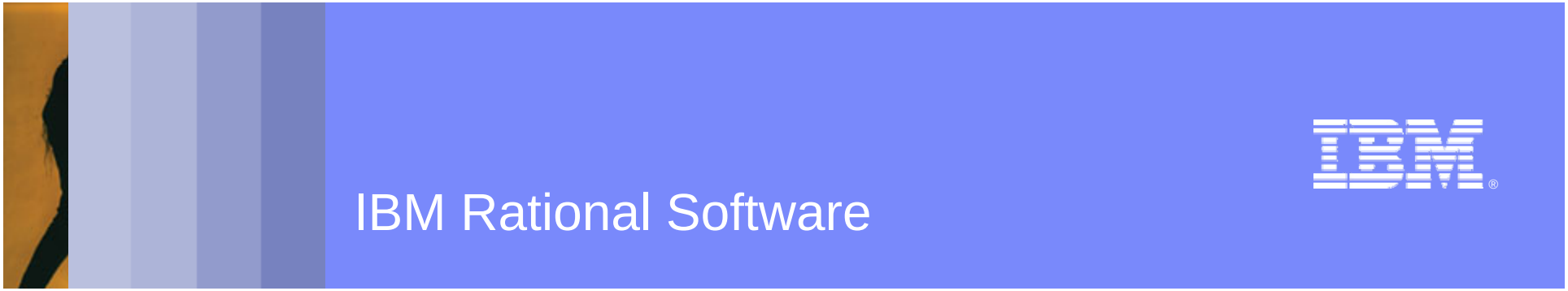


Modeling Software Architectures with UML 2

Bran Selic
IBM Distinguished Engineer – IBM Canada

Rational. software

Outline

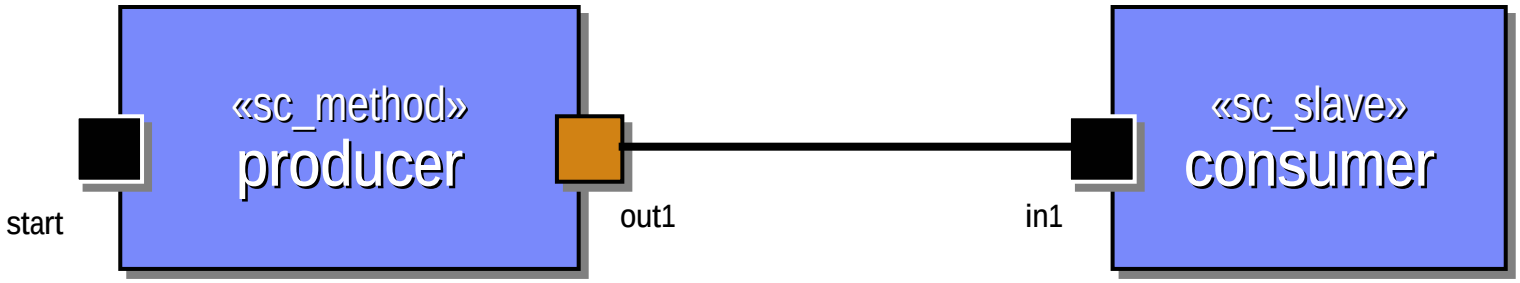
- On Software Architecture and MDD
- Requirements for Modeling Software Architectures
- Architectural Modeling Concepts in UML

A Bit of Modern Software…

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

Can you spot the architecture?

…and Its UML 2 Model



Can you see it now?

Back to Our System……

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << “Sum = “ << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer(“A1”);
A1.out1(link1);
B1 = new consumer(“B1”);
B1.in1(link1);}};
```

Breaking the Architecture….

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
//A1.out1(link1);
B1 = new consumer("B1");
//B1.in1(link1);}};
```

Can you see where?

Breaking the Architecture….

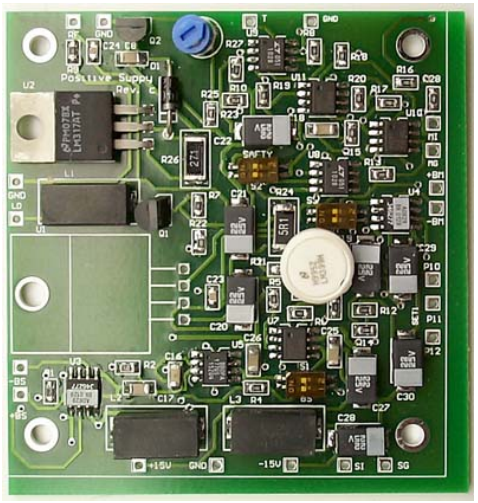
Conclusion:

Models can be quite useful when specifying and communicating software architectures

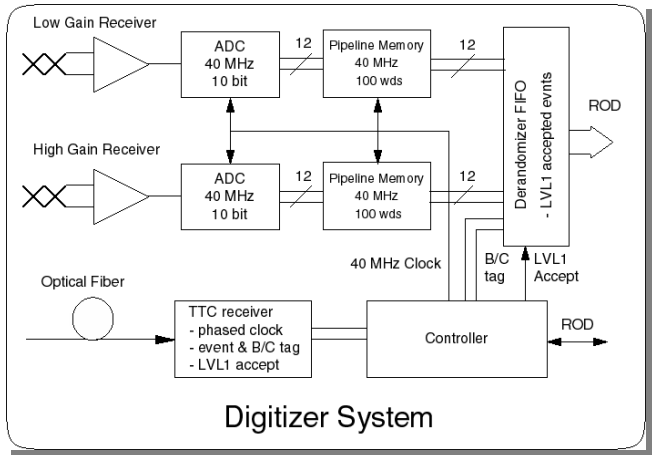
Can you see it now?

Engineering Models

- ***Engineering model:*** A reduced representation of some system that highlights the properties of interest from a given viewpoint



Modeled system



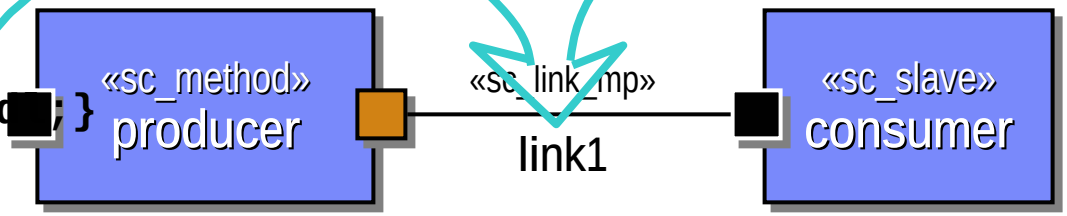
Functional Model

- ***Modeling:*** A fundamental technique for coping with complexity
  - We don't see everything at once – only the important stuff = abstraction
  - We use a representation (notation) that is easily understood

The Software and Its Model

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << end;}
```

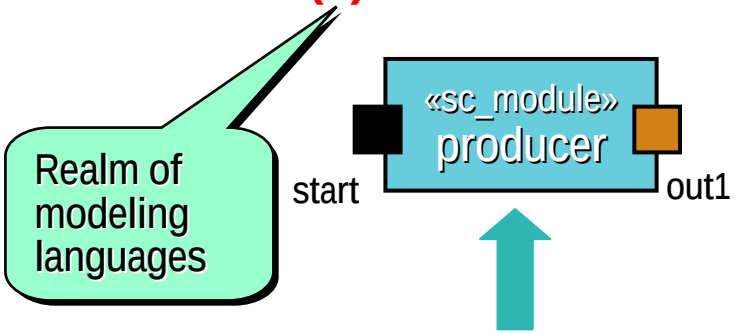
```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```



Model-Driven Development (MDD)

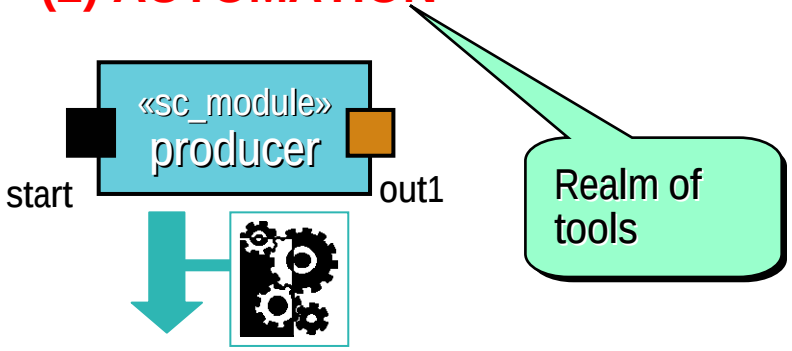
- An approach to software development in which the focus and primary artifacts of development are models (vs programs)
- Based on two time-proven methods:

(1) ABSTRACTION




```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

(2) AUTOMATION

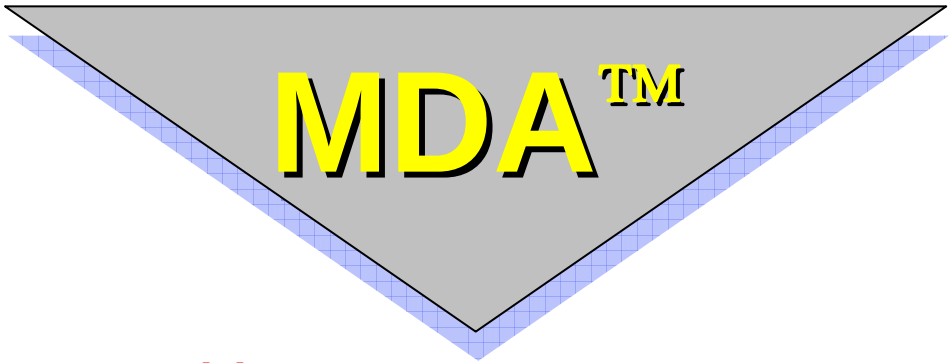



```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

Model-Driven Architecture (MDA)

- An OMG initiative to support model-driven development through a series of open standards

(1) ABSTRACTION **(2) AUTOMATION**



(3) OPEN STANDARDS

- ***Modeling languages***
- ***Interchange standards***
- ***Model transformations***
- ***Software processes***
- ***etc.***

What is Software Architecture?

- IEEE Standard 1471-2000: *Architectural Description of Software-Intensive Systems*

- ***Architecture:*** The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
  - "fundamental" ⇒ irrelevant details are omitted ⇒ abstraction
  - "organization" ⇒ structural *and behavioral*
  - "components" ⇒ architecture involves decomposition into parts
  - "relationships" ⇒ parts are coupled structurally and dynamically
  - "guiding principles" ⇒ like the basic tenets of a constitution

Architecture and Modeling

- *Software architectures are specified by models:*
  - To architect is to model

⇒ Software modeling languages used by software architects must have appropriate architectural modeling capabilities

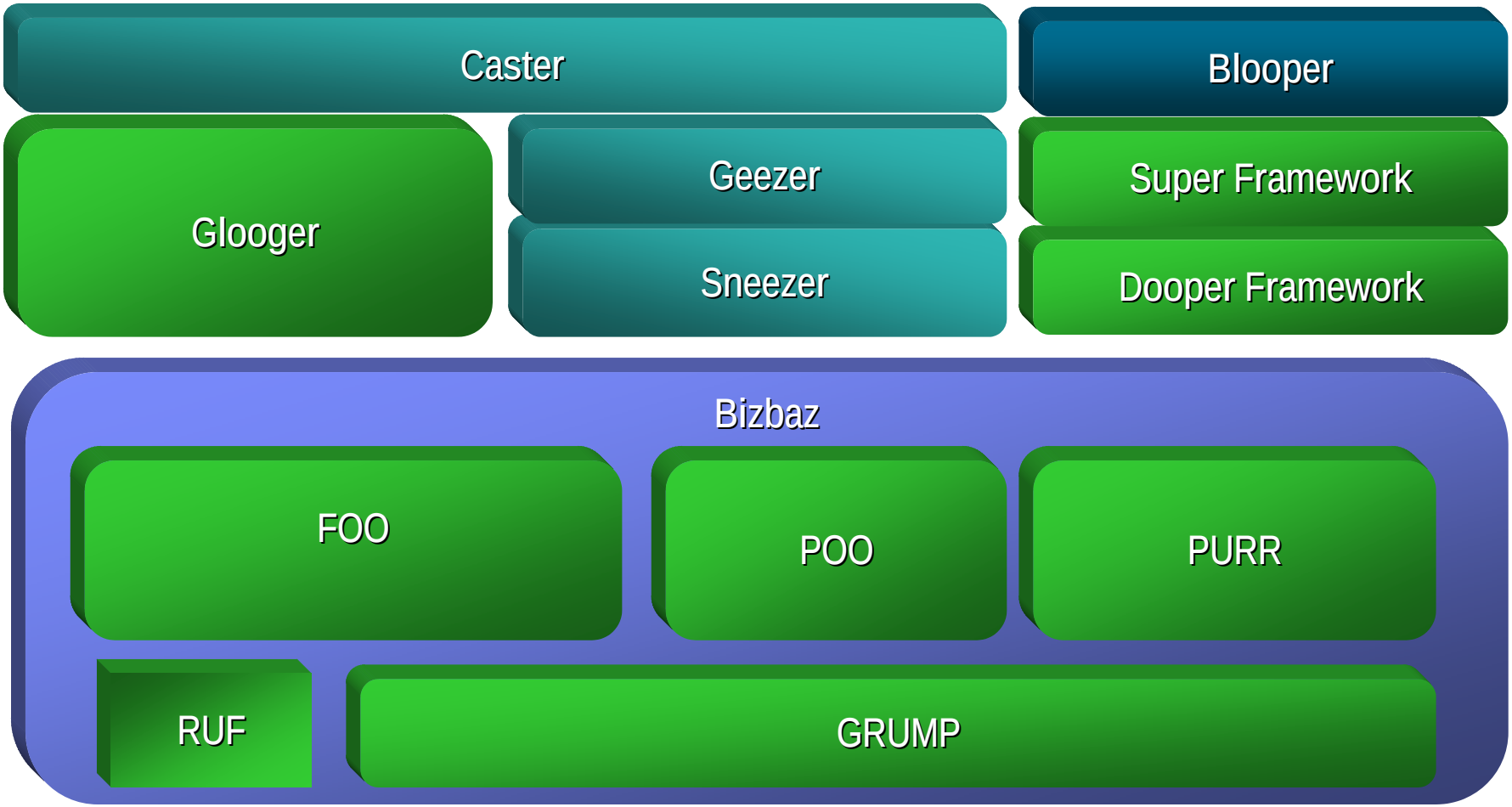
- *What are those capabilities?*

Outline

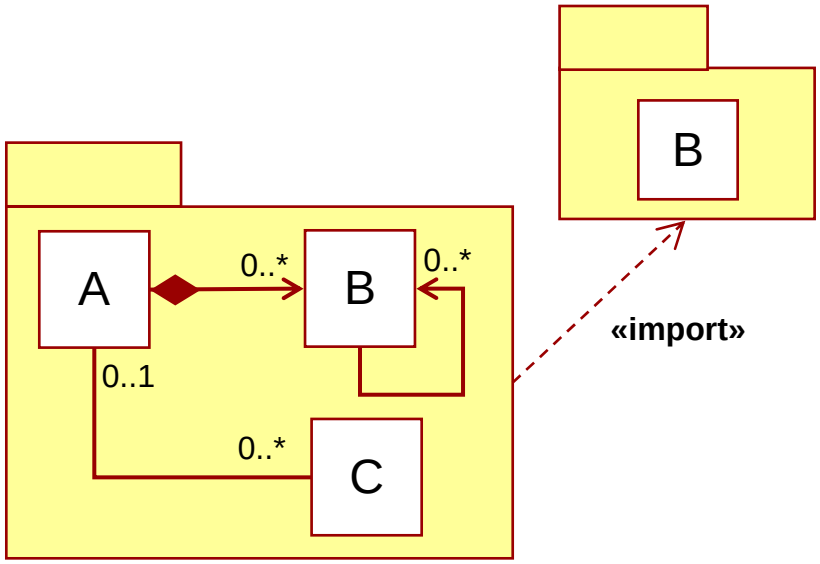
- On Software Architecture and MDD
- Requirements for Modeling Software Architectures
- Architectural Modeling Concepts in UML

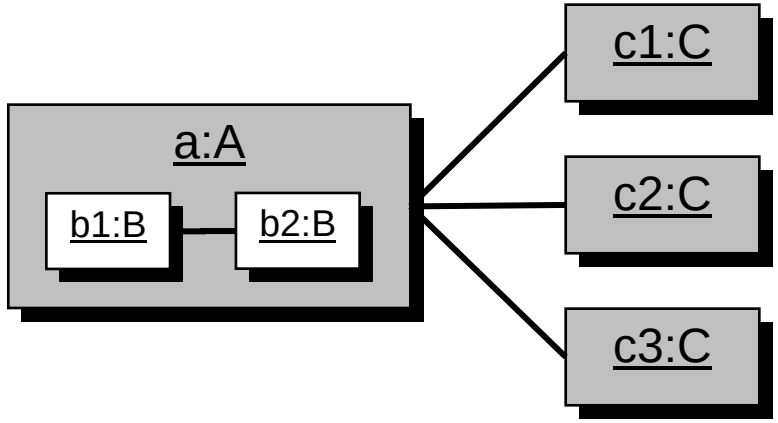
Sample software architecture diagram

- What does it actually mean?



Software Architecture: Run-Time vs Design-Time

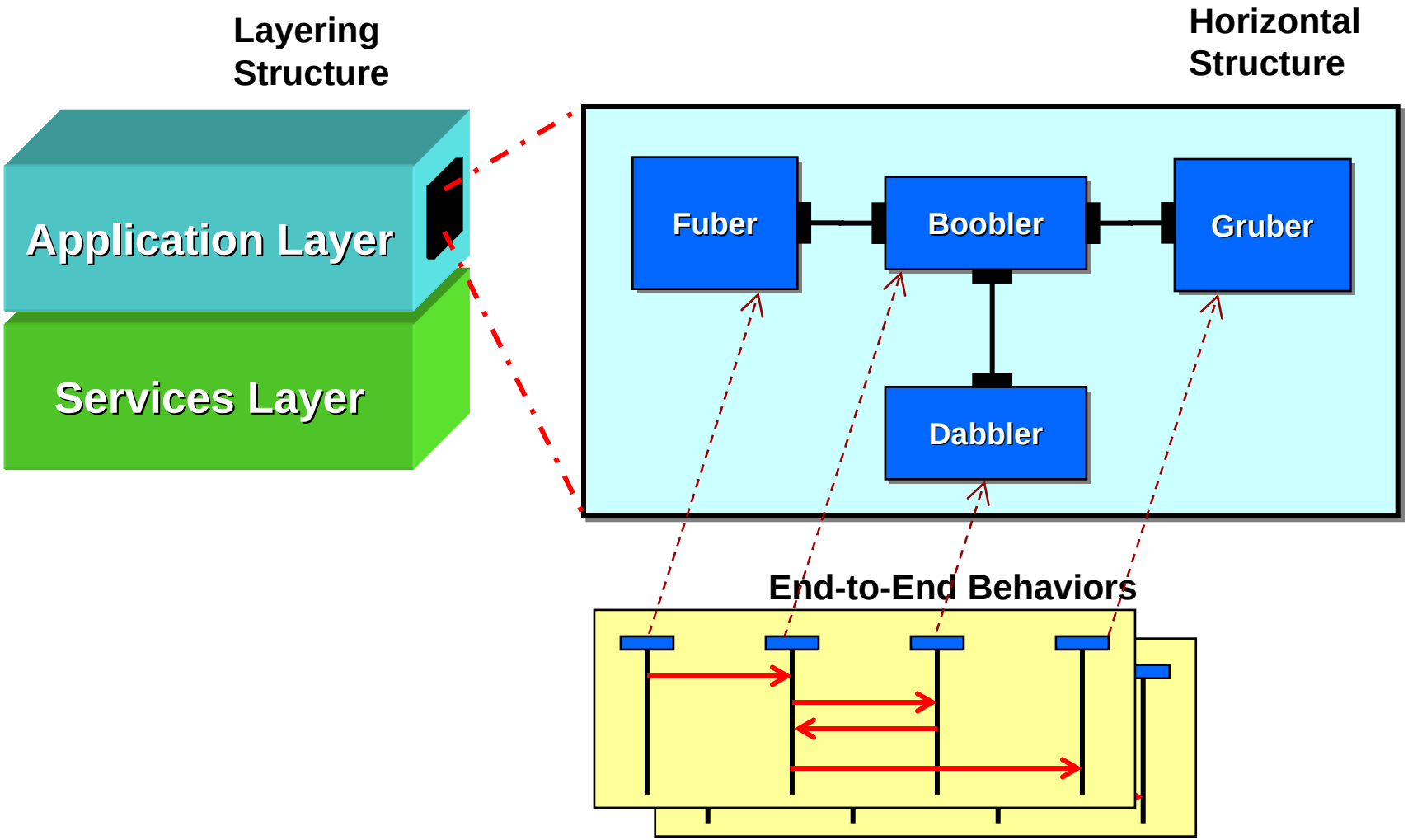




- ***Design-time architecture:*** the static organization of the system specification in a design repository
- ***Run-time architecture:*** the dynamic organization of instances executing in a computer

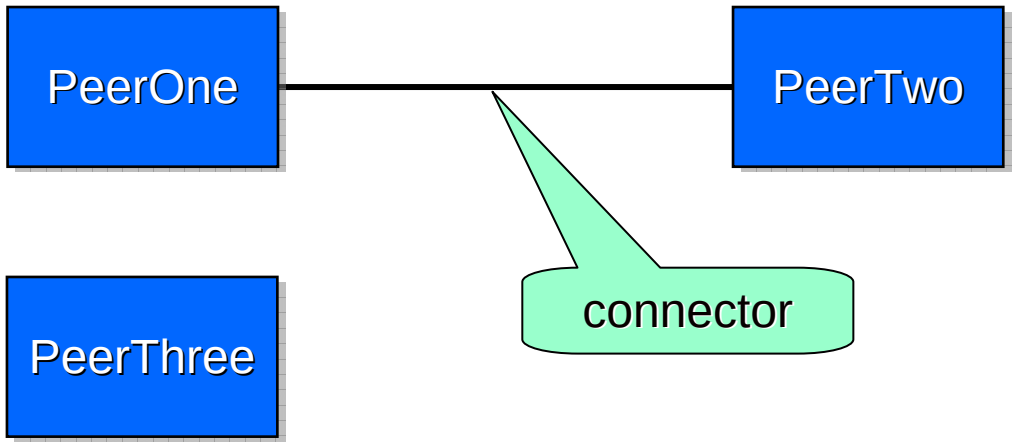
The two are formally related but are distinct!

A Simplified Run-Time Architecture Model Example



The Communicating Peers Structural Pattern

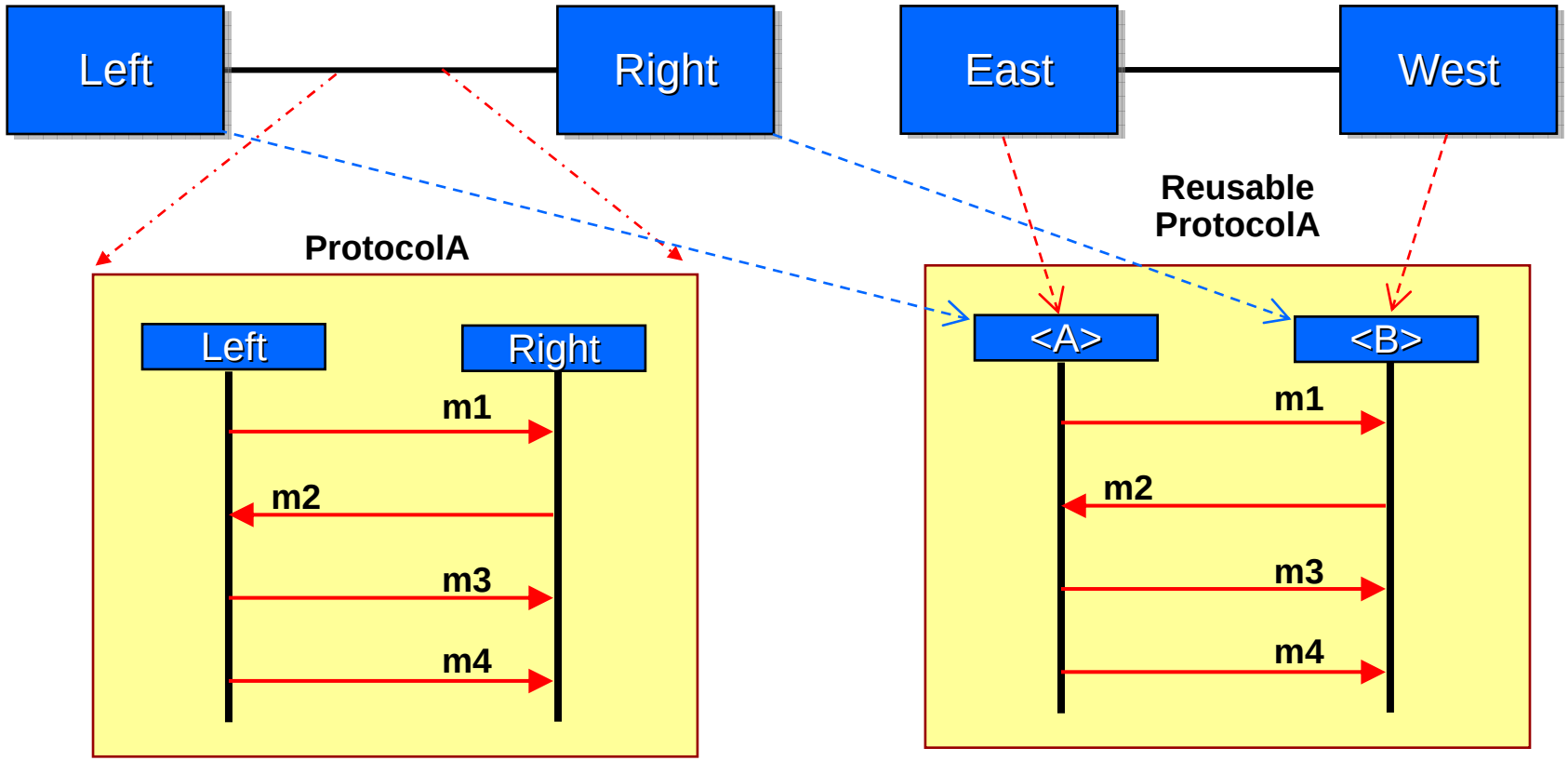
- Two (or more) components that collaborate to achieve some greater objective
  - Each can exist independently of the other



- Connectors clearly specify the intended couplings between components
  - Unconnected components cannot affect each other directly
  - Explicit specification of architectural constraints

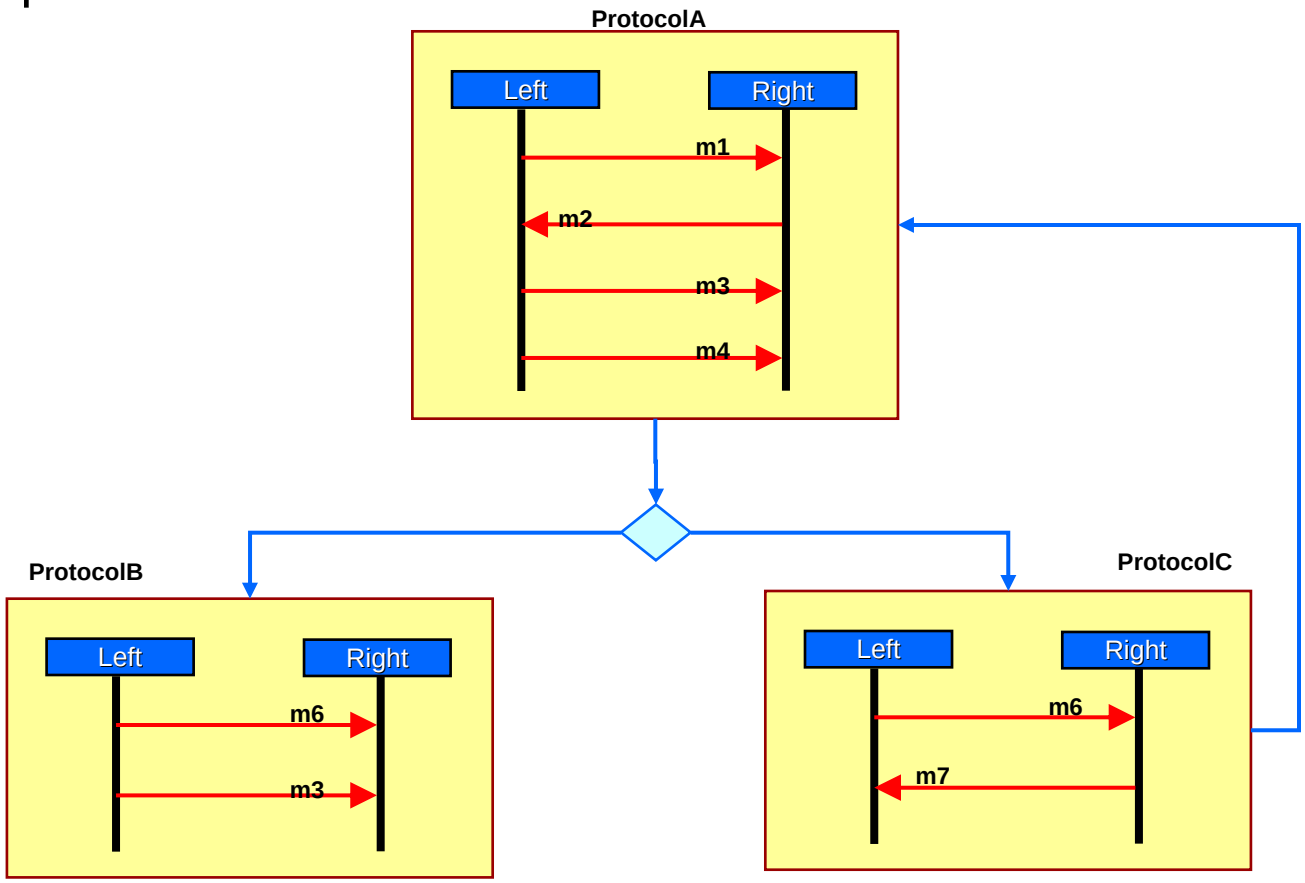
Protocols

- A specification of only valid interactions along a connector
  - Ideally, defined as a reusable behavioral “component”



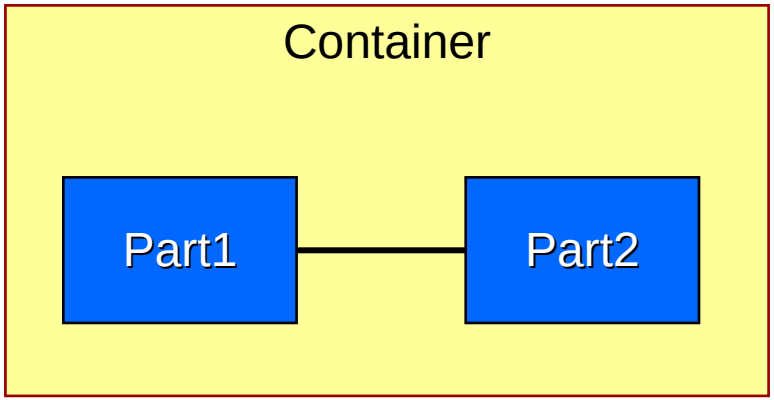
Protocol Composition

- Sometimes it is useful to combine simpler protocols into more complex ones



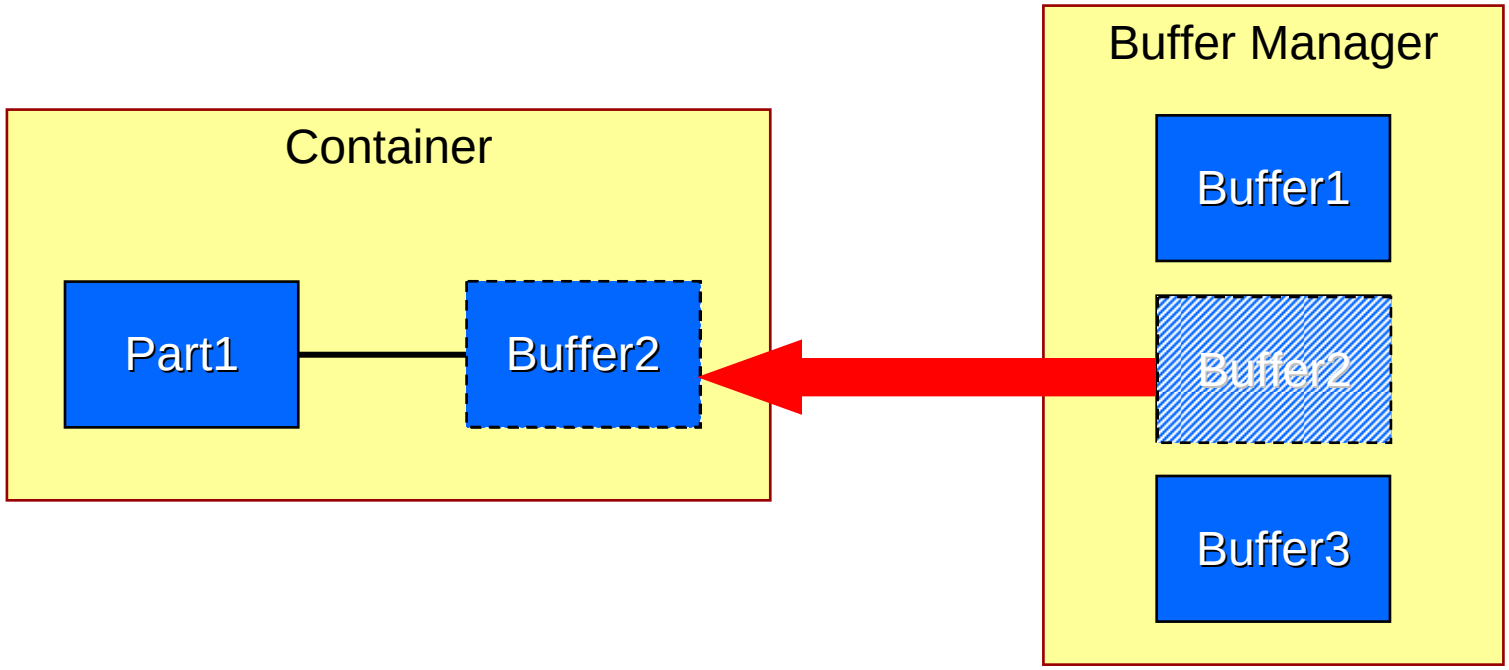
Structural Composition Architectural Pattern

- Part-whole relationship
  - Parts are used to implement the functionality of the container
  - Parts are hidden from other components (minimizes coupling)
  - Parts are owned by the container and cannot exist independently
  - Parts may be created dynamically after the container and destroyed before the container



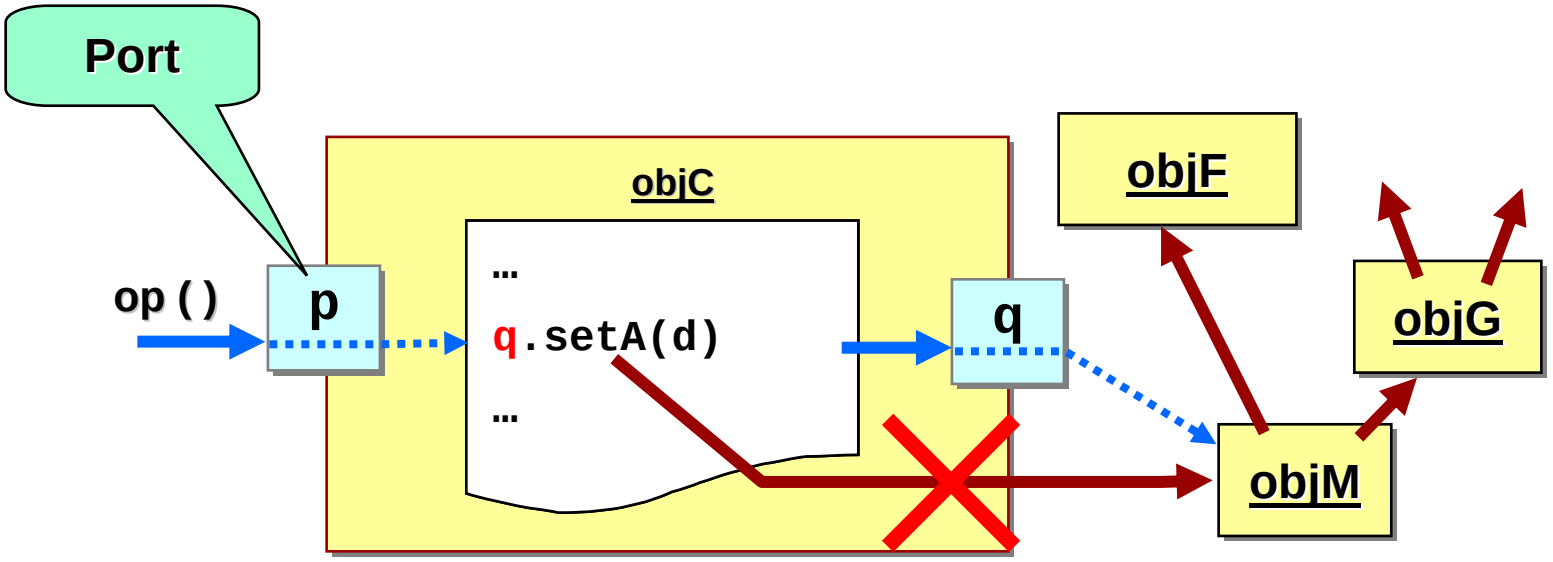
Structural Aggregation Architectural Pattern

- Like composition, except that parts are “borrowed”
  - Parts are actually placeholders for external parts owned by other containers



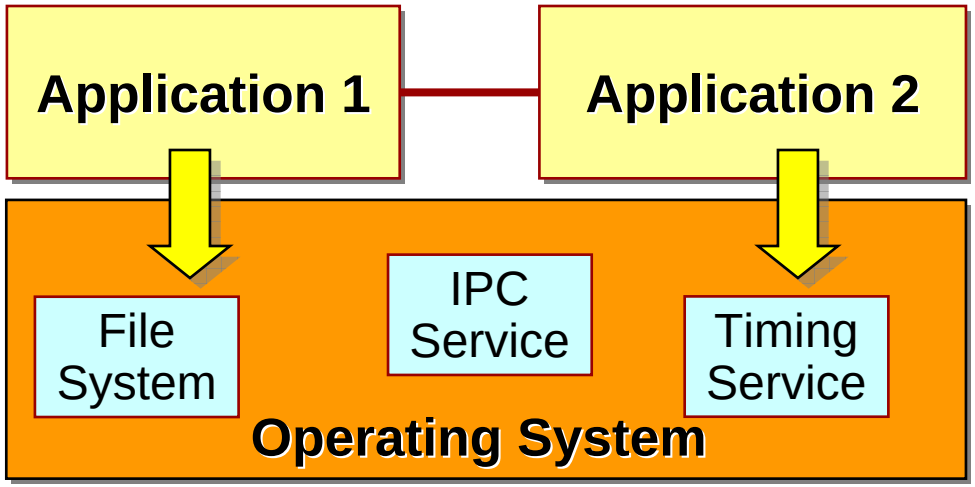
The Port Structural Pattern

- Distinct interaction points of an object for multiple, possibly simultaneous collaborations
- Ports allow an object to distinguish between different external collaborators without direct coupling to them



The Layering Structural Pattern

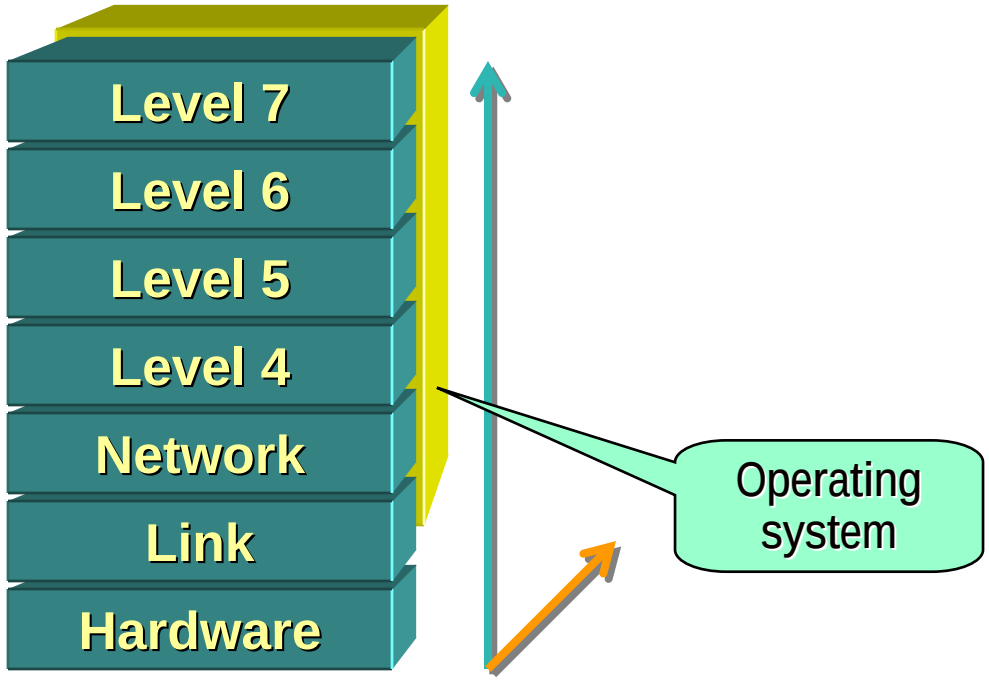
- Upper layers are existentially dependent on lower layer
  - But upper layer does not encapsulate the lower layers ⇒ different from composition
- Lower layer is independent of the upper layer entities



- The lower layer provides a set of shared implementation services
  - These cannot be encapsulated as parts by upper level components since they may be shared by more than one component

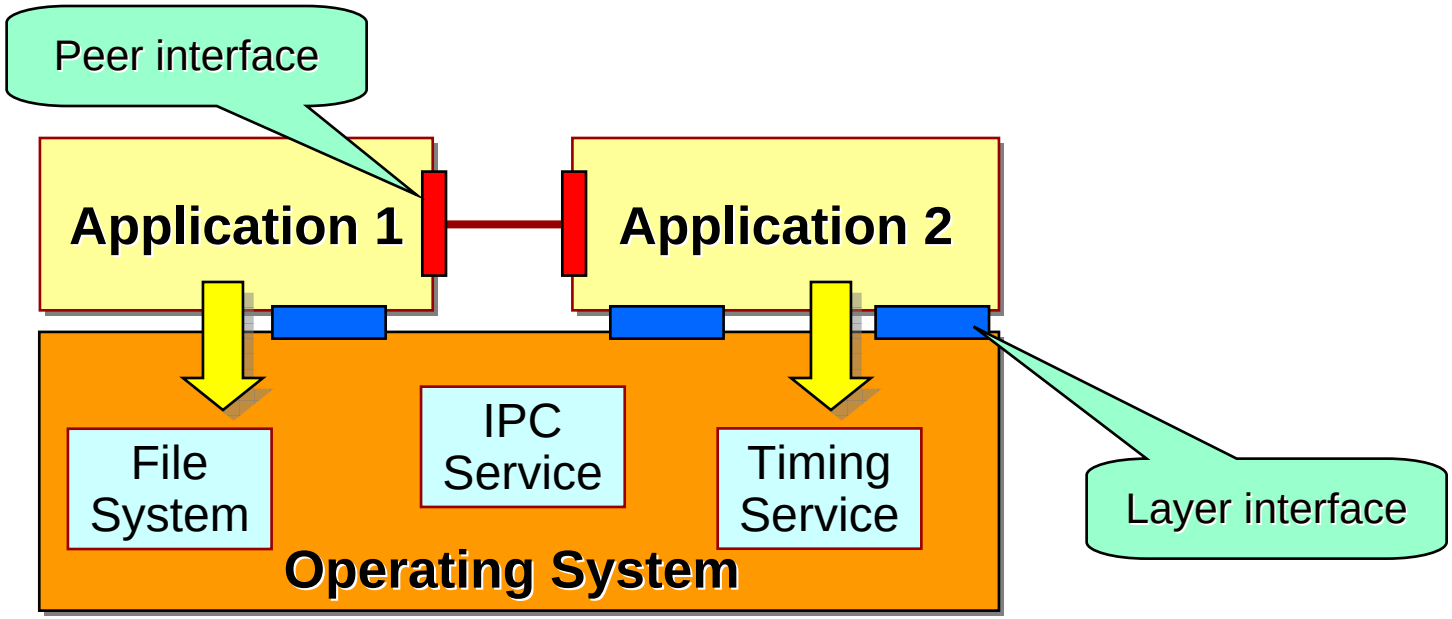
The Dimensions of Layering

- In complex systems, layering is a complex multidimensional relationship
  - e.g., 7-layer model of Open System Interconnection (OSI)



The Layering Structural Pattern (continued)

- Layering implies differentiating two kinds of component interfaces
  - *Implementation-independent peer interfaces*
  - *Implementation-specific layer interfaces* (service access points)

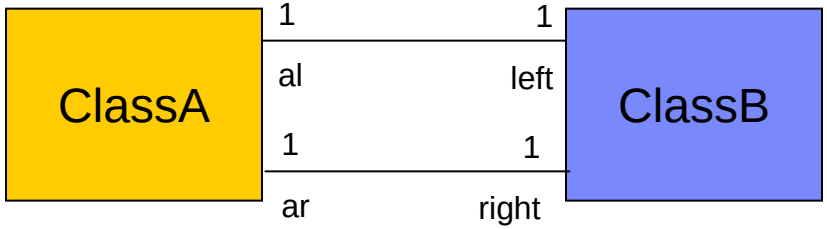


Outline

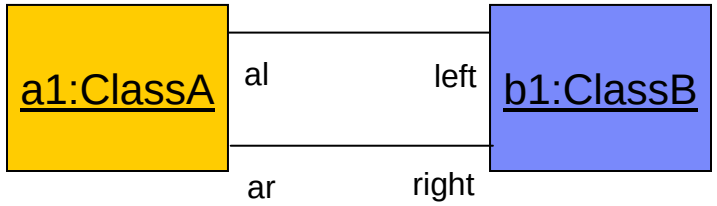
- On Software Architecture and MDD
- Requirements for Modeling Software Architectures
- Architectural Modeling Concepts in UML

Why Class Diagrams are Not Always Sufficient

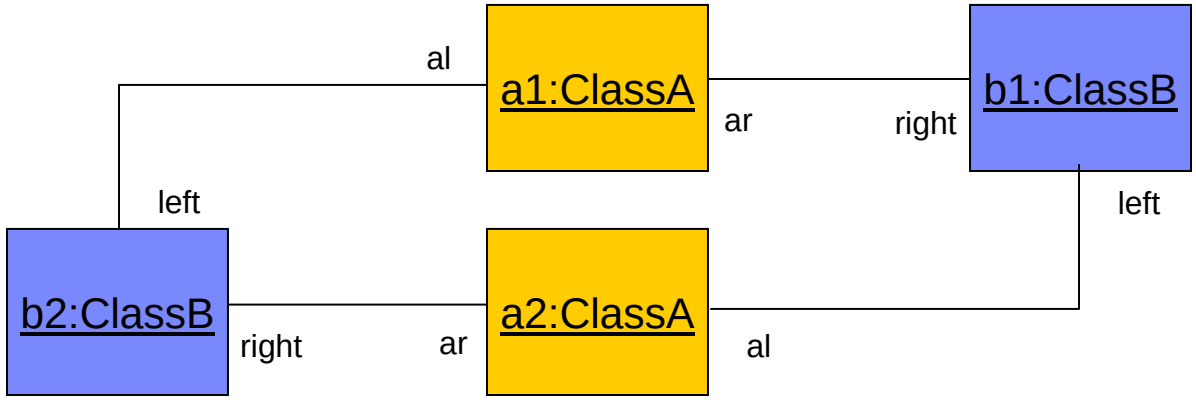
Class diagrams sometimes abstract away key architectural information!



(1)

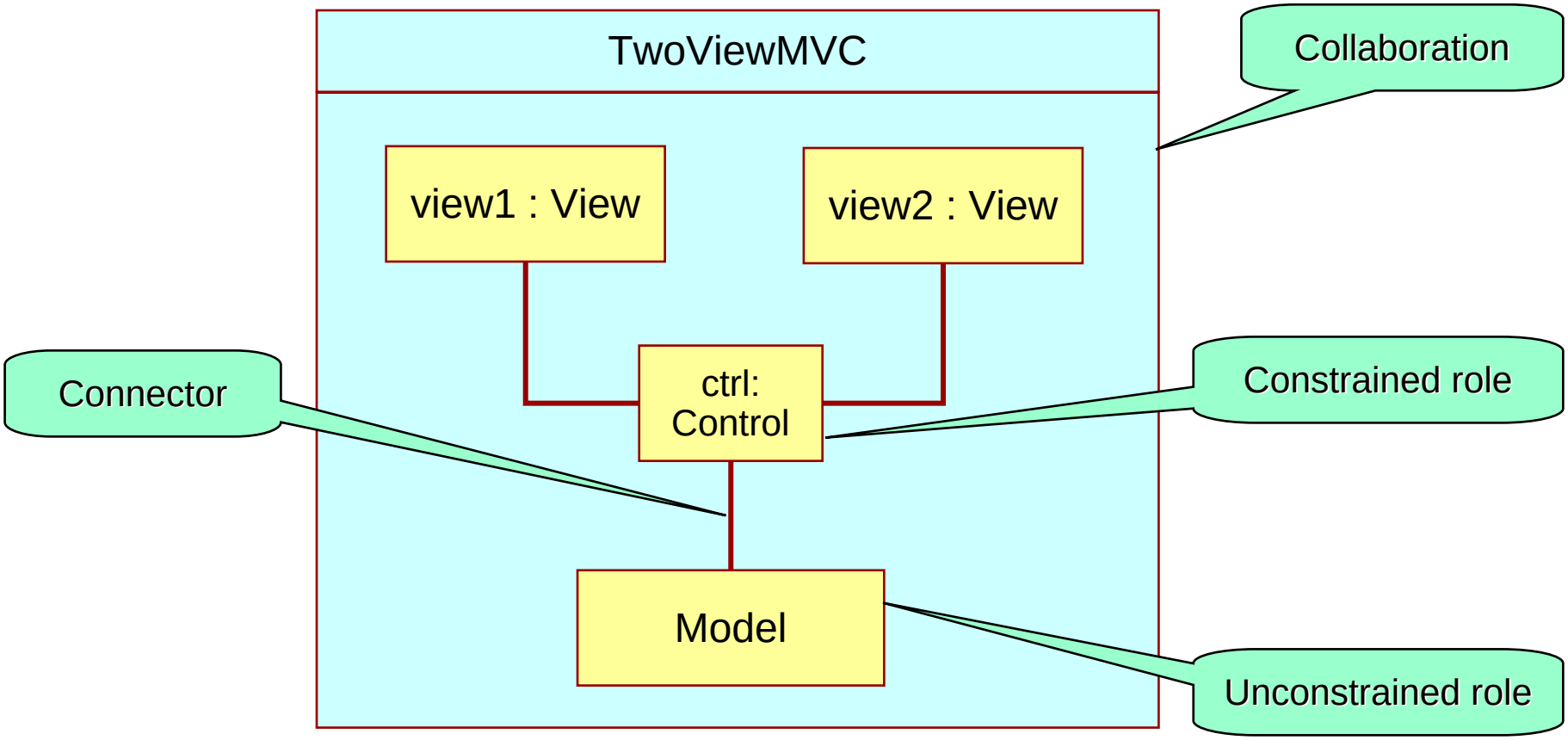


(2)



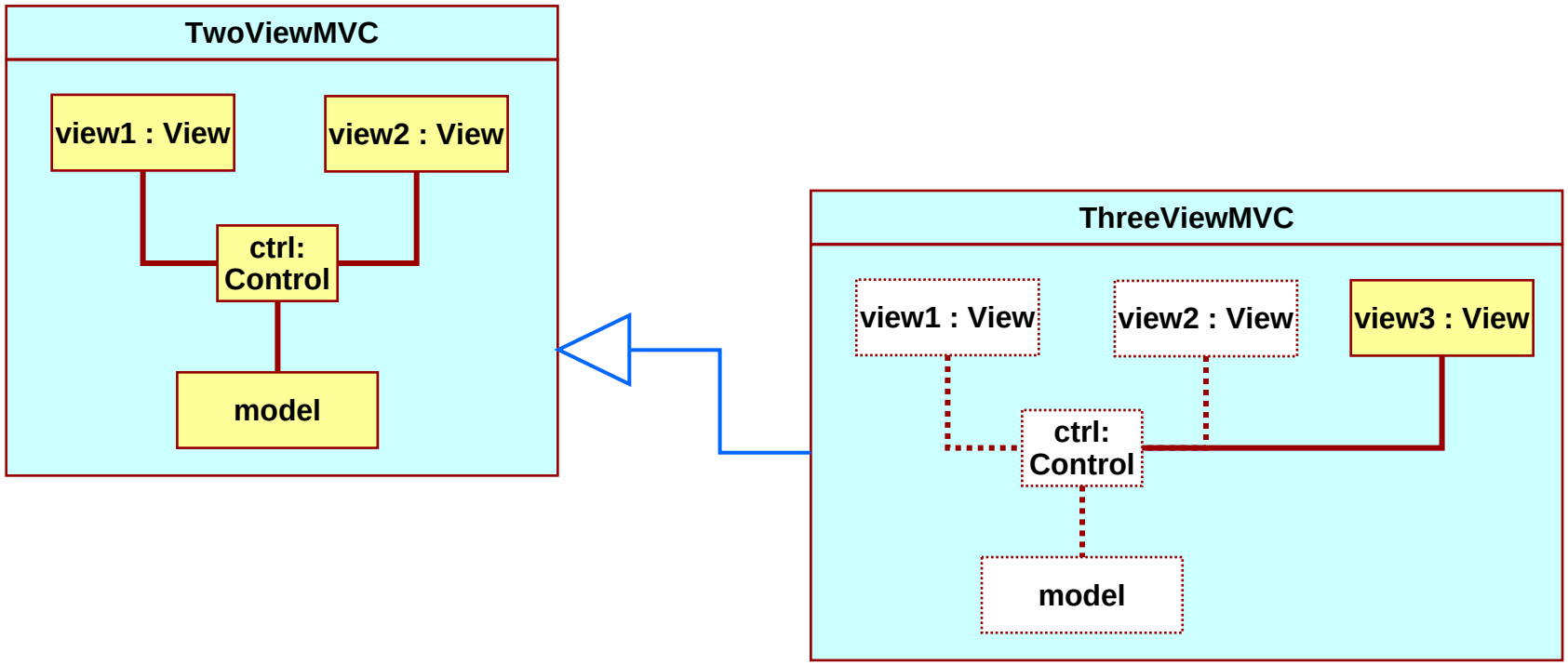
Collaborations in UML 2

- Describes a set of “roles” communicating across “connectors”
- A role can represent an instance or something more abstract



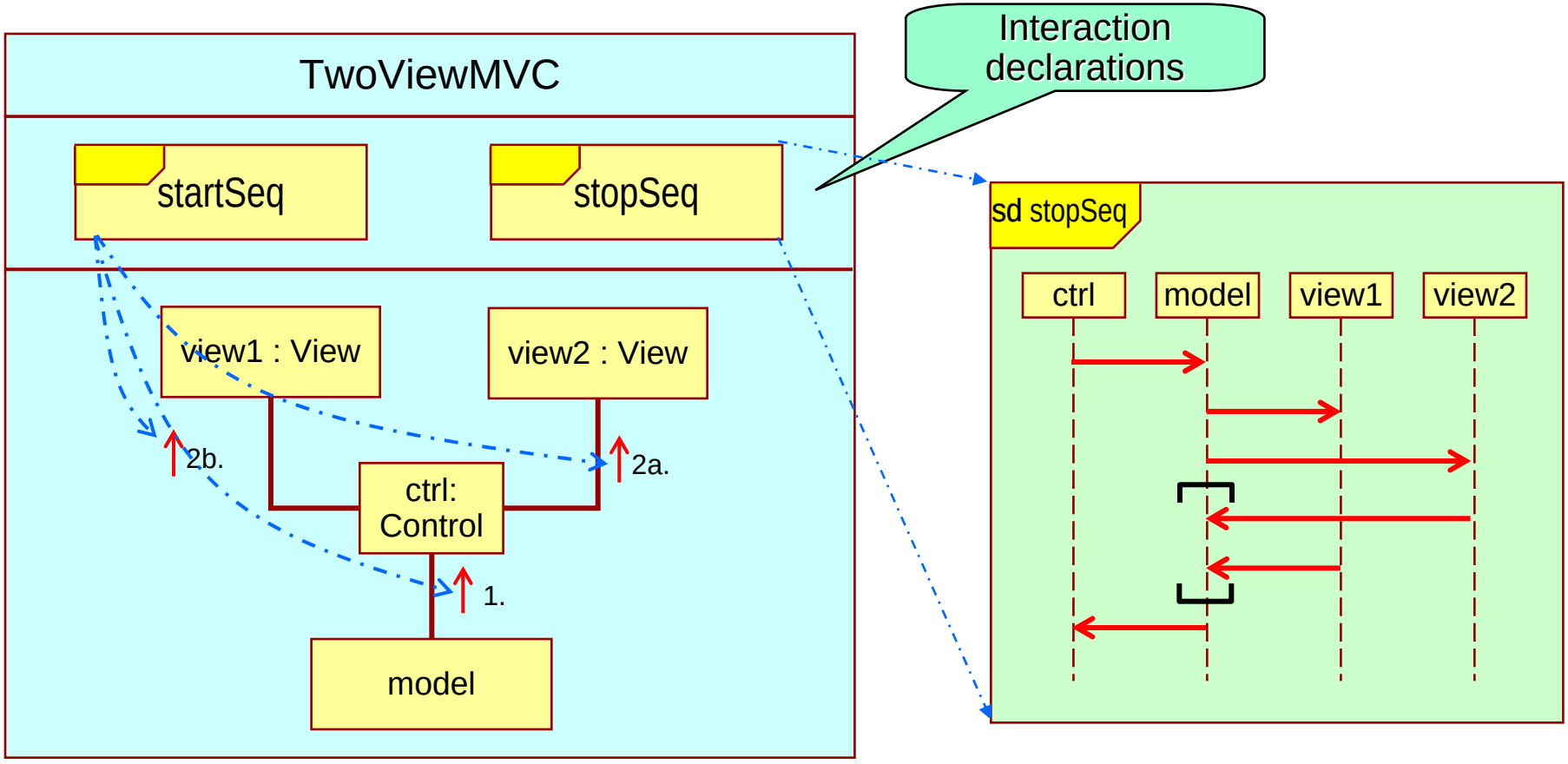
Collaborations in UML 2 (continued)

- Collaborations provide a direct means for modeling the "collaborating peers" architectural pattern
- Collaborations can be refined through inheritance
  - Possibility for defining generic architectural structures



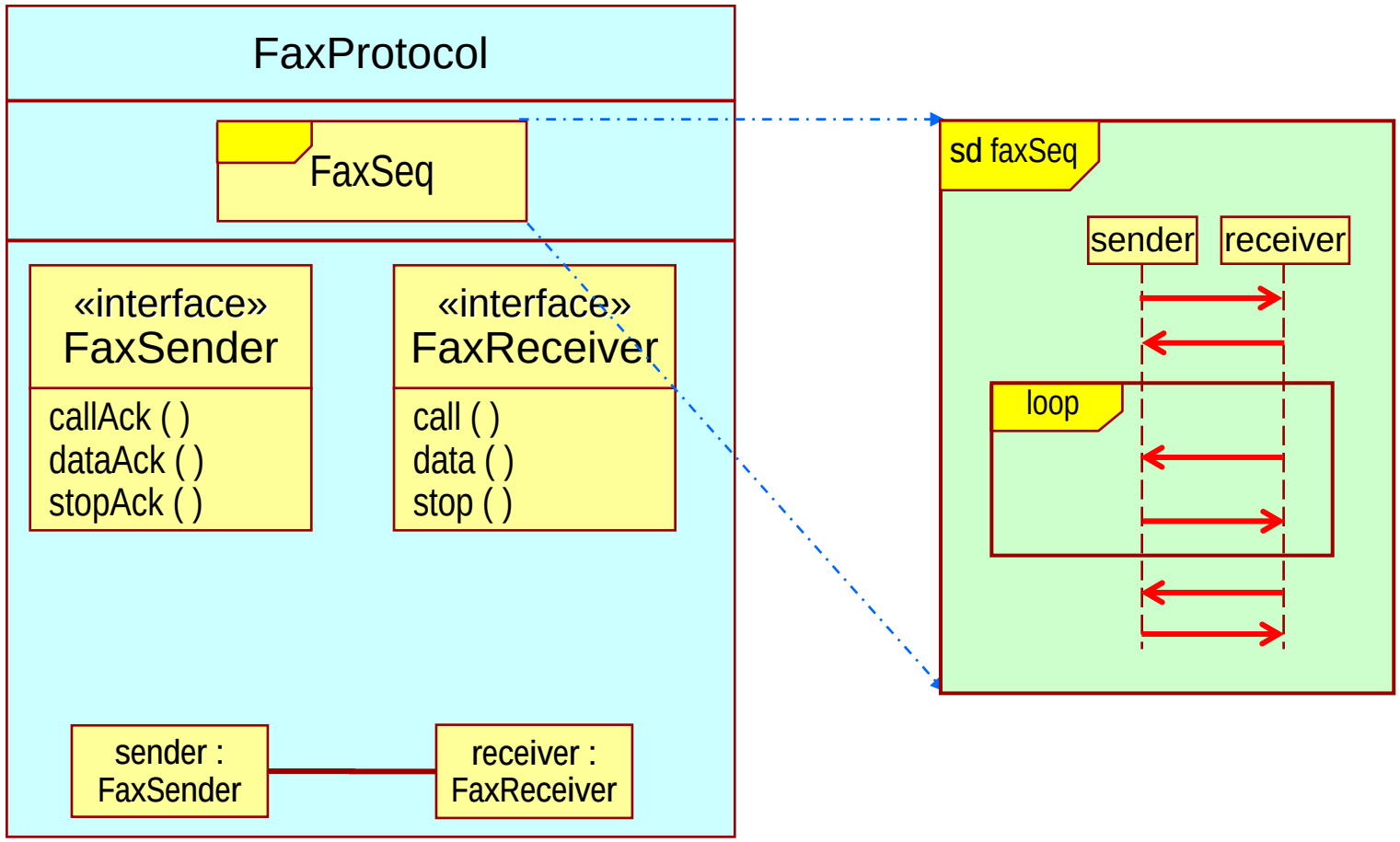
Collaborations and Behavior

- One or more behavior specs can be attached to a collaboration
  - To show interesting interaction sequences within the collaboration

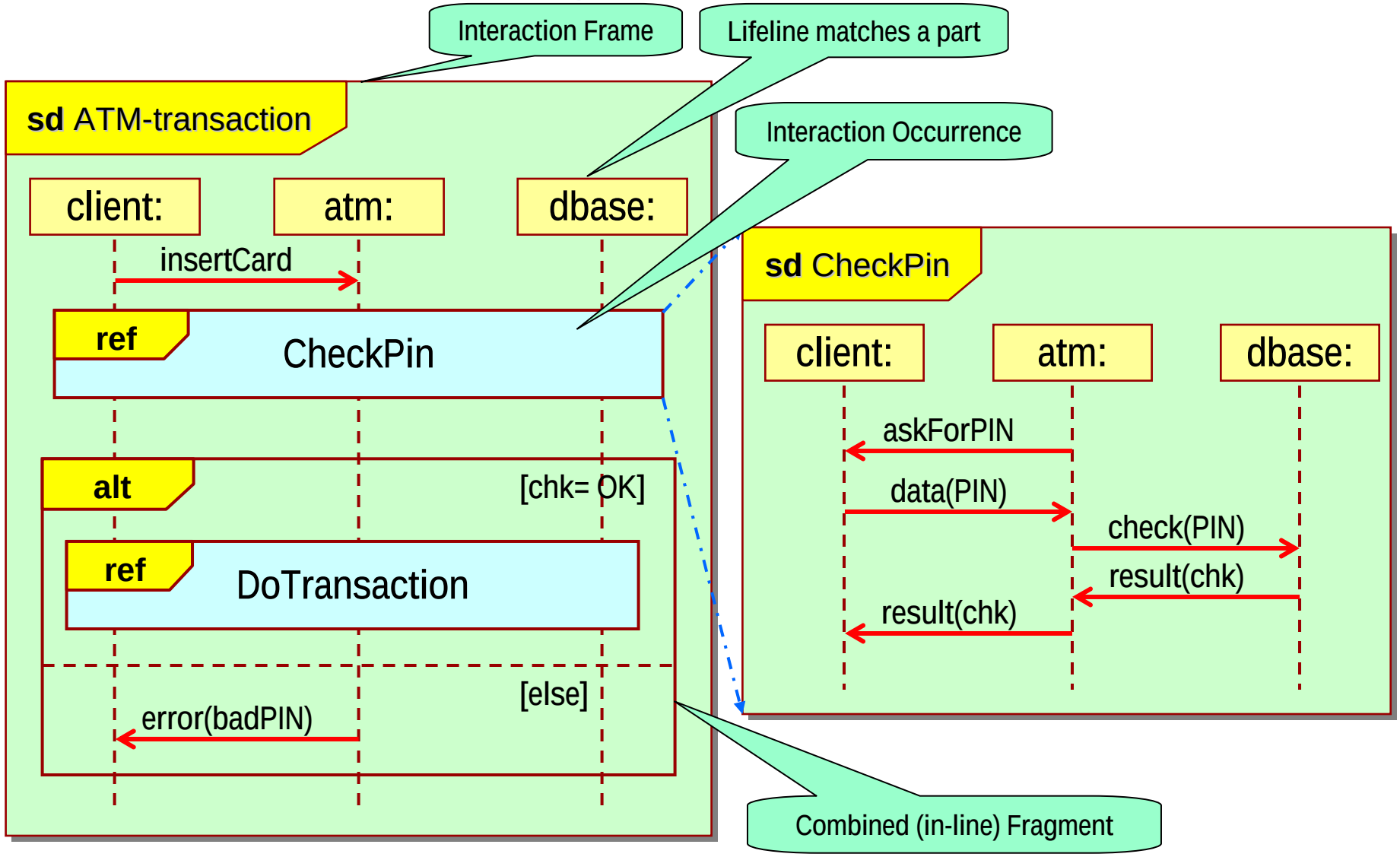


Modeling Protocols

- Usually declared between two or more interfaces
  - But, interfaces cannot be parts because they are not instantiable *per se*



UML 2 Interaction Diagrams

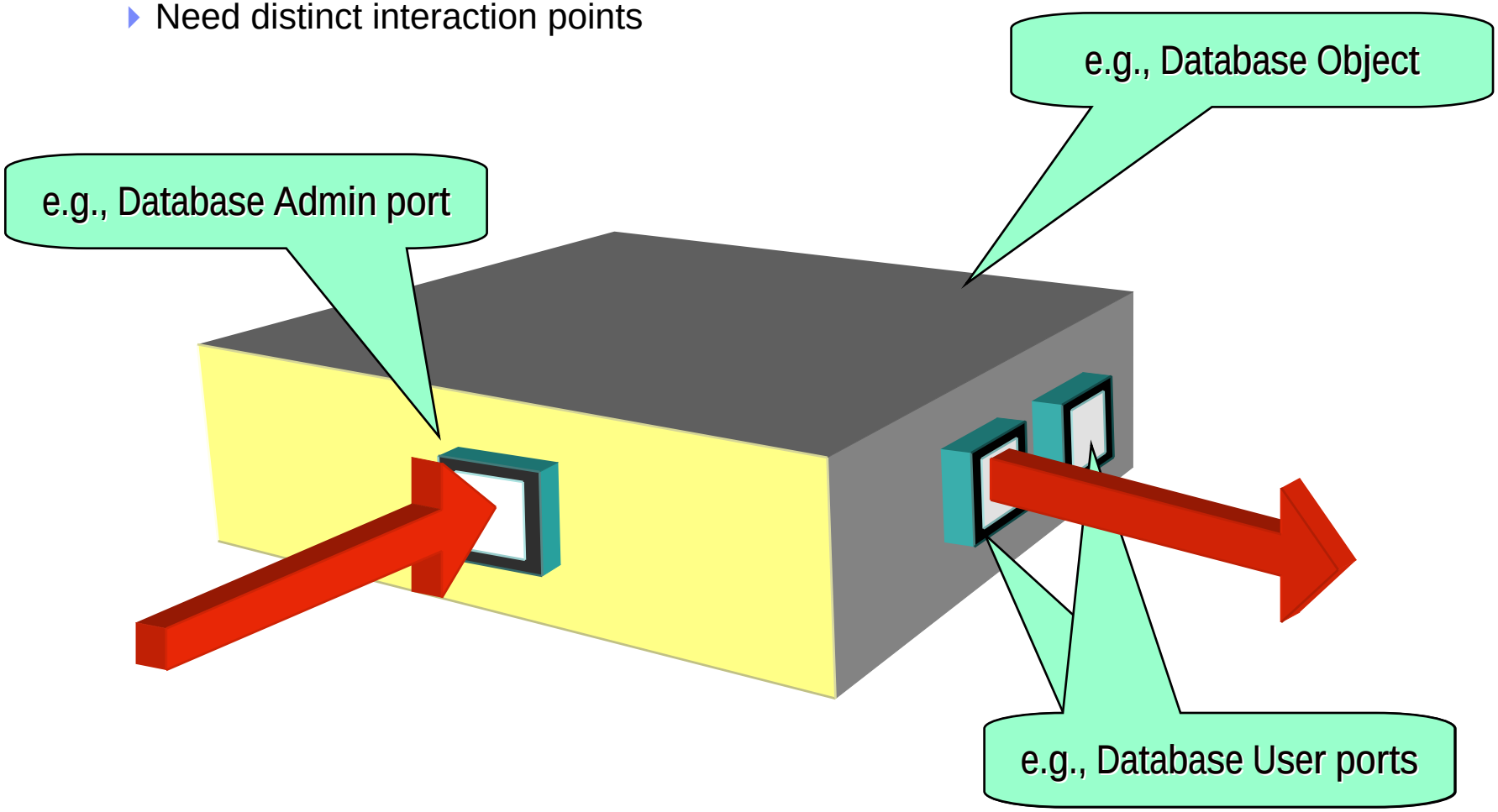


Structured Classes in UML 2

- This concept is closely related to the collaboration concept
- Classes with
  - An internal (collaboration) structure
  - An external structure consisting of Ports (optional)
- Heritage: various architectural description languages (ADLs)
  - UML-RT profile: Selic and Rumbaugh (1998)
  - ACME: Garlan et al.
  - SDL (ITU-T standard Z.100)

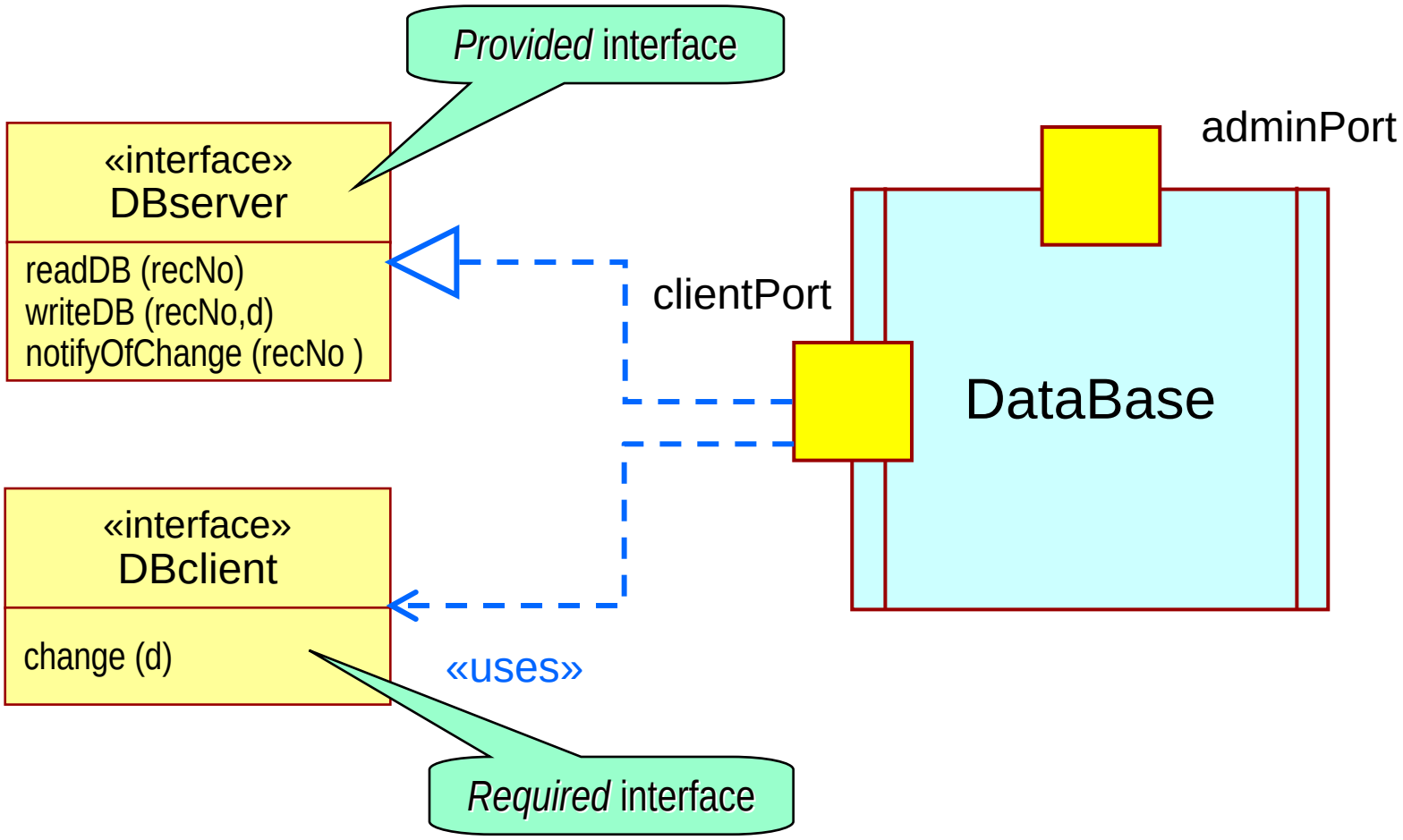
Structured Objects: External Structure (Ports)

- Complex (architectural) objects tend to collaborate with multiple clients
  - Need distinct interaction points

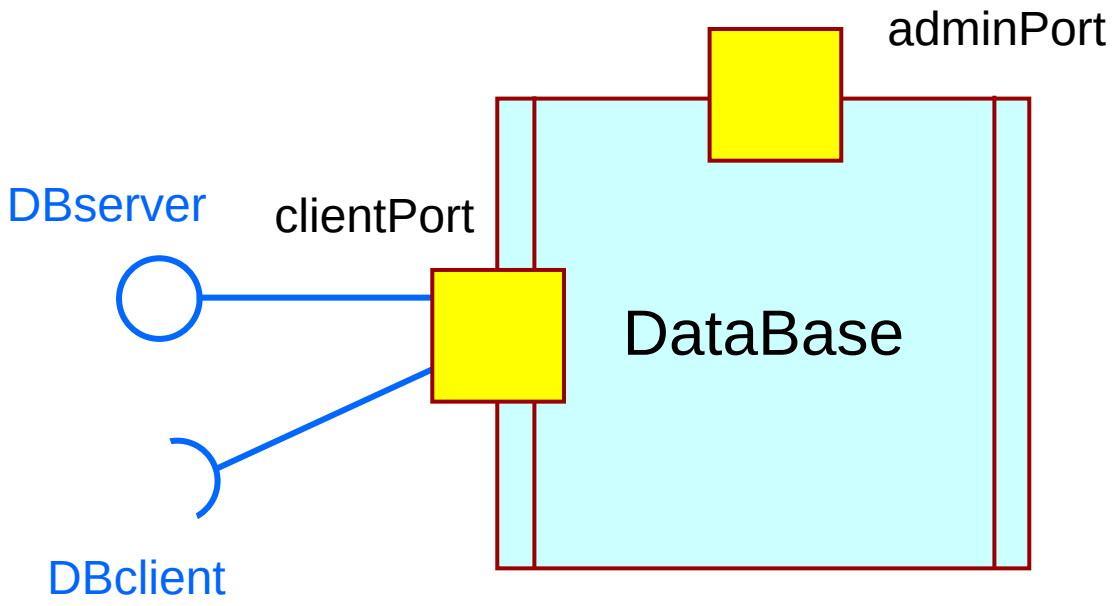


Ports and UML Interfaces

- In general, a port can interact in both directions

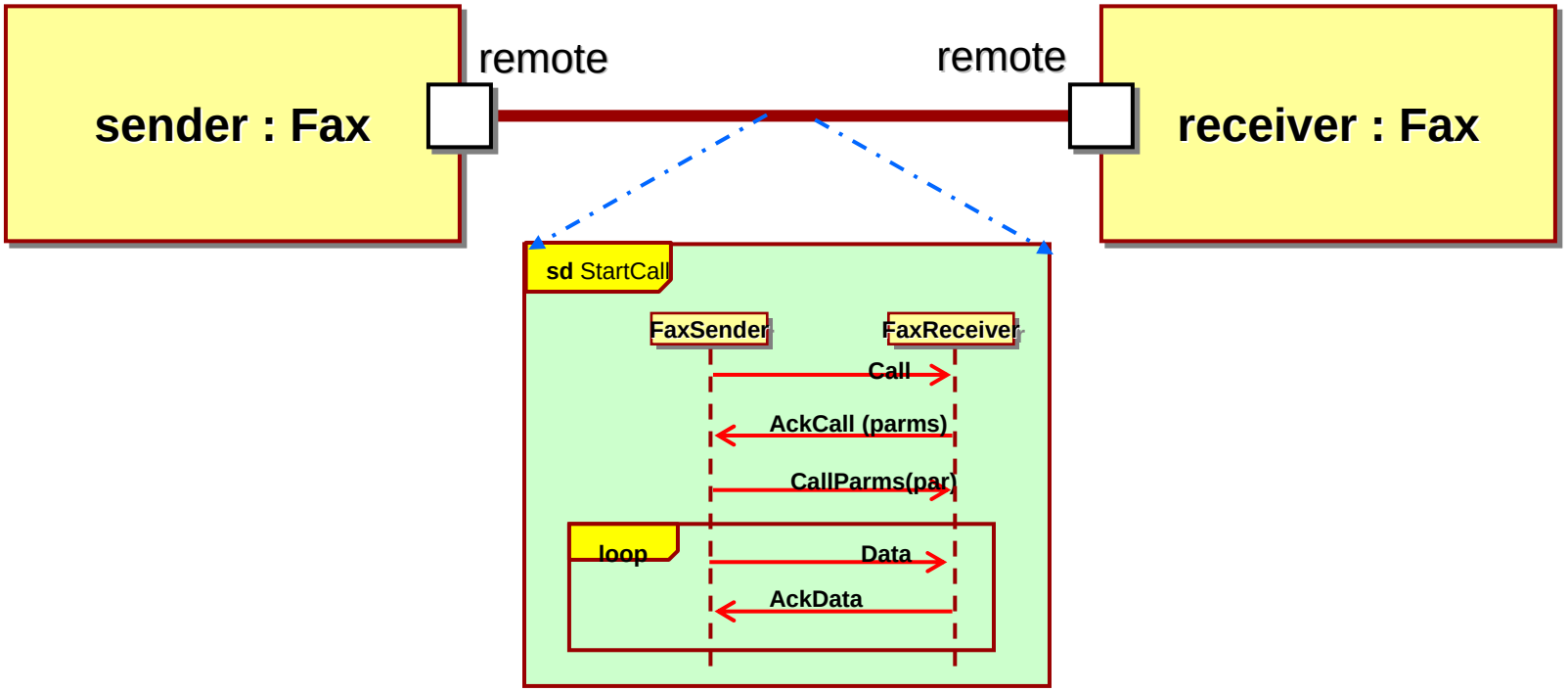


Shorthand Notation



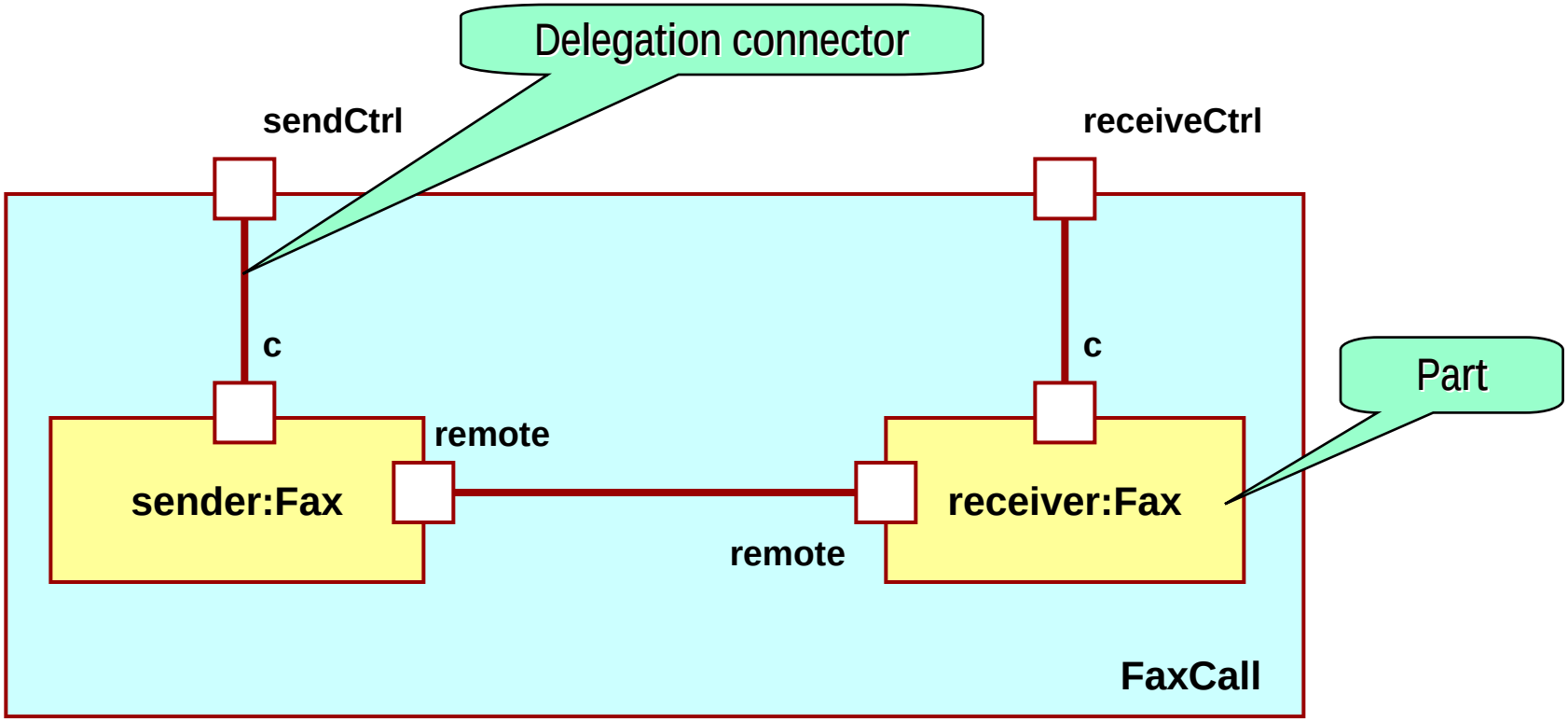
Assembling Structured Objects

- Ports can also be joined by connectors
- Connectors can be constrained to by a collaboration protocol
  - Static type checking for dynamic flow violations are possible
  - Eliminates a major source of “integration” errors



Structured Classes: Internal Structure

- Structured classes may have an internal structure of (structured class) parts and connectors
- Models both *composition* and *aggregation*

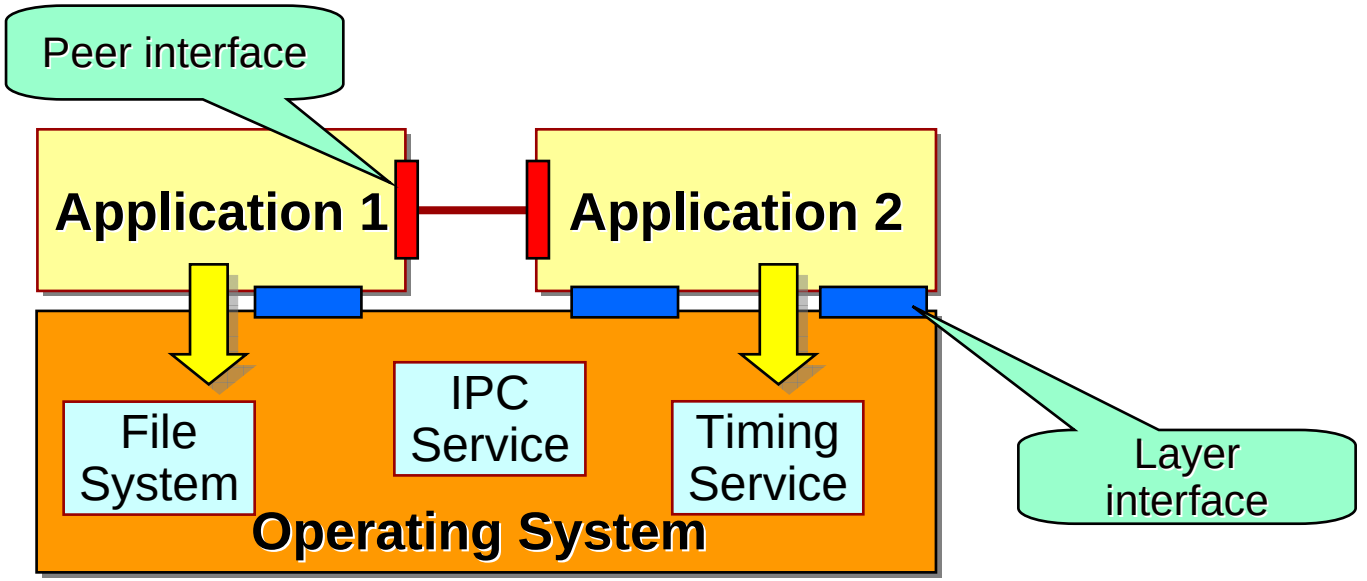


A word about UML components and UML subsystems

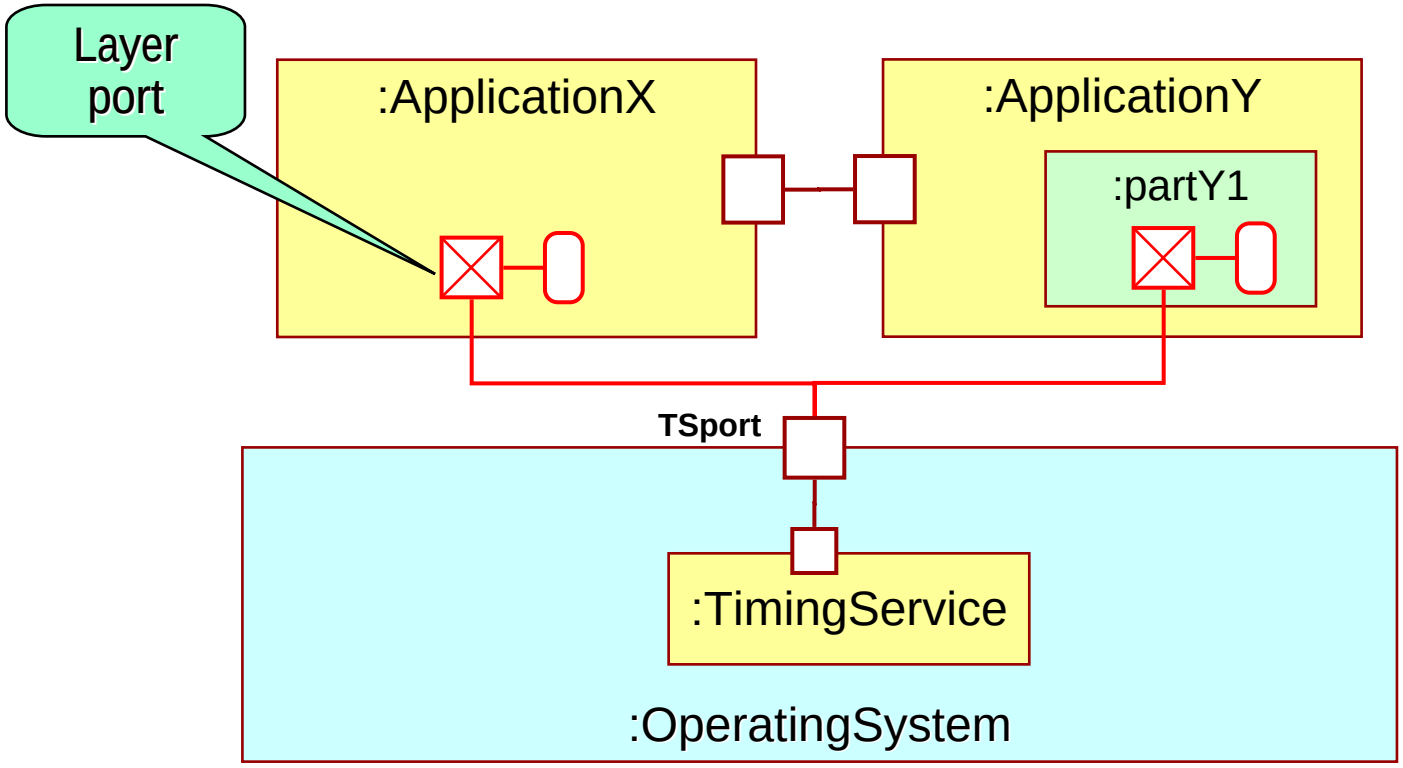
- UML Component = merely a special kind of structured class
  - Can act as a package ⇒ a combined design-time/run-time concept
  - Mixing of these two domains results in some complex semantics
  - Each use of the term “component” needs to be qualified to avoid confusion (run-time or design-time semantics?)
- UML Subsystem = a stereotype of UML Component
  - Its parts can be optionally tagged as being either *«implementation»* or *«specification»* elements
  - Inherits semantic complexity of UML Component concept
  - (NB: There are other ways of distinguishing implementation from specification elements)

Modeling Layers in UML 2

- Layering requires special “layer interfaces” = implementation-specific interfaces that access services of the layer(s) below
- UML 2 models layer interfaces using special *layer ports*
  - *(Ports whose “isService” meta-attribute set to false)*

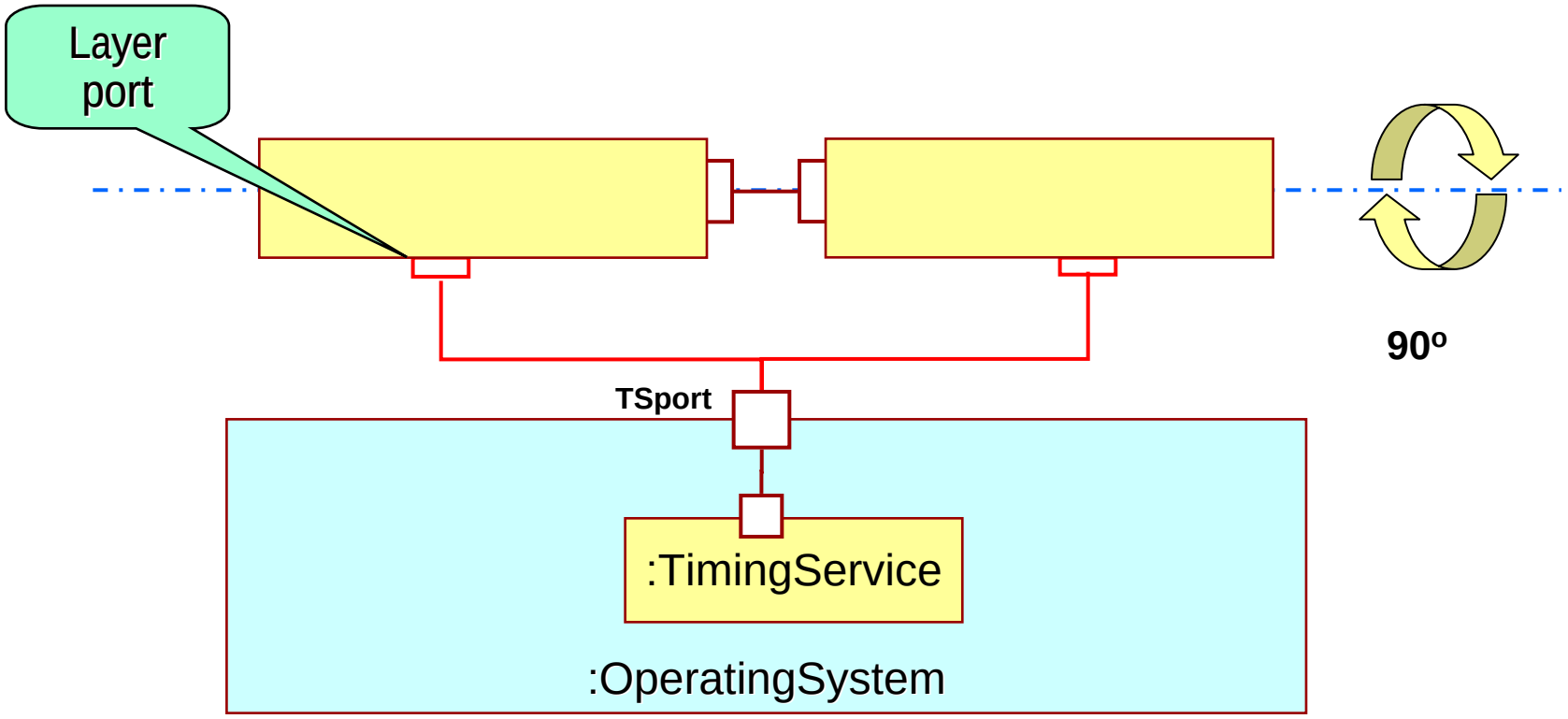


Modeling Layers (continued)



Modeling Layers (continued)

- The layer ports are in a different “dimension” than other ports



Summary

- Software architecture:
  - Defines the most important characteristics of a software design
  - Drives the construction
  - Determines its evolutionary potential

⇒ It is of critical importance to ensure that architectural decisions can be captured *clearly* and *accurately*

- UML 2 has a number of architectural modeling capabilities for most basic architectural design patterns
- Finally:
  - In combination with MDD techniques (e.g., automated code generation) it becomes much easier to ensure that *architectural intent is preserved* during implementation and subsequent system evolution

Rational tools for the software architect

- Rational Software Architect/Modeler/Developer (RSA/RSM/RSD)
  - Release 7.0 (coming) has extensive support for structured class concept
- Rational Rose RealTime
  - Supports an executable version of structured class concepts
  - But, based on a UML 1.4 profile

Questions

Thank You

Bran Selic
(bselic@ca.ibm.com)