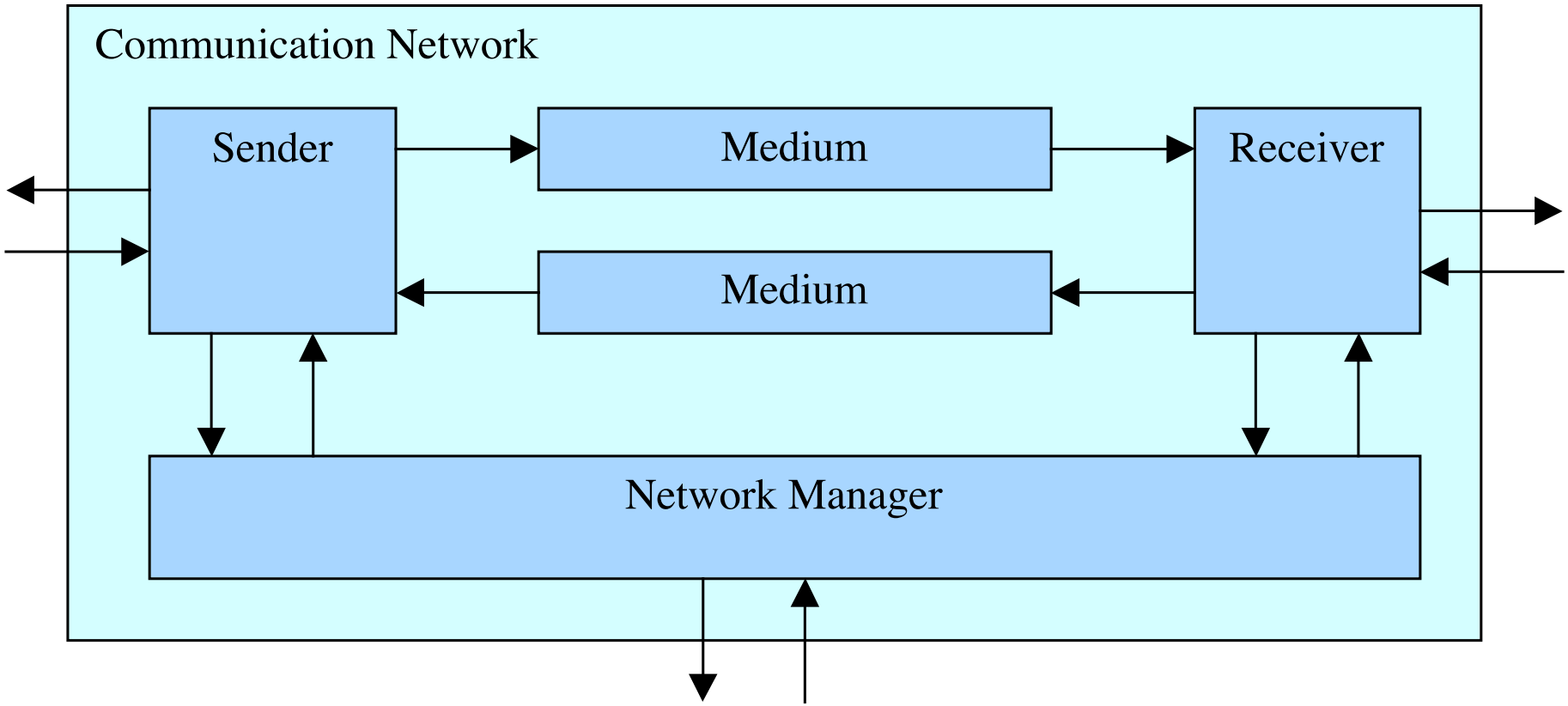
Example: A structured multi-service model



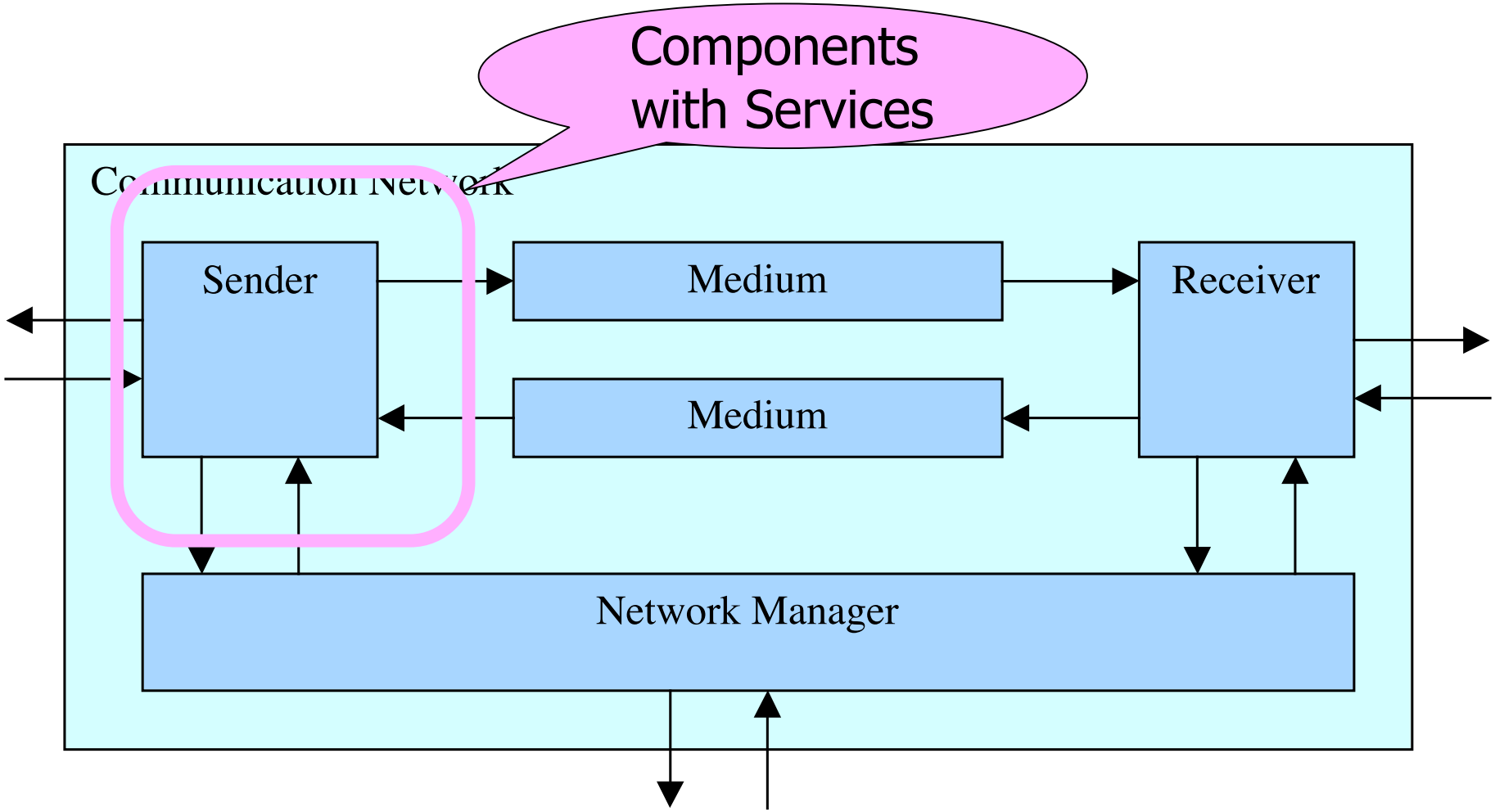
Example: A structured multi-service model



And its architecture



And its architecture



Formal definition of services

Basic philosophy

- Services are formalized by patterns of interactions
- Services are partial behaviours of component interface behaviours
- A component behaviour realizes a family of services

Formal approach:

We specify services in terms of relations on streams

Example: Password service

Let

type PswIn = { quit} ∪ {psw(w): w ∈ Password }

type PswOut = {pok, qok, perror}

PswService(**type** M)

in x': M ⌊ PswIn

out x: M ⌊ PswOut

x = P(x')

where for z: Seq M, x : Stream M ⌊ PswIn, y : Stream : M ⌊ PswOut

valid(w) ⇒ P(‹psw(w)›^z^‹quit›^x) = ‹pok›^z^‹qok›^P(x)

valid(w) ⇒ P(‹psw(w)›^z^‹psw(w')›^x) = P(‹psw(w)›^z^x)

¬valid(w) ⇒ P(‹psw(w)›^x) = ‹perror›^P(x)

P(z^x) = P(‹quit›^x) = undefined

Queue with Password Service

QueuePsw

in x': QIn | PswIn
out y': QOut | PswOut

y' = F(‹›, O).(x') **where for** z: QIn
F(s, L).(‹d›^x) = F(s^‹d›, L).(x)
F(‹d›^s, L).(‹req›^x) = ‹d›^F(s, L).(x)

valid(w) ⇒ F(s, O).(‹psw(w)›x) = ‹pok›^F(s, L).(x)
F(s, L).(‹quit›^x) = ‹qok›^F(s, O).(x)
F(s, L).(‹psw(w)›^x) = F(s, L).(x)
¬valid(w) ⇒ F(s, O).(‹psw(w)›^x) = ‹perror›^F(s, O).(x)
F(s, O).(‹z›^x) = F(s, O).(‹quit›^x) = F(s, O).(x)

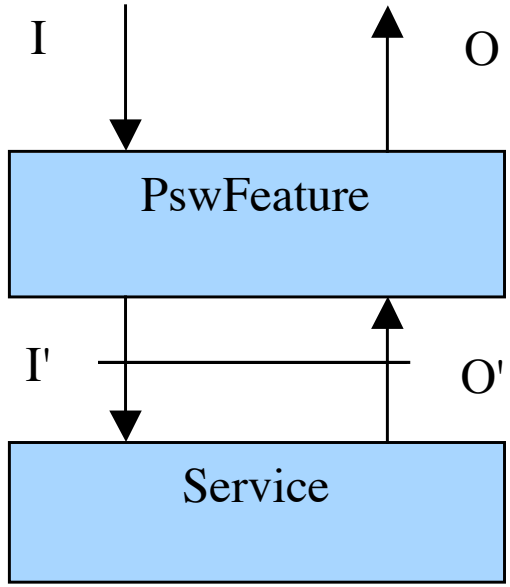
Example: Password feature

Layers allow to define features

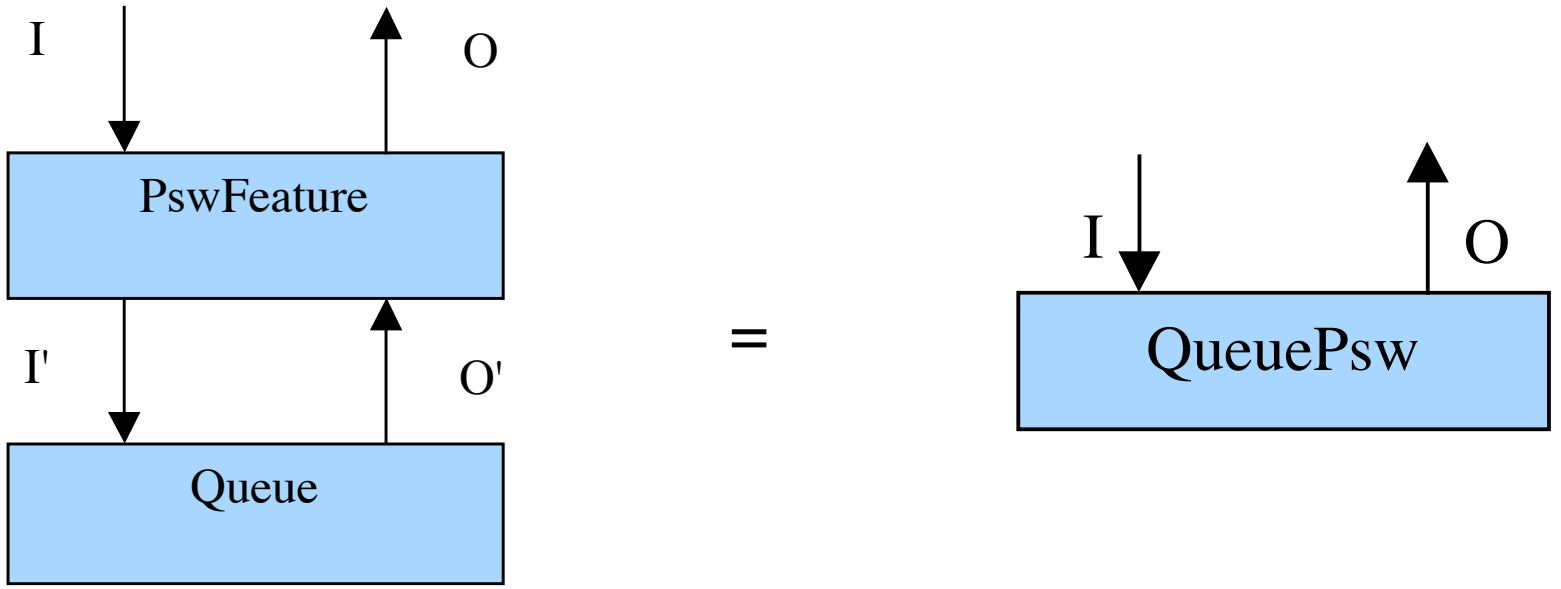
PswService(**type** MIn, MOut)

in x': Min ⌊ PswIn, y : MOut
out y': MOut ⌊ PswOut, x : Min

x = P(x') y' = R(y) **where for** z: Seq M
valid(w) ⇒ P(‹psw(w)›^z^‹quit›^x) = ‹pok›^z^‹qok›^P(x)
valid(w) ⇒ P(‹psw(w)›^z^‹psw(w')›^x) = P(‹psw(w)›^z^x)
¬valid(w) ⇒ P(‹psw(w)›^x) = ‹perror›^P(x)
P(z^x) = P(‹quit›^x) = P(x)
y = A(y') **where for** z: MOut
A(‹z›^x) = ‹z›^A(x)
A(‹pok›^x) = A(‹qok›^x) = A(‹perror›^x) = A(x')



The Queue with password a service refinement



Requirements, services and architectures

Services and functional requirements:

- Structure the user functionality a system in a family of services
- Open issues:
  - ◇ Finding a sufficient family of relations
  - ◇ Verification of these relations

Services and architectures

- The interface for each component can be structure into a family of services
- This structuring can be related to the structuring of the interface of the composed system
- Every service of the composed system uses only a certain subset of the components/services of the architecture

This leads to aspect oriented design!

Outlook and Future Work

- Service-based specification of multi-function systems
- Services described by state machines
- Interaction interfaces
- Conclusions for OO
- Layers as refinement
- Algebra of layers
- Telecommunication layers (multi SAP services)
- Web Services
- Error/ecception handling
- Engineering descriptions
- Tool support