Lecture

Object-Oriented Concepts in Distributed Systems

Summer 1998

Object-Oriented Concepts in Distributed Systems

Contents

D Distributed Systems . . . D.1

A Organizational Topics

- Lecturers
  - Dr.-Ing. Franz J. Hauck (hauck@informatik.uni-erlangen.de)
  - Dr.-Ing. Jürgen Kleinöder (kleinoeder@informatik.uni-erlangen.de)
  - Chair for Operating Systems (Informatik 4)

A.1 Lecture

- Object-Oriented Concepts in Distributed Systems
  - object-oriented programming, distributed systems, middleware (C++, Java, CORBA, Design Patterns, Frameworks)
  - for students in Computer Science and students of the int. masters program in computational engineering
- Time: Tuesday, 10:15 am – 11:45 am
- Location: Auditorium H4

A.1 Lecture (2)

- Language: English
- 10 minute summary of the previous lecture in German
- Video of the 1997 lecture
  "Objektorientierte Konzepte in der Betriebsprogrammierung"
  available via WWW
  (MPEG format)

A.1 Lecture (3)

- Script
  - no real script
  - slides
    - available through our WWW page (postscript or PDF files)
      (but we do not guarantee that they are available in time!)
    - we offer copies at the beginning of each lecture
      (coupon for the complete set: 5 DM)
  - script of the 1997 lecture
    "Objektorientierte Konzepte in der Betriebsprogrammierung"
    available through our WWW page (in german, contains lots of comments!)
  - for further informationen see our references to additional literature at the begginnning of each section

- URL of this lecture
  - http://www4.informatik.uni-erlangen.de/Lehre/SS/V_OODS/

A.1 Lecture (4)

- Credit & examination:
  - 4 hours:
    successful handling of the exercises
  - 2 hours:
    successful handling of the exercises or oral examination
  - with mark:
    oral examination

A.2 Exercise classes

- Teching assistents
  - Dipl.-Inf. Michael Golm
  - Dipl.-Inf. Uwe Rastofer
- Contents
  - Programming assignments on object-oriented programing in distributed systems
  - Topics: Java, C++, CORBA
- Enrollment for the exercises
  - "login: oods" at all CIP Workstations (room 01.155 or 02.151) of the Computer Science Dept.
- Time & Location
  - Wed. 4:15 pm – 5:45 pm, Auditorium H4

B Overview

B.1 Object-Oriented Programming

- Motivation
- Software design
- OOP — fundamental terms
  - Object
  - Method
  - Class
  - Inheritance
- Basic concepts
  - Abstraction
  - Hierarchy
  - Concurrency
  - Encapsulation
  - Typing
  - Persistence
  - Modularization
  - Polymorphism
- Objekt-orientierted analysis and design, design patterns

B.2 Distributed Systems

- Motivation
- Properties (advantages and disadvantages)
- Transparency and scalability
- Selected problems
- Distributed and object-oriented systems
  - Remote Invocation
  - Object Mobility

B.3 Programming Distributed Systems with CORBA

- Motivation
- Survey of the CORBA architecture
- Object Request Broker (ORB)
  - Interface Description Language (IDL)
  - Remote invocation
  - Dynamic invocation
  - Components of the ORB
- CORBA Services
- CORBA Facilities

B.4 Java — OO Language + Virtual Machine for Objects

- Architecture of the Java VM
- JavaOS
- Java processors
- Security concepts
- Java in distributed systems

B Overview (5)

B.5 Frameworks

- Concepts
- Examples

B.6 Research on Distributed Object-Oriented Systems

- Case studies

C Object-oriented Programming

C.1 Overview

- Motivation for the OO paradigm
- Software-design methods
- Basic terms of OO programming
- Basic concepts of the OO paradigm

C.2 References

ABC83. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshot and R. Morrison, “An Approach to Persistent Programming”, *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, 1983.

Boo94. Grady Booch, *Object-Oriented Analysis and Design (with Applications)*, Benjamin/ Cummings, Redwood (CA), 1994.

CoY91a. P. Coad, E. Yourdon. Object-Oriented Analysis. Prentice Hall, 1991.

Coa91b. P. Coad, E. Yourdon. Object-Oriented Design. Prentice Hall, 1991.

Cox86. Brad J. Cox. *Object Oriented Programming*. Addison Wesley, 1986.

CW85. Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol. 17, No. 4, Dec. 1985.

Jac92. I. Jacobson. Object-Oriented Software Enineering — A Use Case Driven Approach. Addison-Wesley, 1992.

MaM88. Ole Lehrmann Madsen, Birger Møller-Pedersen, "What object-oriented programming may be — an what it does not have to be", *ECOOP ’88 – European Conference on OO Programming*, pp. 1 - 20, S. Gjessing, K. Nygaard [Eds.]; Springer Verlag, Oslo, Norway, Aug. 1988.

Mey86. Bertrand Meyer, "Genericity versus Inheritance", *Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA ’87*, pp. 391 - 405, Portland (Oreg., USA), published as *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986.

Mey88. Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall Inc., Hemel Hempstead, Hertfordshire, 1988.

C.2 References (2)

Oes97. B. Oestereich. Objektorientierte Softwareentwicklung: Analyse und Design. Oldenbourg, 1997.

Rum91. J. Rumbaugh. Object-Oriented Modelling and Design. Prentice Hall, 1991.

Str91. Bjarne Stroustrup. *The C++ programming language*, 2. ed., Addison-Wesley, 1991.

Str93. Bjarne Stroustrup, "A History of C++", ACM SIGPLAN Notices, Vol. 28, No. 3, pp.271 - 297, Mar. 1993.

Weg87. Peter Wegner, “Dimensions of Object-Based Language Design”, *OOPSLA ’87 – Conference Proceedings,* pp. 1-6, San Diego (CA, USA), published as *SIGPLAN Notices*, Vol. 22, No. 12, Dec. 1987.

Weg90. Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming”, *ACM OOPS Messenger*, No. 1, pp. 8-84, Jul. 1990.

C.3 Motivation for the OO Paradigm

1 Goals

■ Increasing complexity of large software

- "industrial-strength" software [Boo94]
  - ➤ impossible for one developer to comprehend all details of its design
  - ➤ very long life span
  - ➤ many users depend on their proper functioning
  - ➤ many people responsible for maintenance and enhancement

➜ Software crisis

- Hardwareincreasingly capable
- Software becomes larger and larger
- Costs for maintenance and enhancement rise dramatically
- Not enough good software developers to create the software users need

1 Goals (2)

- Increase the productivity of programmers
  - Design patterns for repeatedly occurring problems
  - Reusage of existing software
  - Better extensibility of software by modularization and clear interfaces
  - Incremental development from small & simple to huge & complex systems
  - Better control over complexity and costs of software maintenance

- Shift from the needs of the machine to abstractions of the problem domain
  - Better understanding of the problem
  - Terminology of the problem domain is reflected in the software solution
    - ➤ better understanding of the solution

1 Classification [Boo94]

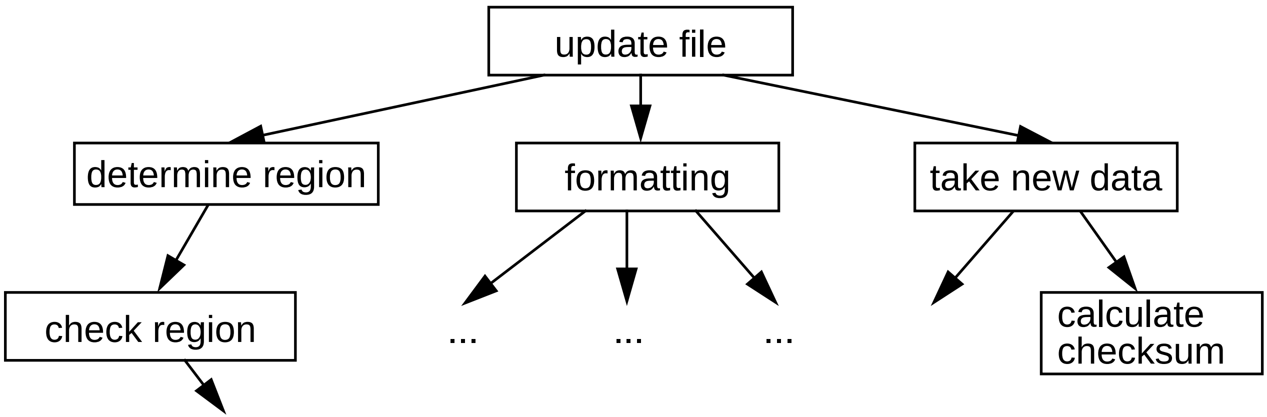
- Top-down structured design (composite design)
- Data-driven design
- Object-oriented design

2 Top-Down Structured Design (Composite Design)

- Units of decomposition: Subroutine
- **Algorithmic decomposition**
- Not suitable for structuring today's large and complex software systems
- Top-down structured design cannot describe:
  – data abstraction & information hiding
  – concurrency
- Problems arise when applications are very complex or when object-oriented languages have to be used
- Widely used technique

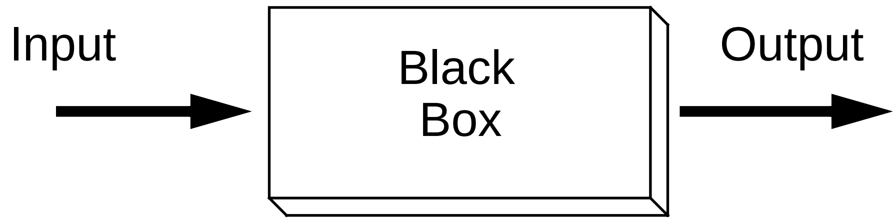
2 Top-Down Structured Design (2) (Composite Design)

- Example:



3 Data-driven Design

- Fundamental concepts from Jackson et al.
- Structure of software is based on mapping inputs to outputs
  - ➔ Application mainly in information management
- Problems with time-critical events



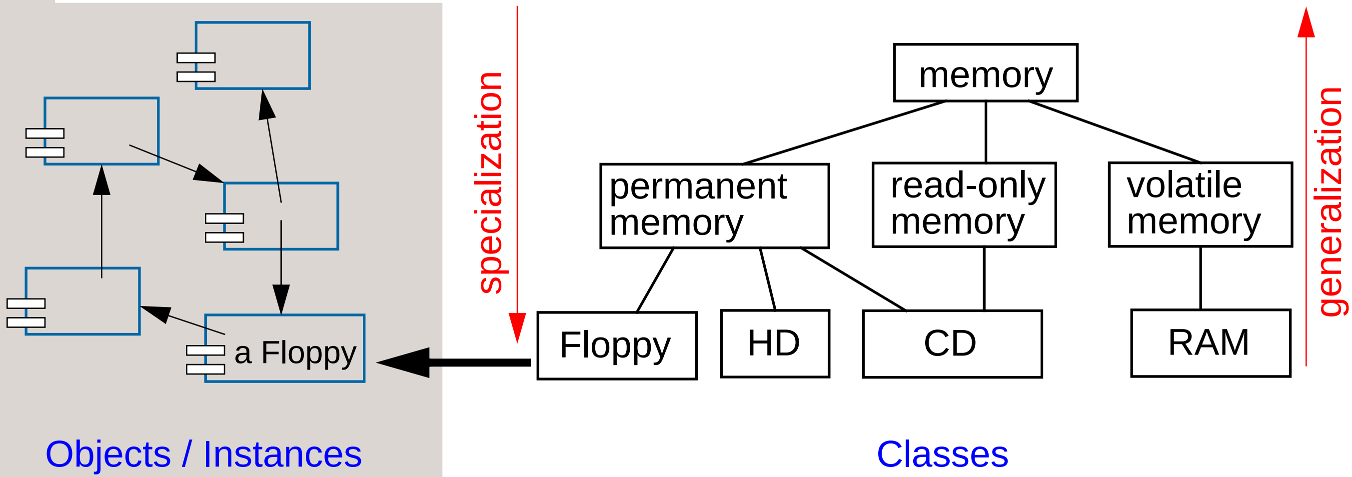
4 Object-oriented Design

Bertrand Meyer:[Mey88]

Computing systems perform certain actions on certain objects;
to obtain flexible and reusable systems,
it is better to base the structure of software
on the objects than on the actions.

… Object-oriented Design (2)

- Software system is modeled as a collection of cooperating objects
- Each object is
  an instance of a class
  in a hierarchy of classes
- Example of a class hierarchy:



4 … Object-oriented Design (3)

- Concepts reflected in the structure of modern programming languages
  - ➤ Smalltalk
  - ➤ C++
  - ➤ Eiffel
  - ➤ Java
  - ➤ Ada

- General basis: object-oriented decomposition

- Advantages:
  - + Reusage of common mechanisms
    - ➔ software becomes smaller
  - + Modifications and improvements of the software become easier
  - + Results are less complex
  - + Better understanding of the principal's ideas

C.5 The Evolution of the Object Model

1 Generations of Programming Languages

- First generation (1954 - 1958)
  - Mathematical Expressions (FORTRAN I, ALGOL 58)
- Second generation (1959 - 1961)
  - Subroutines, separate compilation (FORTRAN II)
  - Block structure, data types (ALGOL 60)
  - Data description, file handling (COBOL)
  - List processing, pointers, garbage collection (Lisp)
- Third generation (1962 - 1970)
  - various successors to ALGOL 60 (ALOGOL 68, Pascal)
  - Classes, data abstraction (Simula)

1 Generations of Programming Languages (2)

- *The Generation Gap* (1970 - 1980)

  many different languages were invented,
  but few endured …

  - C
  - Modula
  - Ada

- Object-oriented programming languages (1980 - today)
  - Smalltalk (successor to Simula)
  - C++ (evolved from C and Simula)
  - Eiffel (evolved from Simula and Ada)
  - Java

C.6 Object-oriented Programming

1 Definition (Grady Booch)

OOP is a method of implementation in which programs are organized as

> **cooperative collections of objects**,

each of which represents an

> **instance of some class**,

and whose classes are all members of a hierarchy of classes united via

> **inheritance relationships**.

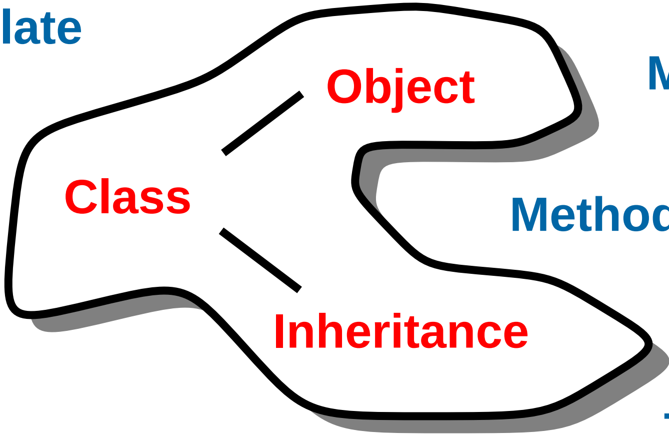
2 Basic Terms

Polymorphism

Destructor

Template

Message



Method

Type

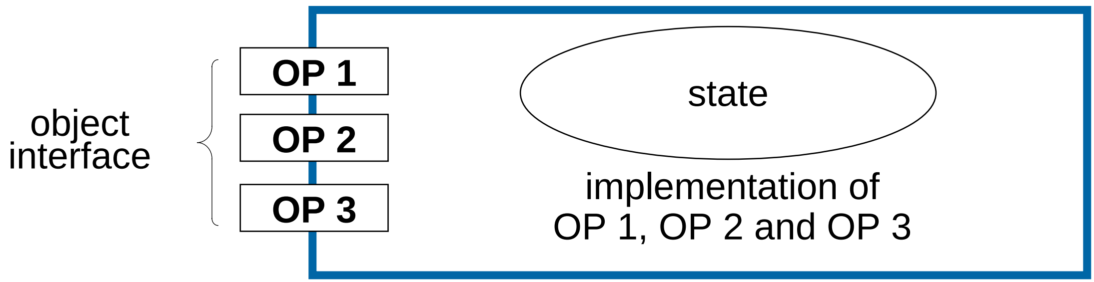
Constructor

Overloading

3 Objects & Methods

- Software developer's view:
  - ◆ an object is a "thing" from the problem domain
    - ➤ has a state
    - ➤ has behavior
    - ➤ has a unique identity

- Program-technical point of view:
  - ➤ an encapsulated unit of data and functions that operate on this data
  - ➤ an object has a clear interface (operations = **methods**)

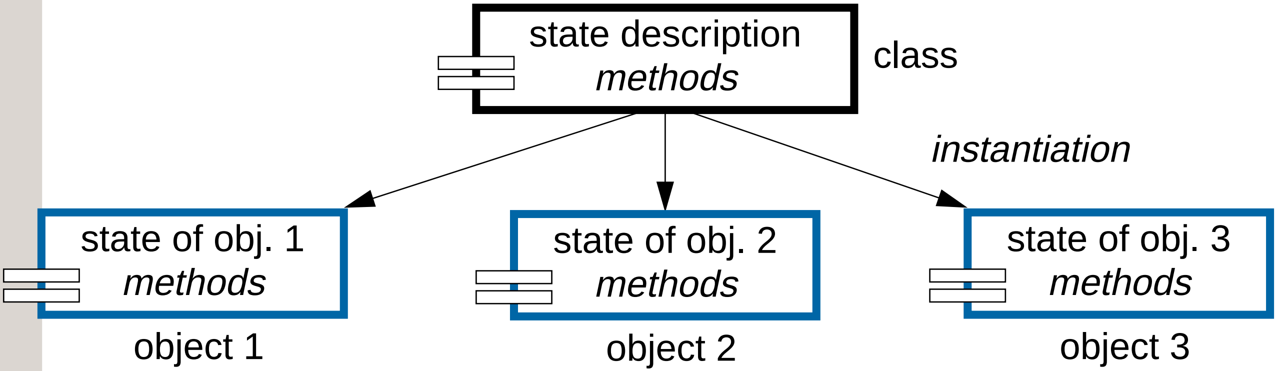


➔ **object-based programming languages** [Weg87]

4 Classes

- Software developer's view:
  - ◆ a class is a set of objects with common structure and common behavior

- Program-technical point of view:
  - ◆ a class is a template for objects
    - ➤ each object is an instance of a class
    - ➤ object creation = *instantiation*



➔ **class-based programming languages**
= objects & classes

5 Objects and Classes in C++

- Class declaration similar to a structure declaration in C
- Access to members of an object (instance variables and methods) with the operators `.` or `->`, like the access to structure components
- Example:

```
// Class counter
class Counter
{
  private:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};
```

6 Methods in C++

- Definition within a class declaration:
  - ➤ method is handled as ***inline*** function
- Definition separate from the class declaration
  - ➤ assignment to class with the ***scope*** operator `::`
  - ➤ method invocations are handled like normal function calls
- Example:

```
class Counter {
  private:
    int value;
  public:
    void incr(); void decr(); int get_value();
};

void Counter::incr()           { value++; }
void Counter::decr()           { value--; }
int Counter::get_value()       { return value; }
```

7 Instantiation in C++

- Instantiation of Objects either
  - ➤ statically at compile time, or
  - ➤ dynamically during run time

★ Static Instantiation

- By object definition
- Example:

```
void main()
{
  Counter c1;            // object c1 of class Counter
  Counter *pc1;          // pointer to an object of class Counter
  ...
}
```

7 Instantiation in C++ (2)

★ Dynamic Instantiation

- C++ operators **new** and **delete**
- Example:

```
class Counter
{ ... };

void main()
{
  Counter c1;                // create object c1 statically
  Counter *pc1;              // pointer to an object of class Counter
  ...
  pc1 = new Counter;
  pc1->incr();
  c1.incr();
  ...
  delete pc1;
  ...
}
```

7 Instantiation in C++ (3)

★ Constructor

➜ Method for the initialization of objects

- method name = class name
  - ➤ method is automatically invoked during instantiation

- Example:

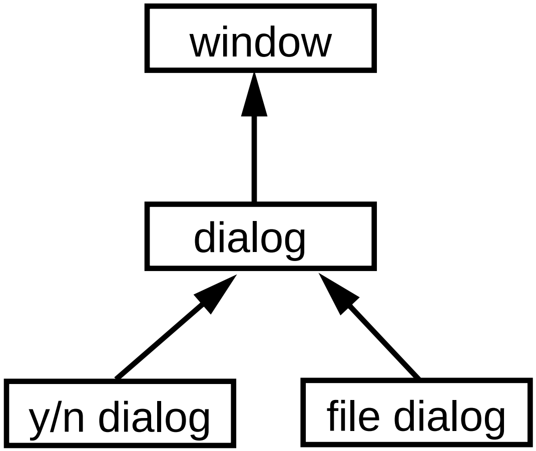
```
class Counter {
  private:
    int value;
  public:
    Counter(int c){ value = c; }         // constructor
    void incr()   { value++; }
   ...
};
...
Counter c1(20);        // create c1, initialize value to 20
cp = new Counter(30);
```

★ Essential Differences to C++

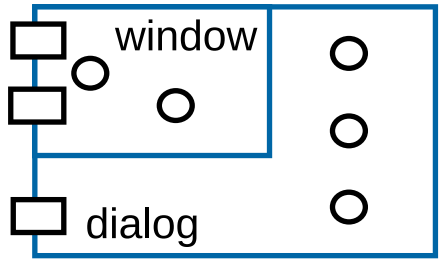
- No static instantiation
- Dynamic instantiation → only references (pointers) to objects
  - ◆ access to object components through object reference and operator .
- No need to delete objects explicitly
  - ◆ automatic garbage collection
- Methods are implemented always in the class declaration
  - ◆ but no in-line mechanism
- No pointer arithmetic

9 Inheritance

- Relationship among classes where
  one class shares the structure and/or behavior
  defined in another class / other classes



inheritence hierarchy

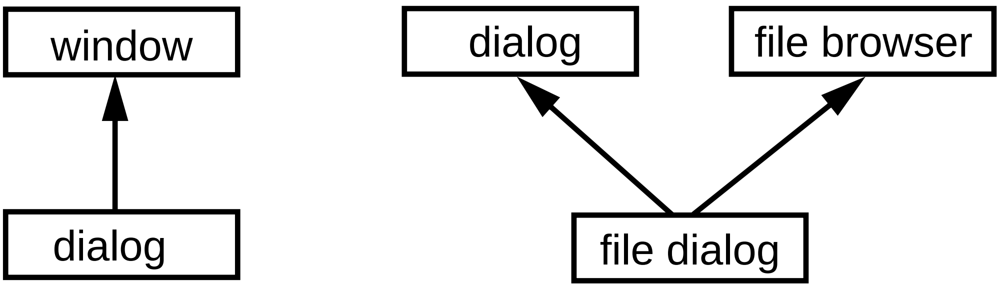


instance of an
inheriting class

9 Inheritance (2)

★ **Terms**

- **Superclass / base class:** class from which another class inherits
- **Subclass:** class which inherits from other class(es)
- **Single inheritance:** subclass has exactly one superclass
- **Multiple inheritance:** subclass has several superclasses



single inheritence

multiple inheritence

9 Inheritance (3)

★ Software developer's view

- Specialization / generalization of classes
- Common aspects of classes are collected in a superclass
- Hierarchy of abstractions:
  - from more general classes to specialized classes and vice versa
- Documentation of the relationship between classes

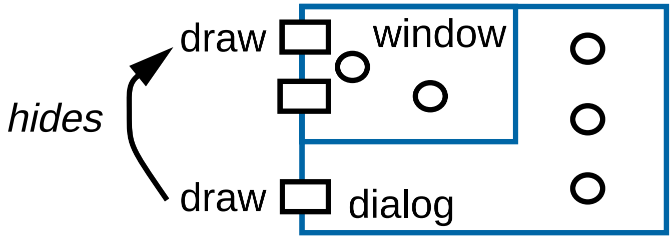
9 Inheritance (4)

★ Program-technical point of view

- Extension of an existing class implementations
  - ➤ additional methods
  - ➤ additional data
- Code reusage:
  no reimplementation of inherited data and methods necessary
- Reimplementation of a method is *possible*, if the method of the superclass is not appropriate for the subclass
- Methods of the superclass can be invoked at an object of the subclass
- Modifications of a superclass effect all subclasses
  (central maintenance)

Reimplementation

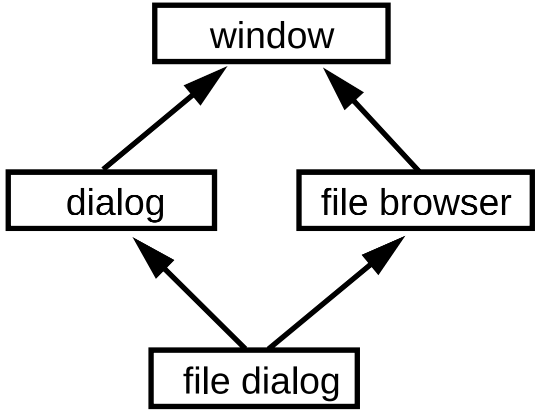
- Reimplementation of a method:
  - ➤ hides the method of the superclass



  - ➤ default behavior: invocation of the subclasses’ method
  - ➤ invocation of the reimplemented method of the superclass?

9 Inheritance (6)

★ Multiple Inheritance



- Problems:
  - ➤ naming conflicts of variables or methods of the different superclasses
  - ➤ inheritance of the same superclass through different paths

- Application:
  - ➤ less important for code reusage
  - ➤ very important to describe type conformance (see section about typing)

10 Inheritance in C++

- Subclass inherits variables and methods of the superclass
- Subclass may modify superclass
  - additional methods and variables
  - modified methods
- Methods of the subclass may access *public* and *protected* components of the superclass
  - public superclass
    - ➔ the *interface* of the superclass is inherited
  - private superclass
    - ➔ the *interface* of the superclass is *not* inherited
    - ➔ objects of the subclass are not type-conform
- *private* data and methods of the superclass are not visible for methods of the subclass

10 Inheritance in C++ (2)

★ Example (1)

```
// Class counter
class Counter
{
  protected:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};

// Subclass resettable counter
class RCounter : public Counter
{
  private:
    int initial;
  public:
    RCounter(int v)      { initial = v; value = v; }
    void reset()         { value = initial; }
};
```

10 Inheritance in C++ (3)

★ Example (2)

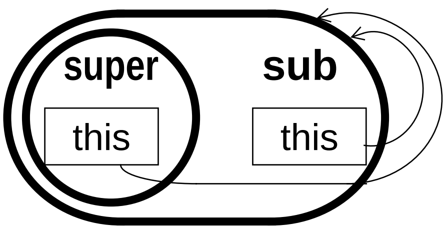
```
// Class window
class Window
{
  protected:
    int x, y, width, height;
  public:
    virtual void init(int x, int y, int w, int h){ initialize }
    virtual void move(int x, int y) { move window }
    virtual void display() { display window }
    virtual void delete() { remove window }
};

// Subclass bordered window
class BorderedWindow: public Window
{
  public:
    virtual void display() { display bordered window }
    virtual void change_width(int x) { change width }
    virtual void change_hight(int y) { change hight }
};
```

11 Dynamic Binding

- Decision which method to execute at run time (dynamic)

```
Window w = new BorderedWindow();
w->display();
```

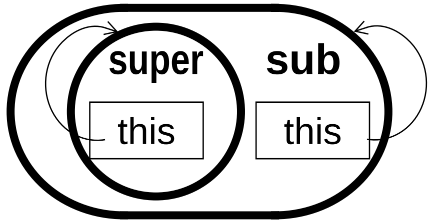
- This is also true if an object invokes a method at itself!
  - Example:
    - **`move()`** finally calls **`display()`** to redraw the window
    - **`BorderedWindow`** inherits **`move()`** from **`Window`**
    - invoking **`move()`** at an instance of **`BorderedWindow`** finally calls **`display()`** of **`BorderedWindow`**



the pointer *this* always references the "whole object" and not just the part of the superclass

11 Dynamic Binding (2)

- Without dynamic binding "true inheritance" is not possible
  - ➔ self reference (pointer *this*) is not adjusted correctly



★ **Static Binding**
Decision which implementation of a method is taken at compile time (depending on the type of the pointer)

- In C++ only "virtual" methods are bound dynamic
  - ◆ other methods are generally bound static

- In Java all methods are bound dynamic
  - ◆ static binding can be enforced by the keyword **final** in the method declaration
  - ◆ such methods cannot be reimplemented in subclasses

```
public final void incr() { value += step; }
```

C.7 Basic Concepts of the OO Paradigm

- Abstraction
  - Encapsulation
  - Abstract data type
- Modularization
- Hierarchy
- Typing
  - Hierarchy of types
  - Polymorphism
  - Genericity
- Concurrency
- Persistence

1 Abstraction

Fundamental concept for solving complex problems

- Emphasize details which are relevant for the entire solution
- Suppress details which are (for the moment) immaterial or diversionary

➜ Object orientation

- important:
  - ➤ Signature of an object
  - ➤ Semantics of an object

  } Outside view

  ➥ contract model: Outside view = contract with other objects

- unimportant:
  - ➤ Implementation of an object

- Describe the abstraction first and think about the implementation later

2 Encapsulation

= Information Hiding

Concealing of the implementation of an abstraction from the users of the abstraction

- Complement to abstraction
  - ➤ Abstraction exposes the external properties of an object
  - ➤ Encapsulation hides the internals

- Fundamental for abstraction

B. Liskov *For abstraction to work, implementations must be encapsulated*

- Encapsulation & object orientation
  - ◆ Representation of the object state
  - ◆ Implementation of the methods

➔ Abstract data type

3 Abstract Data Type

? How can we provide a complete, precise & unambiguous description of an abstraction?

- ADT: Model to describe
  - ➤ Properties of data structures (Semantics!)
  - ➤ Operations of the data + semantic effects
  - ➤ NOT: Implementation of the data structures

- Specification → Description of all essential properties

- Overspecification: Details about the representation/implementation

➔ ADT & object orientation

◆ Class = Implementation of an ADT

◆ Data abstraction: Object state is accessible only through methods

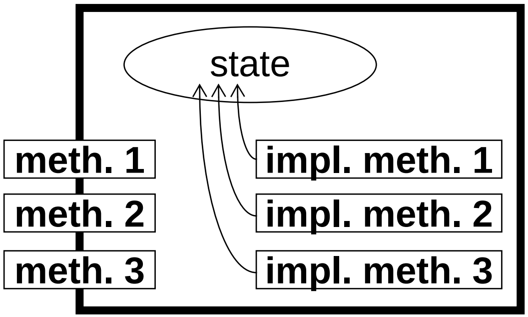
3 Abstract Data Type (2)

Example: Specification of an ADT “Stack”

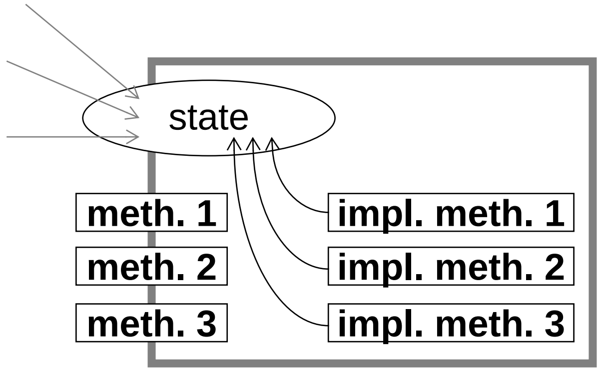
```
TYPES
        STACK[X]

FUNCTIONS
        empty: STACK[X] → BOOLEAN
        new: → STACK[X]
        push: X × STACK[X] → STACK[X]
        pop: STACK[X] → STACK[X]
        top: STACK[X] → X

PRECONDITIONS
        pre pop(s: STACK[X]) = (not empty(s))
        pre top(s: STACK[X]) = (not empty(s))

AXIOMS
        ∀x:X, s:STACK[X]:
        empty(new())
        not empty(push(x,s))
        top(push(x,s)) = x
        pop(push(x,s)) = s
```

3 Abstract Data Type (3)

- Data abstraction in OOP
  - ➜ the object state is accessible only through the methods of the object



object as implementation of an ADT



object without data abstraction

- Data abstraction in C++ and Java
  - ◆ Scope rules (private / protected)

4 Modularity

Partitioning a program into individual components can reduce its complexity

- Problem partitions are easier to comprehend
- Different problem partitions may be assigned to different developer teams
- Module = separate unit in development

- **most important: partitioning creates boundaries = interfaces**
  - → well-defined
  - → documented

  invaluable for the comprehension of the whole problem

- Many programming languages make a difference between interface and implementation of a module

- Structured Design: Grouping of subprograms

- OOD: Grouping of classes and objects (from the design’s logical structure)

5 Hierarchy

- ◆ Abstraction & encapsulation help to hide details of components
- ◆ Modularity helps to cluster related abstractions

- Comprehension of large problems still difficult
  - ➔ too many abstractions
  - ➔ additional means to organize abstractions

- Abstractions often form hierarchies
  - ➤ common properties → more general abstractions
  - ➤ differences → specialization
  - ➔ Hierarchy: Ordering of abstractions

- Hierarchy & object orientation
  - ➤ Class structure: Inheritance → “is a” hierarchy
  - ➤ Object structure: Aggregation → “part of” hierarchy

6 Typing

- Concept derives primarily from the theory of ADTs

Booch: *Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at most, they may be interchanged only in very restricted ways*

- Typing enables a programming language to check and enforce design decisions
  - ➔ essential for programming-in-the-large [Weg87]

- Strong typing:
  Conformance of all types in an expression is guaranteed
  - ➔ Static typing:
    conformance checked completely at compilation time
    - ➤ less flexibility
      - cannot detect compatible types
      - cannot support dynamic binding
  - ➔ More flexibility with polymorphism and genericity

7 Type Hierarchy

- Often 1:1 relationship between classes and types — but not necessary
  - ➤ different classes may implement the same type
  - ➤ one class may implement different types

- Hierarchy of classes: superclass ← subclass
  - ◆ Objective: code inheritance
  - ◆ Subclass not necessarily confirming to the type of the superclass

- Hierarchy of types: supertype ← subtype
  - ◆ Objectives: – behavior inheritance
    – declaration of conformable types (➜ polymorphism)

- Type inheritance (subtyping) as means for deriving types
  - ➜ Relationship between types becomes comprehendible
    - ➤ allows the identification of conformable types

8 Polymorphism

- Polymorphism = *the ability to take several forms*
  - ➤ various types for values or variables
  - ➤ various types for the parameters of functions
  - ➤ various types for the operands of an operator

- Example:

  operator "+" works with operands of type **`int`** and **`real`**

- Classification

  - ***polymorphism***
    - ***universal***
      - ***parametric*** (genericity)
      - ***inclusion*** (subtyping)
    - ***adhoc***
      - ***overloading***
      - ***coercion*** (type casting)

9 Polymorphism in C++

- Overloading polymorphism
  - ➤ Function-name overloading
  - ➤ Operator overloading

- Inclusion polymorphism
  - ➤ public inheritance

- Coercion polymorphism
  - ➤ Cast operator

9 Polymorphism in C++ (2)

★ Function-name Overloading

- several functions with the same name but different signature within one scope

- compiler selects the proper function
  - ➔ static binding (*early binding*)

- Example:

```
int max (int a, int b) { return((a>b):a?b); }
float max (float a, float b) { return((a>b):a?b); }
char *max (char *a, char *b) {
        if ((strcmp(a,b)> 0) return a; else return b; }
```

- Special cases:
  - ➤ several constructors for a class
  - ➤ operator overloading
  - ➤ overloading of a method in the context of inheritance

9 Polymorphism in C++ (3)

★ Cast Operator / Coercion Polymorphism

- In most programming languages automatic type conversion for basic types (e.g. short → int → double ← float)

- Cast operator (cast method) allows the definition of an explicit conversion of the class type into another type

```
class::operator type(void) { ... }
```

- Cast operator is like a normal unary operator

9 Polymorphism in C++ (4)

★ Cast operator — Example

```
class time
{
  private:
    int hour, min, sec;
  public:
    ...                             // Konstruktor

    operator long () {              // cast-Operator
      return(hour*3600 + min*60 + sec);
    }
};
main() {
  time now(1,10,2);
  long t;

  t = now + 10;                 // ⇒ t == 4212
...
```

9 Polymorphism in C++ (5)

★ Inclusion Polymorphism — *THE Polymorphism in OOP*

Inheritance + Virtual Methods + Object References

- Object reference (pointer) has a type (= class)
  - instances of this class and all of its subclasses may be assigned to the ref.
  - on method invocation, the actual implementation of the method is designated not by the class of the pointer but by the class of the current object

➜ You may always assign any type-conformable object to an object reference and everything will work

- you may pass it as parameter to a method
- the programmer of the method did not need to know anything about your new subtype — as long as it conforms to the supertype his method expects

9 Polymorphism in C++ (6)

★ Virtual Methods & Inclusion Polymorphism — Example:

```
class geo_obj {                          // general superclass
  public:
    virtual void draw();
};

class circle : public geo_obj {          // subclass
  public:
    void draw();
}

class square : public geo_obj {          // subclass
  public:
    void draw();
}

main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```

Types & C++
Abstract Classes

- Superclass declares methods and its signatures but does not define them
  - – **pure virtual functions**

  ➜ superclass defines only a type

- Subclasses define various implementations of the methods

  ➜ each subclass is one implementation of the type

- Not possible to instantiate objects from the superclass

- Example:

```
class geo_obj {                          // abstract calss
  public:
    virtual void draw() = 0;             // pure virtual function
};
class circle : public geo_obj {          // subclass
  public:
    void draw() { ... }
}
```

11 Types & Java
Abstract Classes

- As in C++
  - instantiation not possible
  - missing parts have to be completed in the subclass

- Example:

```
abstract class geo_obj {                // abstract class
  public abstract void draw();          // pure virtual function
};
class circle extends geo_obj {          // subclass
  public void draw() { ... }
}
```

12 Types & Java Interfaces

- 2 possibilities to declare a type
  - ◆ by a class definition
    - ➜ class inheritance automatically leads to type inheritance
  - ◆ by a interface declaration
    - ➜ separate type declaration
- Example:

```
public interface Printable {
    public void Print();
}

public class MyPoint extends Object implements Printable {
    ....
    public void Print() {
        System.out.println("x="+x+" y="+y);
    }
}
```

12 Types & Java Interfaces (2)

- Inheritance + multiple inheritance for interfaces
- One class may implement several types
- Type conformance is transitive
- Exceptions are an element of the type interface
- Examples:

```
interface Streamable extends FileIO, Printable {
    // additional Methods
    public void test() throws TestException;
}

class Test implements Streamable, Testinterface {
    ...
}
```

13 Genericity

Concept that allows the definition of the type of some unit of the programming language at invocation or instantiation by a parameter

- OOP: generic classes ➔

  generic class ➔ instantiation (+ parameterization) ➔ actual class

  actual class ➔ instantiation of objects

- Example:
  - ◆ general stack class
    - ➤ int stack
    - ➤ real stack
    - ➤ string stack

- Most benefit of genericity can also be achieved by inheritance

- Implemented in Ada, Eiffel and C++ (*Templates*)

14 Genericity & C++: Templates

- Objective: definition of a class without finalization of types
  - ➤ dynamic type checking at run time or
  - ➤ static type checking at compile time + parameterizable classes
- Template = parameterizable class
- Example:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
              { index = 0; array = new T[n]; }
    void push(T elem)
              { array[index++] = elem; }
    T pop(void)
              { return(array[index--]); }
};
```

- Parameterization of class on object instantiation

- Examples:

  - Instantiation and usage of a stack with elements of type float:

```
stack <float> s(10);
float a;
s.push(a);
a = s.pop( );
```

  - Instantiation and usage of a stack with elements of type integer:

```
stack <int> s(10);
...
```

15 Concurrency

Several threads of control are processed in parallel on different processors or quasi-parallel on one single processor

- Concurrency is orthogonal to object orientation (in general)

but: concurrent solutions are much more complex than sequential ones

- Granularity: concurrency / objects (capsules)
  - ➤ Finer grained concurrency
    - ➜ even object-internal concurrency
  - ➤ Coarser grained concurrency
    - ➜ just object-external concurrency

- Integration of concurrency control into OO programming languages
  - ➤ orthogonal languages
  - ➤ non-orthogonal languages
    – uniform / non-uniform languages

16 Concurrency & Java

- Thread concept and synchronization mechanisms are integrated
  - ➥ non-orthogonal, non-uniform language
- Creation of threads via thread class
- Example:

```
class MyClass implements Runnable {
        public void run()  {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();    // create object
Thread t1 = new Thread(o1);    // create thread to run in o1

t1.start();                    // start thread

Thread t2 = new Thread(o1);    // create second thread to run in o1

t2.start();                    // start second thread
```

★ Synchronization

- Monitors: exclusive execution of methods of an object
  - ◆ Example:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

- Conditions: releasing a monitor while waiting on an event

17 Persistence

★ **Motivation for Persistence** [ABC83]

- “active” data → programming language facilities / run-time environment
- “passive” data → DBMS or file system

➜ 2 different views of data

➜ Disadvantages for the programmer
  - ➤ Conversion between active and passive data necessary
  - ➤ Data type protection of programming language is lost

Persistence (2)

General Definition

- Persistence is the property of data through which
its existence transcends time
(i. e. it continues to exist after its creator ceases to exist)
and/or space
(i. e. its location moves from the address space in which it was created).

Spectrum of Persistence

1. Transient results in expression evaluation
2. Local variables in procedure activations
3. Global variables and heap items whose extent is different from their scope
4. Data that exists between executions of a program
5. Data that exists between various versions of a program
6. Data that outlives the program

Persistence (3)

★ Persistence in Object-oriented Systems

- Objects survive the termination of the environment (thread, application execution) in which they were instantiated or used

- Powerful mechanism in OO operating systems for
  - ➤ Data storage
  - ➤ Data transport

- Examples
  - ➤ File systems
  - ➤ Database systems
  - ➤ Persistent communication objects

- Properties of the object-oriented programming model are automatically inherited by all mechanisms which are constructed on its basis

C.8 OOA & OOD

1 Overview

- **OOA: *What* shall my system do?**
  - ◆ Basis: Analysis of the “real world”
    - ➤ Components, terms, tasks
    - ➤ Requirements and constraints
  - ◆ Abstraction from unimportant aspects & implementation details
- **OOD: *How* does my system do it?**
  - ◆ Transformation from the analysis model to an implementable model
  - ◆ Add aspects of the implementation environment
  - ◆ Structural and strategic decisions
    - ➤ Threads, distribution, IPC, error handling, garbage collection, …
- **Design patterns:** Guidelines for the OOD

2 OOAD Methods

- Notation
- Process
- Well-known examples for OOAD methods: Booch, OMT and UML
- CASE tools support OOA and OOD
  - ➤ Diagrams
  - ➤ Documentation
  - ➤ Support for multiple users
  - ➤ Consistency checks
  - ➤ Generation of code skeletons

3 OOA — Process

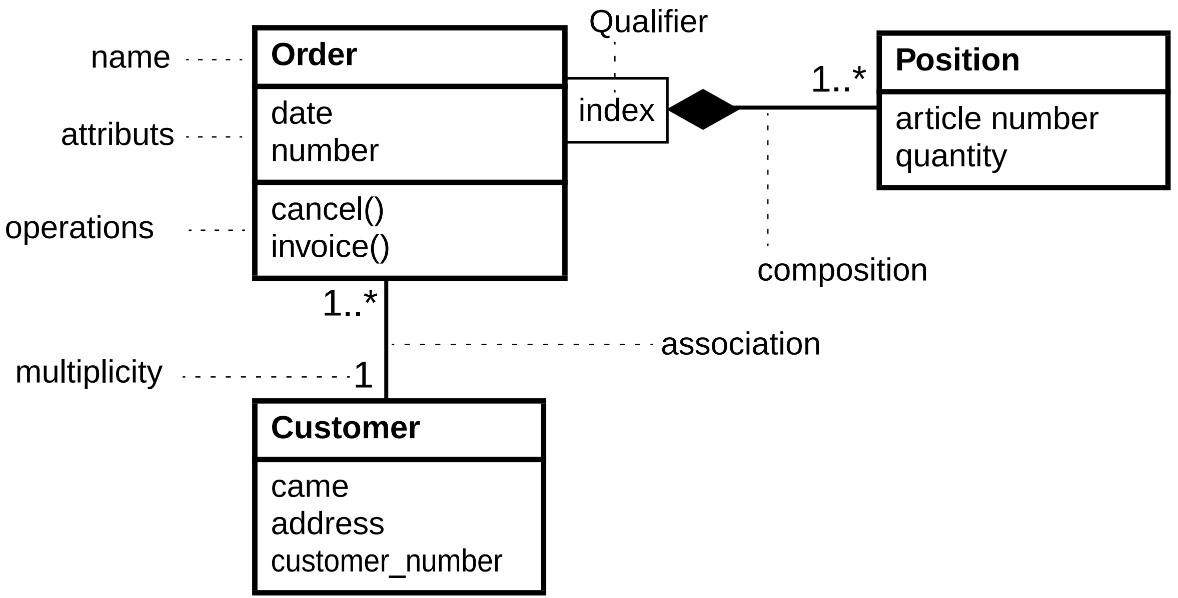
- Requirement Analysis
  - ➤ Determine problem domain
  - ➤ Specify goals from the user's point of view
  - ➤ Performance and architectural requirements
- Use cases
  - ➤ Describe interaction between “user” and application
  - ➤ Participants: actor & use case
  - ➤ Substantiates the requirements analysis
- Find objects
  - ➤ Look for terms of the problem domain (nouns in the description)
- Organize objects (requirements model -> analysis model)
  - ➤ different object types (interface objects, entity objects, control objects)
- Refine structure

4 OOD

- Transition from OOA to OOD often very smooth
  - ➤ When implementation aspects cannot be kept back any longer
- Transformation from the analysis model to an implementable model
- Class design
  - ➤ Find software classes for the classes of the analysis model
  - ➤ Keep up class boundaries (Analysis -> Design -> Implementation)
    - ➥ Traceability
- System design
  - ➤ problem-independent aspects
    (distribution, concurrency, resource management, …)
- Program design
  - ➤ Programming language
  - ➤ Error messages, exceptions, performance optimizations

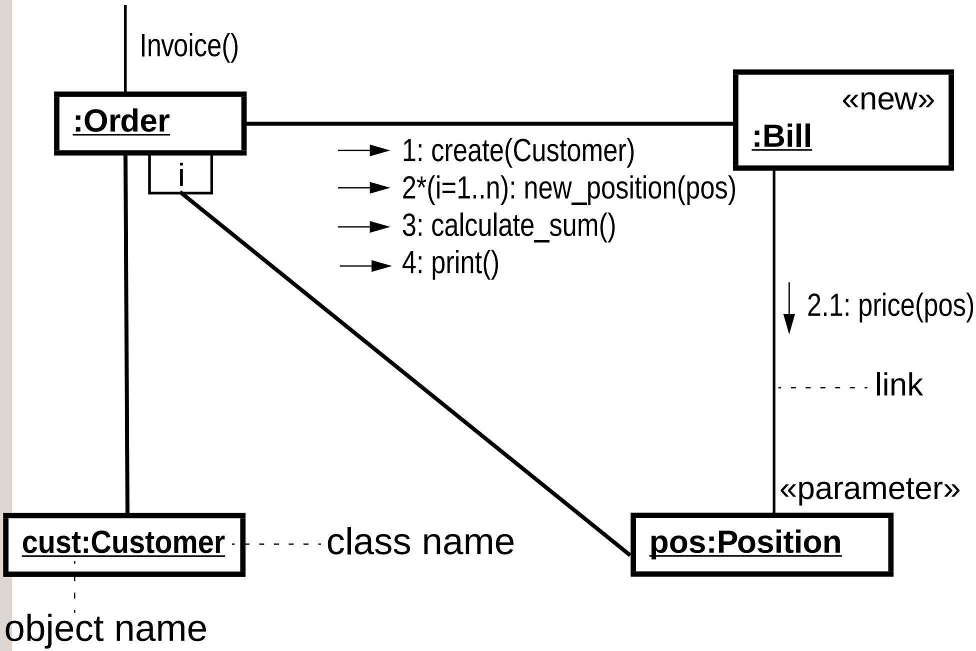
5 UML Notation(1): Class Diagrams

- Class with its attributes (variables) and operations (methods)
- Relations between classes
- Example:



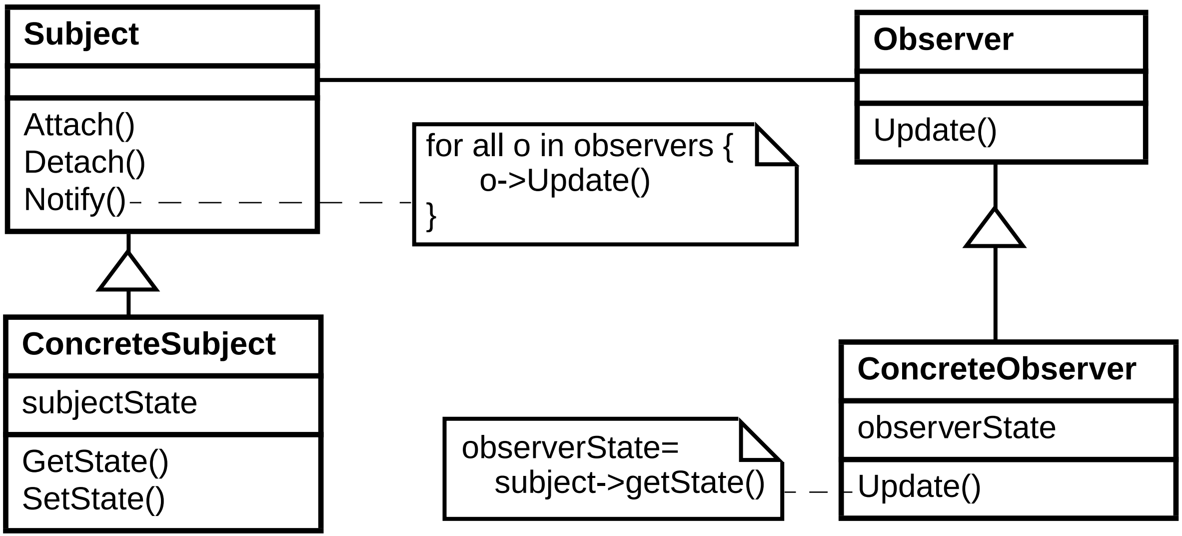
6 UML-Notation(2): Collaboration Diagrams

- Model collaboration of objects to fulfil a certain task
- Example:



7 Design Patterns

- A design pattern describes a solution for a class of problems
- Components of a pattern: Name, problem class, solution, consequences
- Example:
  Observer Design Pattern: Other objects depend on the state of an object and have to be informed about changes of the state



D Distributed Systems

D.1 Definition and Motivation

- *"Distributed System"*
  Definition according to Tanenbaum and van Renesse
  - ◆ It looks like an ordinary centralized system.
  - ◆ It runs on multiple, independent CPUs.
  - ◆ The use of multiple processors should be invisible (transparent).
- *"Distributed System"*
  Definition according to Mullender
  - ◆ Additionally: Not any single points of failures
- Definitions are not precise
  - ◆ Sometimes it is hard to identify a centralized or a distributed system.
  - ◆ Definitions are always based on certain characteristics that are important.

1 Motivation

- Efficiency to cost ratio
  - High performance computers are very expensive
  - Microprocessors became very cheap
  - Multiple microprocessors can easily have more computing power than a high performance computer and cost much less.
- Centralized CPU vs. personal computer
  - Response time of centralized systems is very bad at high load.
  - Personal computers are available for a single user.
  - More computing power available for a single user: better user interfaces, etc.
- Inherent distribution
  - People are distributed
  - Information is distributed
  - Devices are distributed

1 Motivation (2)

- Expandability, incremental growth
  - ◆ It is easier to add a new computer to a distributed system than to extend a high performance machine.
- Availability
  - ◆ Distributed systems can have redundant components (CPUs, memory, communication channels, etc.)
  - ◆ System runs on when a component fails.
- Reliability
  - ◆ Reliability needs availability.
  - ◆ Reliable systems mask failures (e.g., CPU failure, communication failures, etc.)
- Scalability
  - ◆ "No" restriction on the maximum size of the system.

2 Advantages

- Costs
  - ◆ Distributed systems can be much cheaper at same capacity.
  - ◆ Expensive devices (e.g., color printers) can be shared by many users.
- Efficiency
  - ◆ Distributed systems can be much more efficient than any available high performance computer.
- Inherent Distribution
  - ◆ Distributed systems model the inherent distribution of today's organizations.
  - ◆ People can communicate via distributed systems. Some day, a distributed system might replace the POTS (plain old telephone system).
- Reliability
  - ◆ Distributed systems can be made very reliable. However, this is a difficult task.

2 Advantages (2)

- Incremental Growth
  - Distributed systems can be easily extended.
- Load Balancing
  - Unlike individual PCs, a distributed system can grant peak performance to a single user without annoying other users.

3 Disadvantages

- Concurrency
  - Distributed systems are inherently concurrent.
  - Controlling concurrency is complex.
  - Combining well-understood components can generate new problems not apparent to the components.
- Propagation of effect
  - One malfunctioning computer can bring down the whole system.
  - There can be unforeseen dependences between components.
- Security
  - It is harder to secure a physically distributed system.
  - Communication channels can be wire tapped and eavesdropped.
  - Data access could not be controlled on certain sites.

3 Disadvantages (2)

Efficiency

- Distributed systems can only gain efficiency for the total output of the entire system. If you cannot parallelize your application you cannot benefit from the available high performance.

Load Balancing

- It is hard to balance the load because the physical distribution of resources may not match the distribution of demands.

Scalability

- A working system with ten nodes may fail miserably when it grows to a hundred nodes.

Complexity

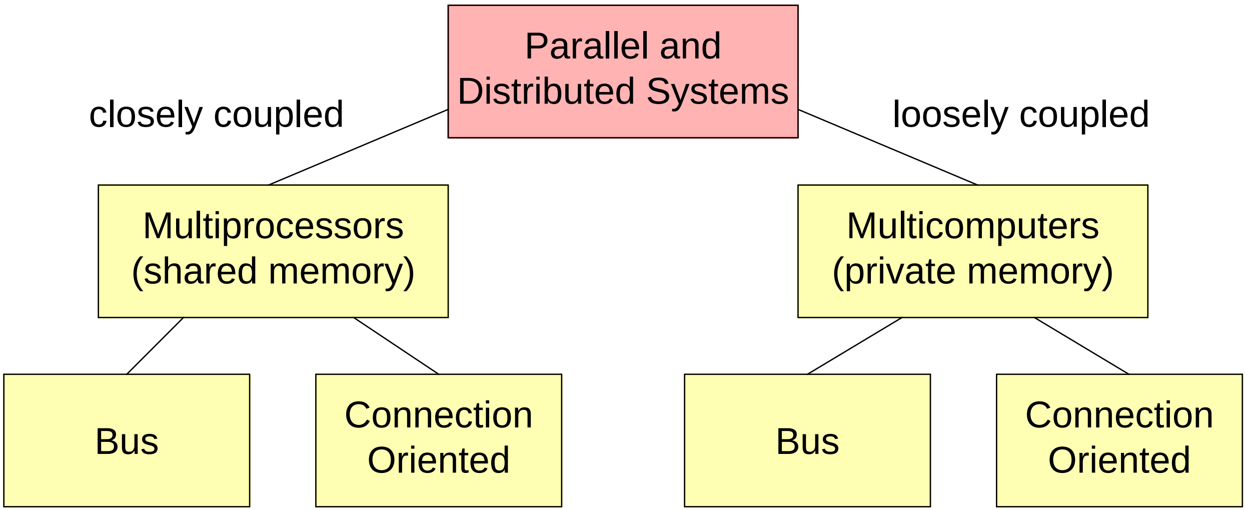
- All in all, a distributed system is much more complex than a centralized one (e.g., dealing with partial failures, concurrency, load balancing, etc.)

D.2 Taxonomy

- Classification according to Flynn (1972)
  - SISD – Single Instruction Stream, Single Data Stream
    all current single CPU computers (PCs, Mainframes)
  - SIMD – Single Instruction Stream, Multiple Data Streams
    high performance computers, vector computers
  - MISD – Multiple Instruction Streams, Single Data Stream
    no known system available that implements this category
  - MIMD – Multiple Instruction Streams, Multiple Data Streams
    systems with independent CPUs
- Distributed systems are always seen as MIMD computers

D.2 Taxonomy (2)

- Taxonomy of parallel and distributed computer systems

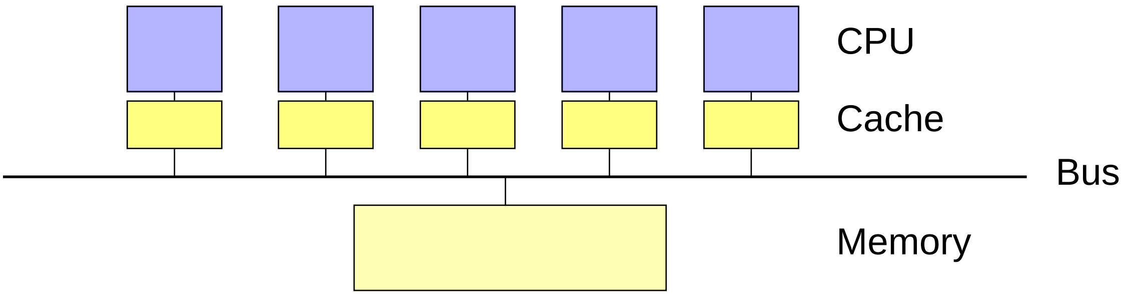


according to Tanenbaum 1995

1 Multiprocessors

- Shared memory
  - ◆ All CPUs share the memory
  - ◆ Memory is coherent
    - Written data items are immediately visible to other CPUs

- Bus-based systems
  - ◆ CPUs access memory via a bus
  - ◆ Limited number of CPUs
  - ◆ Increased performance by CPU-side caches
  - ◆ Cache consistency achieved by bus snooping

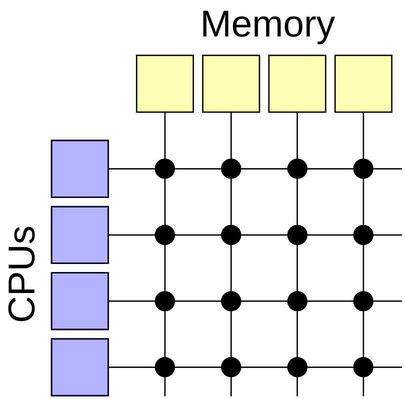


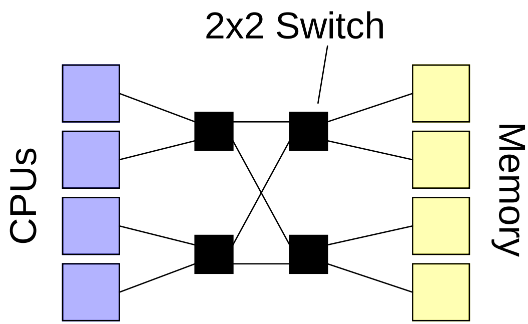
1 Multiprocessors (2)

Connection-oriented systems

- For more than 64 processors bus-based systems fail
- Cross-bar switch

Omega switching network



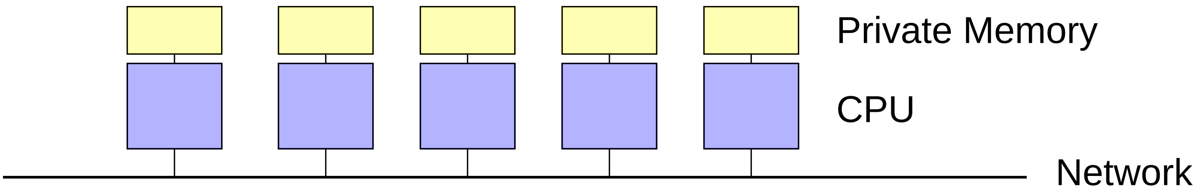


- Cross-bar switches need n^2 switches
- Omega networks need $n \cdot \log_2 n$ switches
- Slow memory access
- Solution: hierarchical systems (NUMA = Non uniform memory access)

2 Multicomputers

- Each CPU has its own private memory

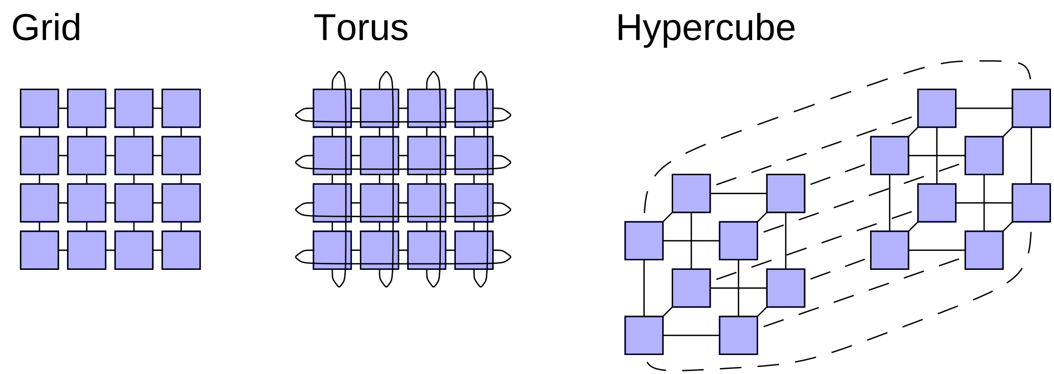
- Bus-based multicomputers
  - ◆ Workstations in a LAN



  - ◆ CPUs connected to a fast communication bus

2 Multicomputers (2)

- Connection-oriented multicomputers
  - Examples of topologies:



  - Each CPU is connected to a number of other CPUs

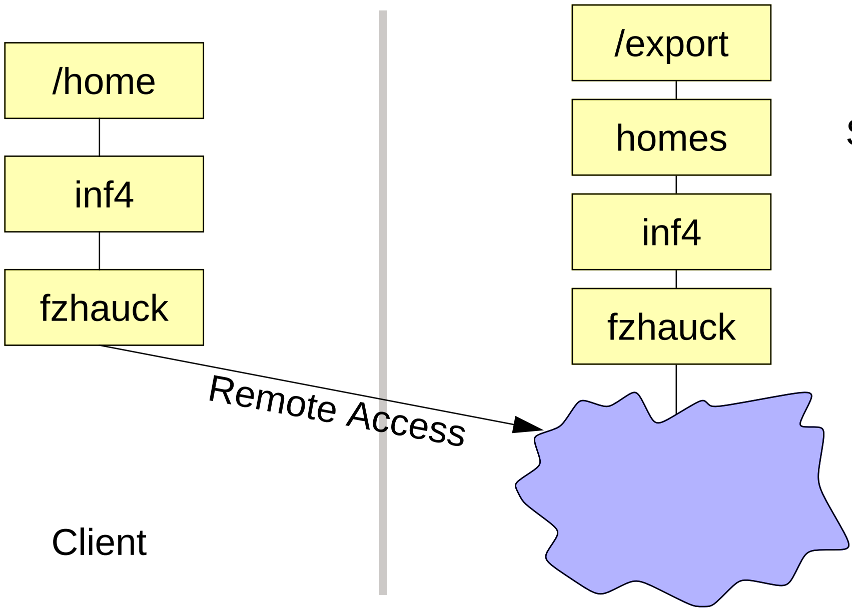
- Computers in a wide area network?
  - Bus-based, as each CPU is virtually connected to every other
  - Connection-oriented, as there is no uniform access to other CPUs

3 Network Operating Systems

- Early distributed systems
- Loosely-coupled systems
  - Multicomputers usually in a LAN
- One (but not necessarily the same) operating system on each system
  - Users act locally
  - Users have access to remote systems
    - Remote login: `rlogin faui04a`
    - Remote copy: `rcp faui04a:aFile myCopy`
    - Shared file systems
    - Shared devices (e.g., printers)

3 Network Operating Systems (2)

- Shared file systems
  - ◆ Users can operate on remote files as on local files
  - ◆ File server provide remote access to local files
  - ◆ Local file name is not necessarily equal to remote file name



4 True Distributed Systems

- Same operating system on each node

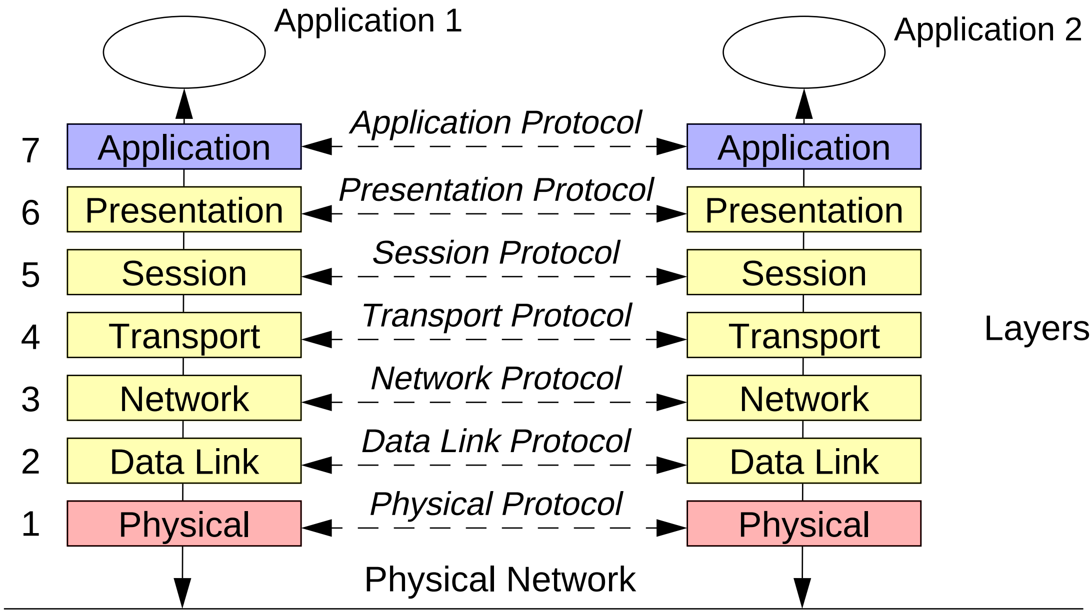
- System behaves like a uniprocessor
  - Users should not see any differences
    if they access the system from another node.
  - The identity of the local computer is not important.
  - File sharing semantics is usually well-defined.

- Transparencies
  - Location transparency — location of resources is irrelevant
  - Migration transparency — resources may move
  - Replication transparency — resources may be replicated
  - Concurrency transparency — multiple accesses to a resource at a time
  - Parallelism transparency — activities may be executed in parallel

Communication Models

- Communication needs agreement
  - Protocols

1 Protocol layers according to the ISO OSI reference model



1 Protocol Layers (2)

- Physical Layer
  - ◆ Transmission of 0s and 1s on the wire
- Data Link Layer
  - ◆ Sending bits; separating frames or packets; checking frame integrity
- Network Layer
  - ◆ Routing of messages in larger networks
- Transport Layer
  - ◆ Implementation of reliable connections
  - ◆ Fragmentation and reassembling
- Session Layer
  - ◆ Dialog control; synchronization

1 Protocol Layers (3)

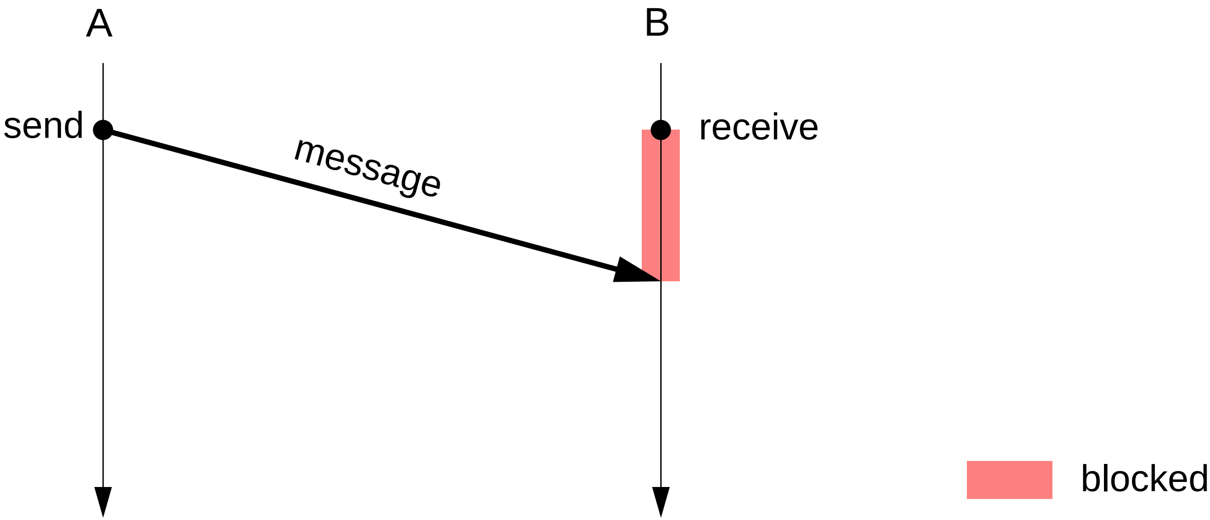
- Presentation Layer
  - ◆ Transparency of different internal representations of data
- Application Layer
  - ◆ Set of application protocols
    - Electronic mail protocol
    - File transfer protocol
    - etc.

2 Classification

- Synchronity
  - Is the sender blocked until the receiver gets a message, or not?
- Pattern of Interaction
  - Message Passing — a message is sent from one party to the other
  - Request-Reply (Client-Server) Interaction —
    there is a message to the receiver and a message back to the original sender
- Addressees
  - One receiver
  - Multiple receivers (group communication, multicast, broadcast)

2 Datagramm Message

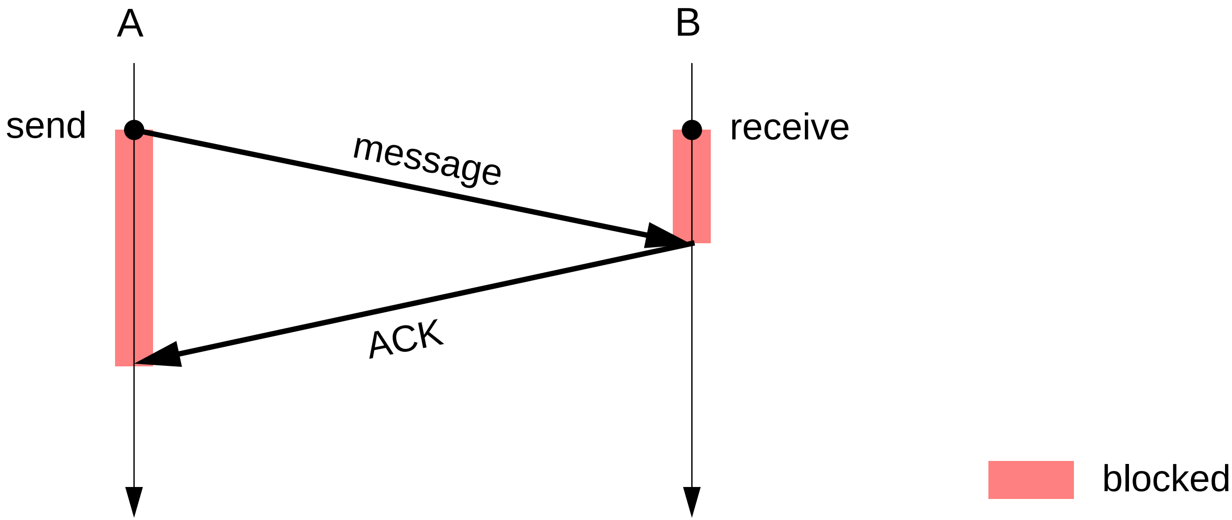
- Message passing; asynchronous send



- Sender can proceed immediately
- Receiver may be blocked until a message arrives
- Needs buffer space for not yet received messages

3 Rendezvous Model

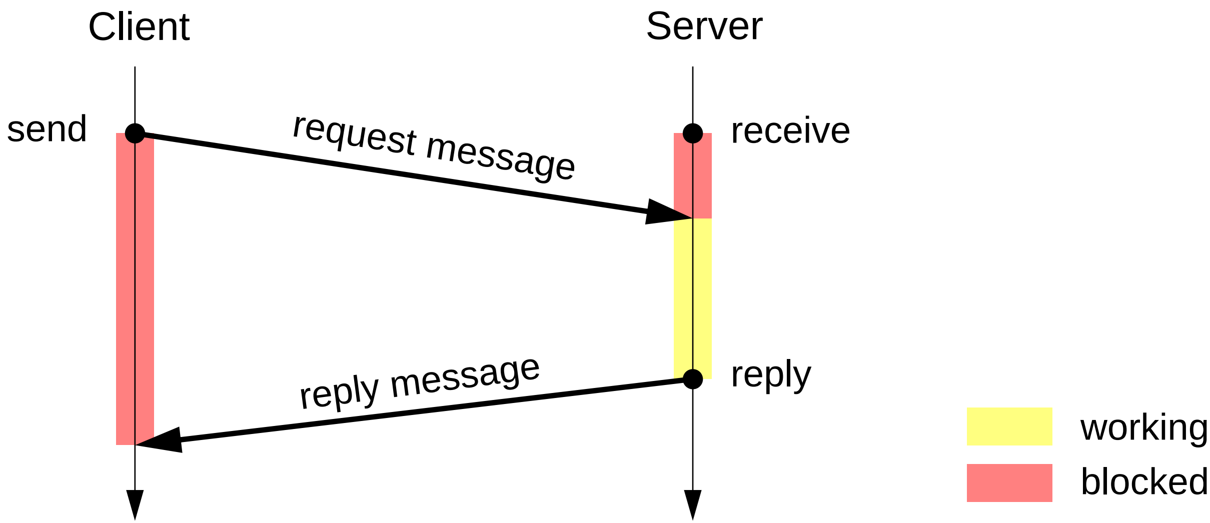
- Message passing; synchronous send



- ◆ Sender waits until message is received
- ◆ Receiver may be blocked until a message arrives
- ◆ Needs no buffer space

4 Synchronous Request-Reply Model

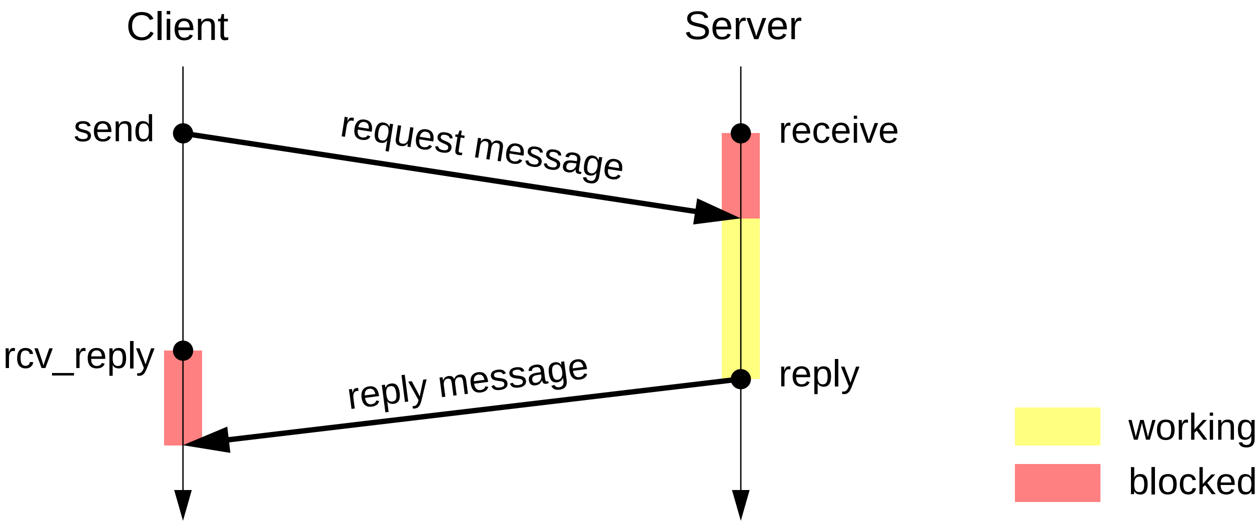
- Request-reply interaction; synchronous send



- ◆ Client waits until reply message is received
- ◆ Server may be blocked until a request message arrives
- ◆ Client and server do not work concurrently
- ◆ Well known representative is the RPC (remote procedure call)

5 Asynchronous Request-Reply Model

- Request-reply interaction; asynchronous send



- ◆ Client and server can work concurrently
- ◆ Basis for group communication

6 Reliability

- It is possible that messages get lost if we do not use a reliable connection
  - Reliable connections introduce acknowledge messages (ACK)
  - For simple message passing this means a lot of overhead

★ Combining reliability with the request-reply interaction model

- Possible errors
  - Server crash
    failure model is: total amnesia
    (server looses all knowledge of former requests)
  - Request message gets lost
  - Reply message gets lost

- Ideal semantics
  - *exactly-once*
    The request is processed exactly once at the server side.

6 Reliability (2)

At-Least-Once Semantics

- Request is processed once or more times
- Client will never notice an error message, but it may notice that the request was processed multiple times: operations need to be *idempotent*.

Implementation

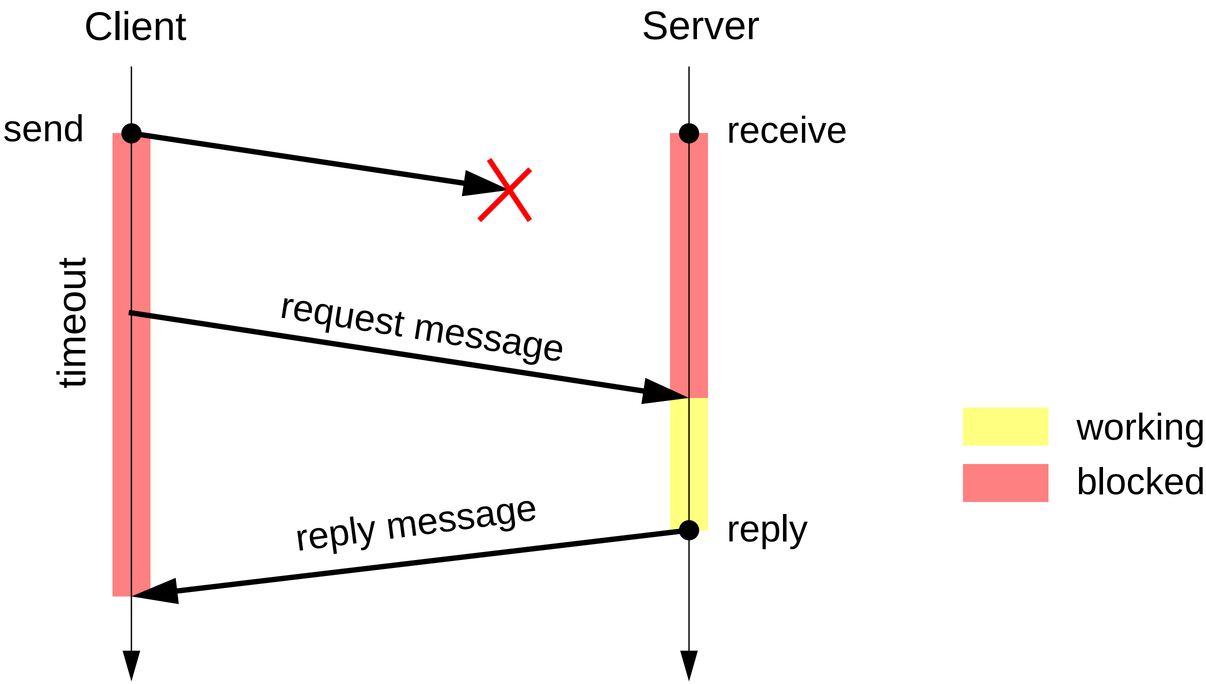
- If the client does not get a reply message after some time (time-out), it resends the request.
  - There is no additional functionality needed at the server side.
  - However, the server can ignore resent requests if it can detect them.

6 Reliability (3)

- **At-Most-Once Semantics**
  - The request is processed once or not at all.
- Simple implementation (at the client side only)
  - If the reply message does not arrive within a certain period of time an error is returned to the caller (at-most-once semantics).
  - Otherwise, the result is returned (exactly-once semantics).
- More complex implementation
  - Client repeats request message after time-out (hides message losses on the wire).
  - Client has to identify server crashes (error code to the caller, at-most-once semantics).
  - Server keeps reply messages (resend possible if message gets lost)
  - Server has to identify and ignore old requests after server crash.
  - If the result is returned we have exactly-once semantics.

6 Reliability (4)

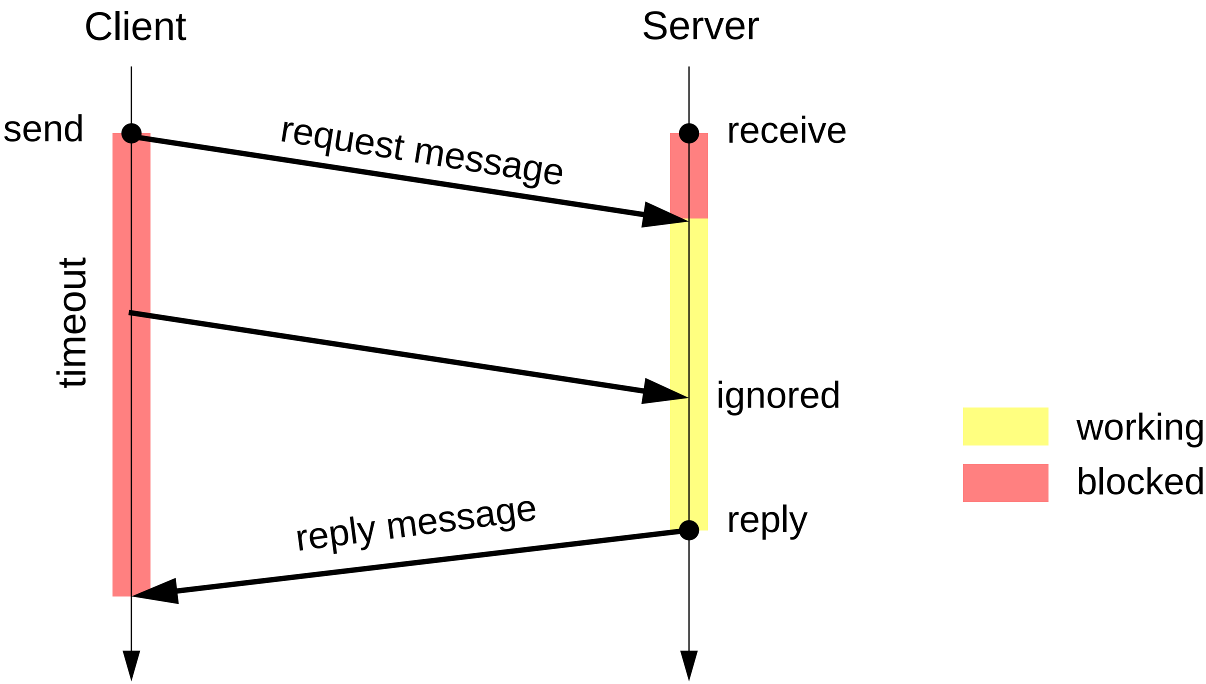
- Request message gets lost



- Request is repeated

6 Reliability (5)

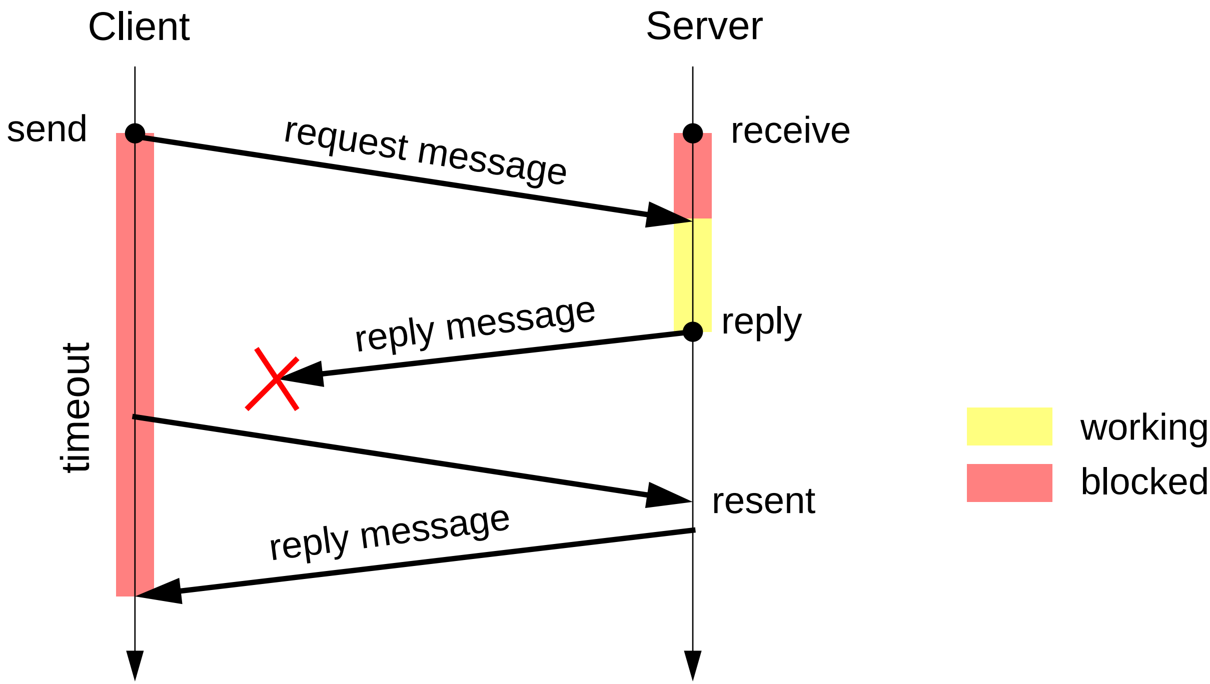
▲ Processing has not yet finished



◆ Repeated request is ignored

6 Reliability (6)

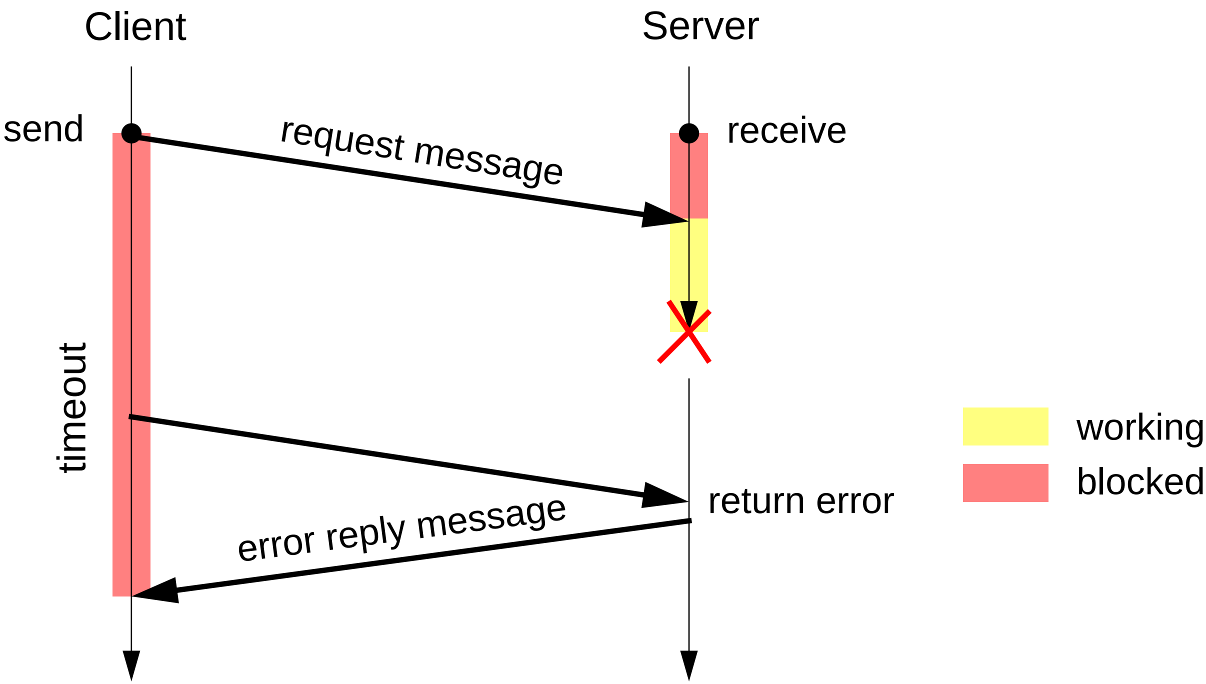
▲ Reply message gets lost



◆ Server keeps reply message and resends it

6 Reliability (7)

▲ Server crashes



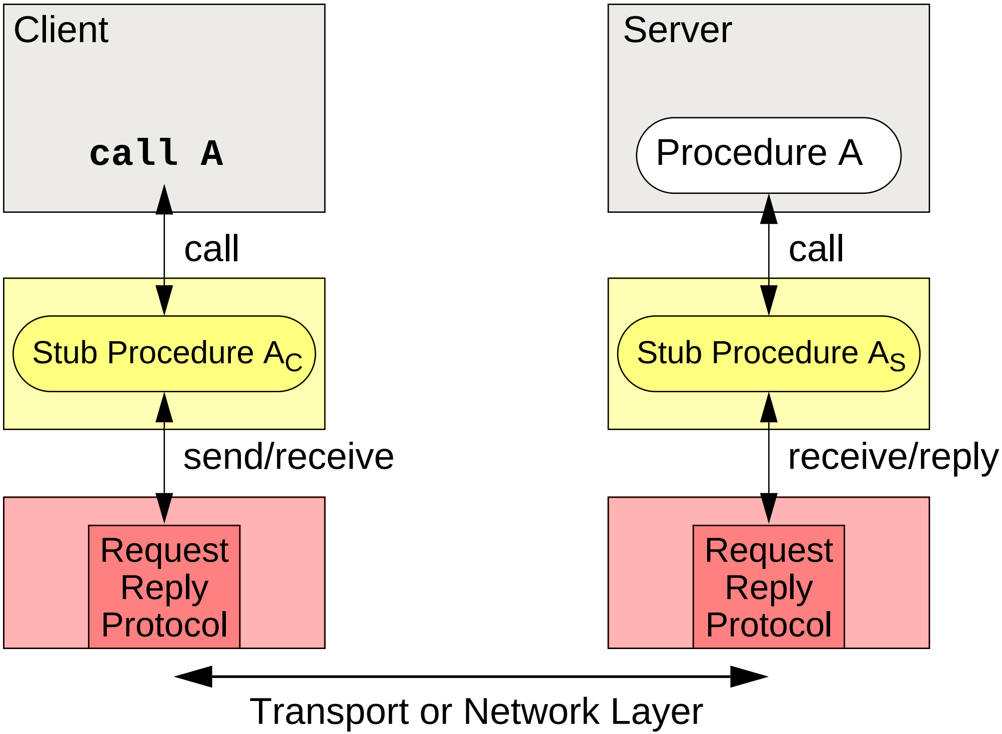
◆ Server identifies old requests (old generation number) and returns error code (at-most-once semantics)

7 Remote Procedure Calls

- Request-reply model can be used to implement RPCs [Birrell and Nelson 1984]
  - Instead of sending a request message, we invoke a remote procedure
  - Instead of receiving a reply message, we get the results of the invocation
- ★ Invocation of a procedure is location-transparent
  - Syntax may be the same for local or remote invocation
  - Very intuitive
  - No need for explicit usage of send and receive primitives
- Implementing RPCs
  - Stub procedures on client and server side

7 Remote Procedure Calls (2)

- Implementing RPCs using stub procedures



acc. to Nehmer 1995

7 Remote Procedure Calls (3)

- Client stub procedure
  - ◆ Marshalling of parameters (composing a request message)
  - ◆ Sending request message
  - ◆ Waiting for reply message
  - ◆ Unmarshalling of the result
  - ◆ Implementing delivery semantics
- Server stub procedure
  - ◆ Receiving request message
  - ◆ Unmarshalling of parameters
  - ◆ Invoking server procedure
  - ◆ Marshalling of the result
  - ◆ Sending reply message
  - ◆ Implementing delivery semantics

7 Remote Procedure Calls (4)

Problems with RPCs

- Marshalling of parameters
  - Number and types must be known (cmp. with C: `printf( "Count %d\n", count )`)
- Parameter passing semantics
  - *Call-by-value:* no problem
  - *Call-by-reference:* How to implement?
- No global variables
- Semantics
  - Server crashes; no exactly-once semantics
- Performance
  - No concurrency
  - Large parameter data
  - Short procedures

7 Remote Procedure Calls (5)

- Automatic generation of stub procedures
  - ◆ Tools generate code for:
    - parameter marshalling
    - client stub procedure
    - server stub procedure
    - server loop waiting for request messages
- Binding client stubs to server stubs
  - ◆ Server stub has a network address that must be known to the client stub
  - ◆ Problem: How does the client know its server?

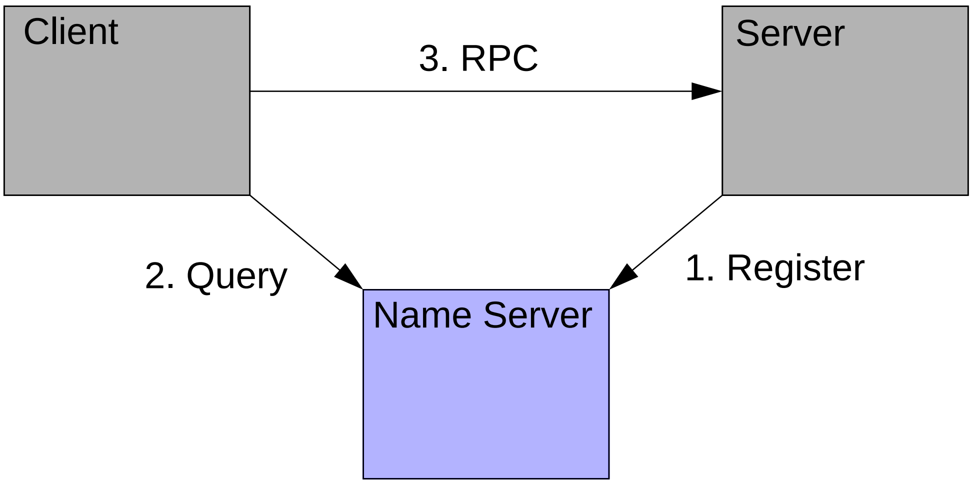
★ Name server

- ◆ Symbolic names are converted to network addresses

8 Name Server and Binding

- Well known name server converts names to addresses
  - Client knows a unique name for its server and the address of a name server
  - Name server converts this name to a dynamic network address

  - Client can always bind to the server
  - Server has to register its dynamic network address with the name server



9 Group Communication

Motivation

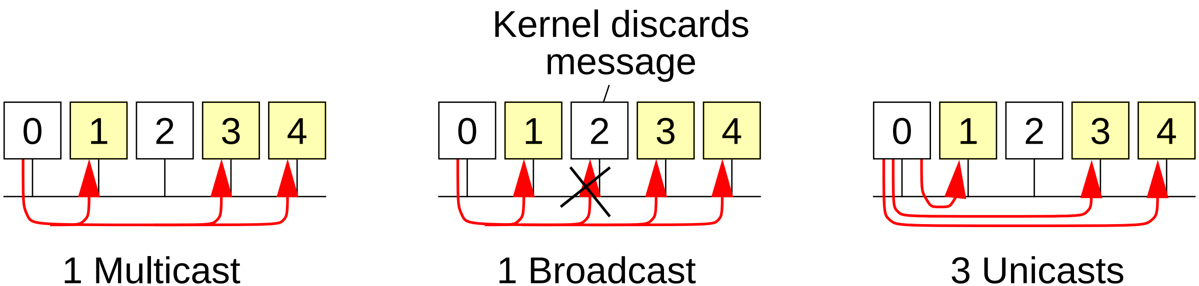
- Often more than one server need to be informed
  - multiple servers administrate a resource
  - multiple redundant servers (no “single point of failure”)

Terminology

- Unicast
  - One receiver (1:1)
- Multicast
  - Multiple receivers (1:n)
- Broadcast
  - All receivers of a special group (1:n)

9 Group Communication (2)

- Implementation of multicast
  - Using a hardware-based multicast
    - e.g., Ethernet multicast
  - Using a hardware-based broadcast
    - e.g., Ethernet broadcast
    - filtering of not addressed parties at receiver side
  - Using unicast messages
    - sending an individual message to each party



acc. to Tanenbaum 1995

Group Communication (3)

- Primitives for group communication
  - Message passing
    - Same primitives as for unicasts (send, receive) and multiple addressees for send
    - Different primitives: group_send, group_receive
  - Request-reply interaction
    - Multiple rcv_reply invocations to get all reply messages

- Variants of group communication semantics
  - *Reliability:* none, k-reliable, atomic/reliable
  - *Message ordering:* none, FIFO order, causal order, total order

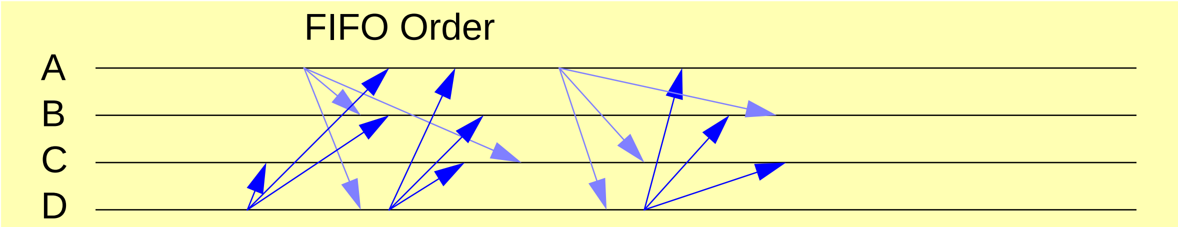
Group Communication (4)

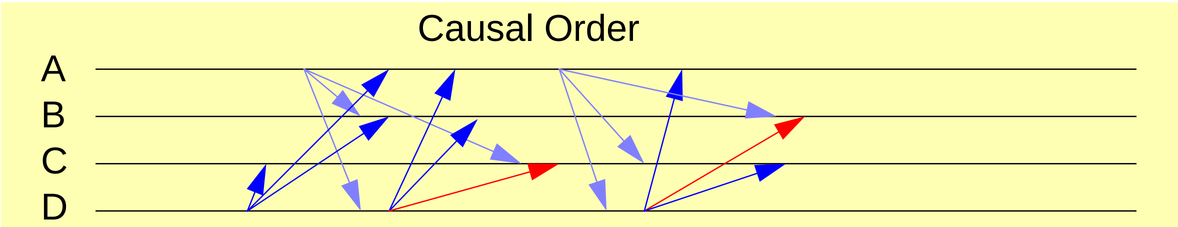
- Reliability
  - **None:** messages may arrive or may not arrive at a receiver
  - **K-reliable:** at least k members of the group receive the message
  - **Atomic/reliable:** all members or none of them receive the message

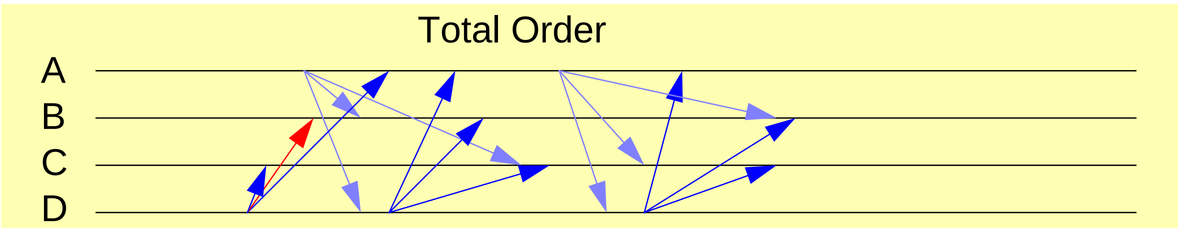
- Message ordering
  - **None:** messages arrive in arbitrary order at a receiver
  - **FIFO order:** messages arrive in the order sent by the sender
  - **Causal order:** causality of messages is reflected in the order of arrival
    - If a member of the group receives a message A and then sends a message B to the group, each group member will first receive A and then message B.
  - **Total order:** as causal order, but additionally not causally dependent messages arrive in the same order at each receiver

9 Group Communication (5)

- Examples for different message ordering

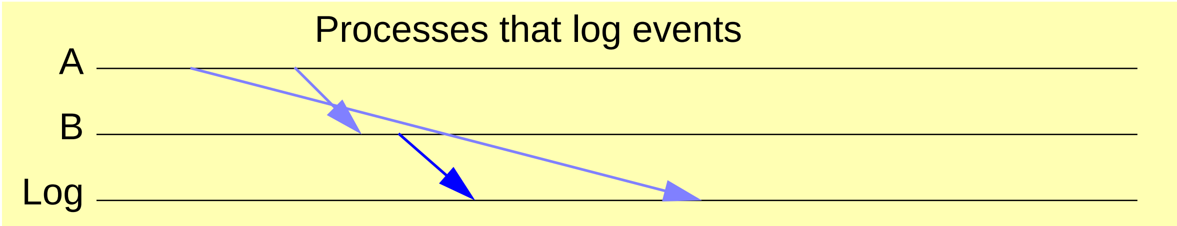






D.4 Selected Problems of Distributed Systems

- Causality
  - ◆ Simple message passing may violate causality



Log entries are not causally ordered

- Synchronization of processes
  - ◆ Semaphores and monitors depend on coherent shared memory
  - ◆ No shared memory on multicomputer systems

- Synchronization of clocks
  - ◆ System clocks are never exactly synchronized in distributed systems

D.4 Selected Problems of Distributed Systems (2)

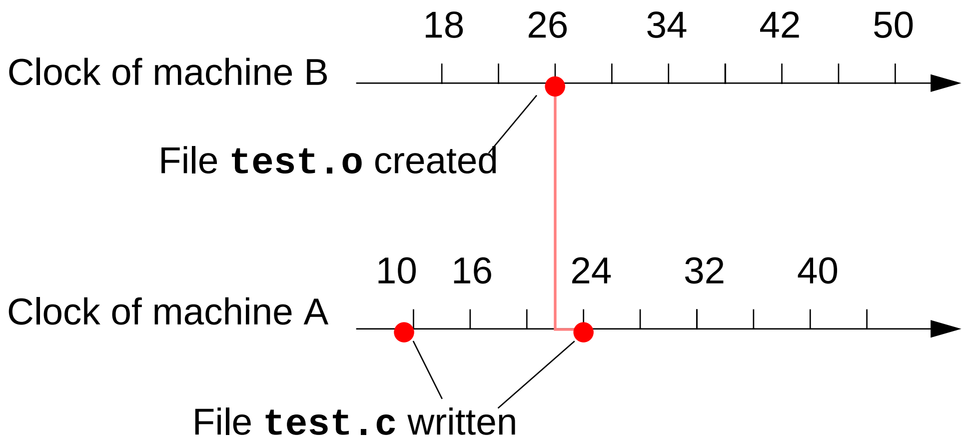
- Example: UNIX *make* command

Makefile

```
test.o: test.c

test: test.o
```

◆ Editor runs on machine A

◆ Compiler runs on machine B



➜ *Make* command will not notice necessary update!!

1 Logical Clocks

- Usually the precise absolute time is not necessary
  - We only need to know when one event causally depends on another
  - a → b is read "b is causally dependent on a"
  - If a → b and b → c then a → c (transitivity)
  - If neither a → b nor b → a is true then a and b are said to be **concurrent**
- Clock condition:
  - If an event b causally depends on an event a then timestamp of a must be less than the timestamp of b
  - $a \rightarrow b \;\Rightarrow\; T(a) < T(b)$
- Algorithm of Lamport (1978)
  - Messages as the only means for communication
  - Fulfills clock condition

1 Logical Clocks (2)

- Example

A	B	C
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

without logical clocks

A	B	C
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

with logical clocks

◆ Send event happens before arrival: send time must be less than arrival time!

◆ Solution: adjust local clock

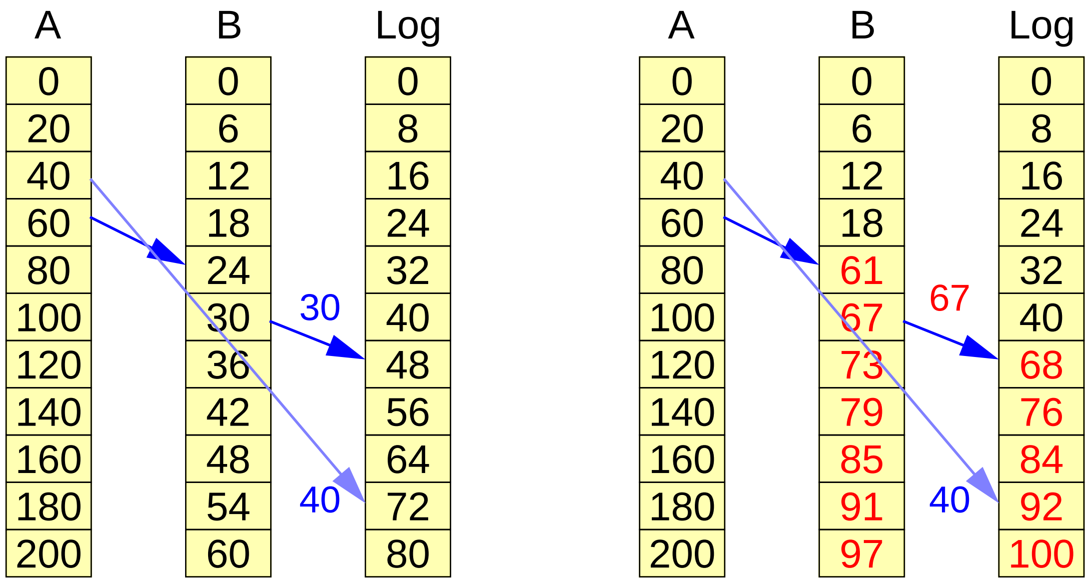
1 Logical Clocks (3)

Lamport's algorithm

- Each process has its own logical clock
  (a counter LC that is used for timestamping of events)
- Logical clock ticks for each local event
  - *Local event:* LC := LC + 1
  - *Send event:* LC := LC + 1; send(message, LC)
  - *Receive event:* receive(message, LC_S); LC := max(LC, LC_S) + 1
- Fulfills clock condition
- Reverse clock condition is **not** fulfilled!
  - $T(a) < T(b) \;\not\Rightarrow\; a \rightarrow b$

1 Logical Clocks (4)

- How does it help?
  - ◆ Logging processes: timestamp log messages with local clock

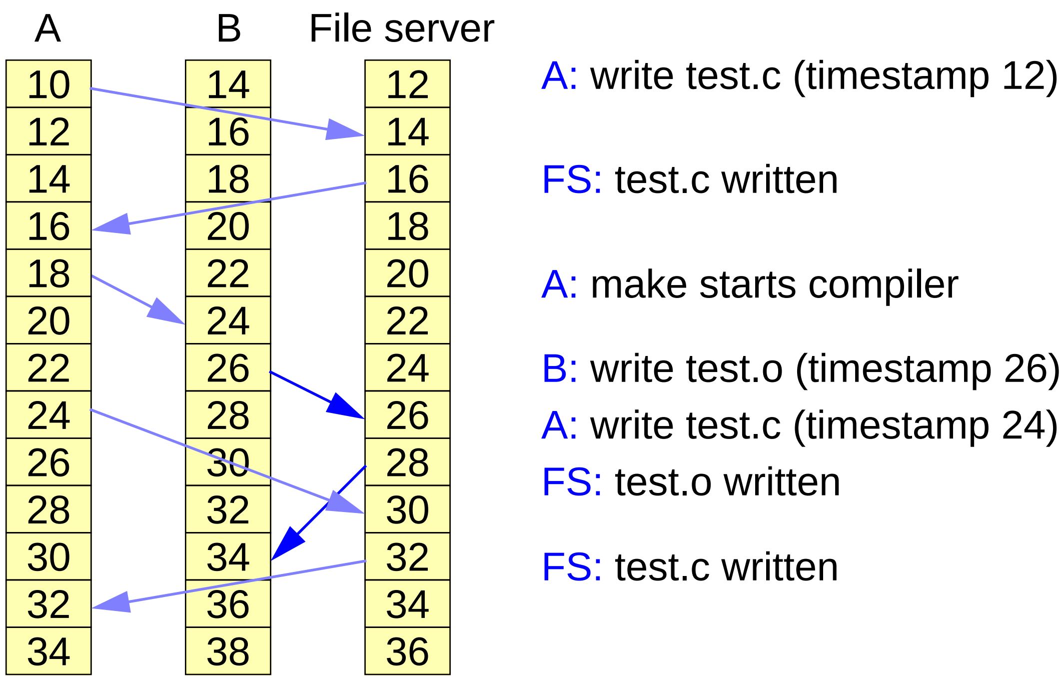


without logical clocks | with logical clocks

  - ◆ Logical clocks help to figure out an order of the log entries that reflects causality

1 Logical Clocks (5)

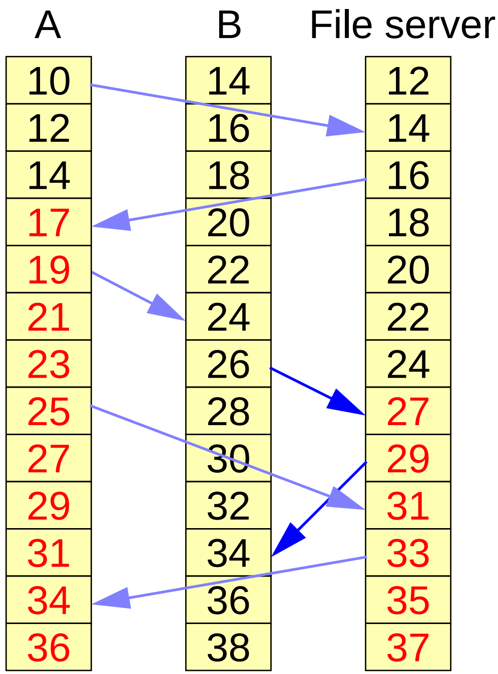
- Does it help for the "make" example?



without logical clocks

1 Logical Clocks (6)

- Does it help for the "make" example?



with logical clocks

A: write test.c (timestamp 12)

FS: test.c written

A: make starts compiler

B: write test.o (timestamp 26)

A: write test.c (timestamp 25)

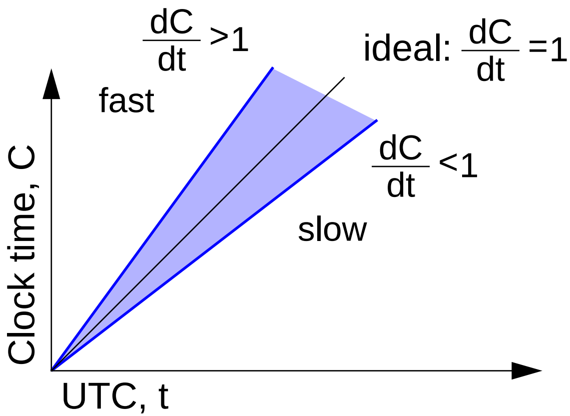
FS: test.o written

FS: test.c written

- ◆ NO!!

2 Clock Synchronization

- Local clocks are realized by software
  - ◆ Interrupts by timer chips count ticks
  - ◆ Local clock has a drift to UTC (Universal Coordinated Time)



  - ◆ Synchronize local clocks to minimize drift to UTC
  - ◆ Sources: DCF77, GEOS, GPS, Atomic clock

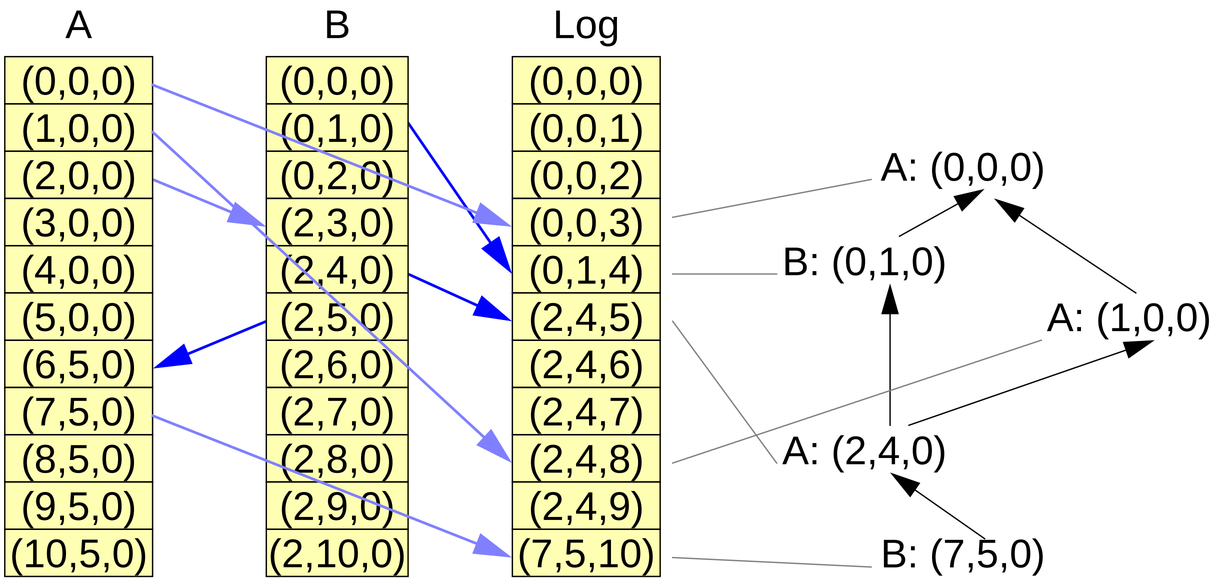
3 Vector Time

- Sometimes we would like to know whether two events are causally dependent by looking at their timestamps
  - Corresponds to reverse clock condition
  - Impossible to derive with logical clocks

- Vector time introduced by Mattern (1989)
  - Each process i of k processes maintains a clock vector V_i of k clocks
  - Local event: $V_i[i] := V_i[i] + 1$
  - Send event : $V_i[i] := V_i[i] + 1$; send(message, V_i)
  - Receive event: $V_i[i] := V_i[i] + 1$; receive(message, V_S);
    $\forall j : V_i[j] := \max(V_i[j], V_S[j])$
  - Comparing two time vectors:
    - $a \leq b :\Leftrightarrow \forall i : a[i] \leq b[i]$
    - $a < b :\Leftrightarrow (a \leq b) \wedge (a \neq b)$
    - $a \,||\, b :\Leftrightarrow \neg (a < b) \wedge \neg (b < a)$

3 Vector Time (2)

Example: Logging Processes

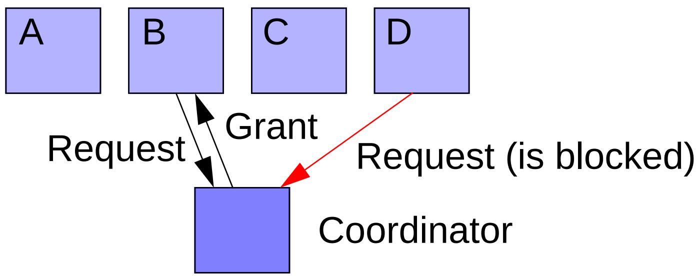


- From the log we can identify causality of all logged events

4 Mutual Exclusion

- Semaphore needs coherent shared memory
  - ◆ Multicomputers cannot use a semaphore

- Centralized semaphore server and request-reply interaction
  - ◆ Centralized component (coordinator) acts like a semaphore
  - ◆ Every process has to contact the coordinator to get access to a critical region



  - ◆ Process B sends a release message to the coordinator after leaving the critical region
  - ◆ Single point of failure

4 Mutual Exclusion (2)

- Distributed algorithm
  - ◆ Lamport (1978)
  - ◆ Improved by Ricart and Agrawala (1981)

- Algorithm by Ricart and Agrawala
  - ◆ Total ordering of events
    - Lamport's logical clock value plus process ID (time, pid)
    - The tuple makes timestamps of different events different and comparable
  - ◆ Group of processes that may enter a critical region
  - ◆ Process that wants to enter the region has to send a message to all others:
    - group_send(LC, pid)
    - Send must be reliable
    - Process waits till all other group member grant permission to enter the critical region

4 Mutual Exclusion (3)

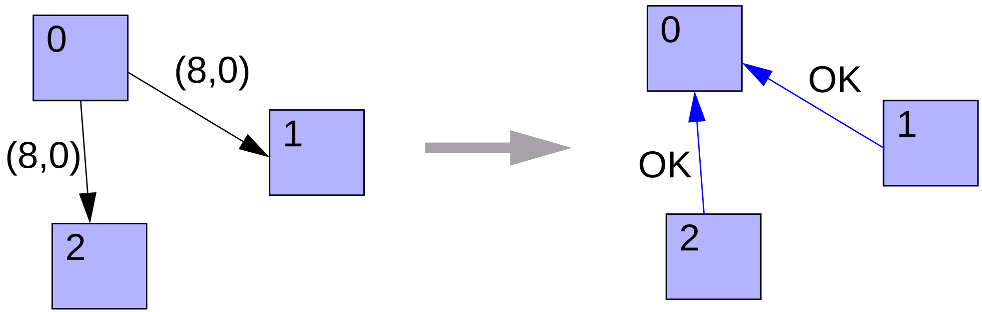
- If a process receives a message it does the following:
  - The receiver is not in the critical region and does not want to enter it:
    send(OK) to the original sender
  - The receiver is in the region:
    the message is enqueued
  - The receiver is waiting for entering the critical region:
    The receiver compares the timestamps of the incoming message with the timestamp of its own request message

    The own timestamp is lower:
    the message is enqueued

    The own timestamp is higher:
    send(OK) to the original sender
- After leaving a critical region a process sends back an OK for all enqueued request messages and deletes those messages

4 Mutual Exclusion (3)

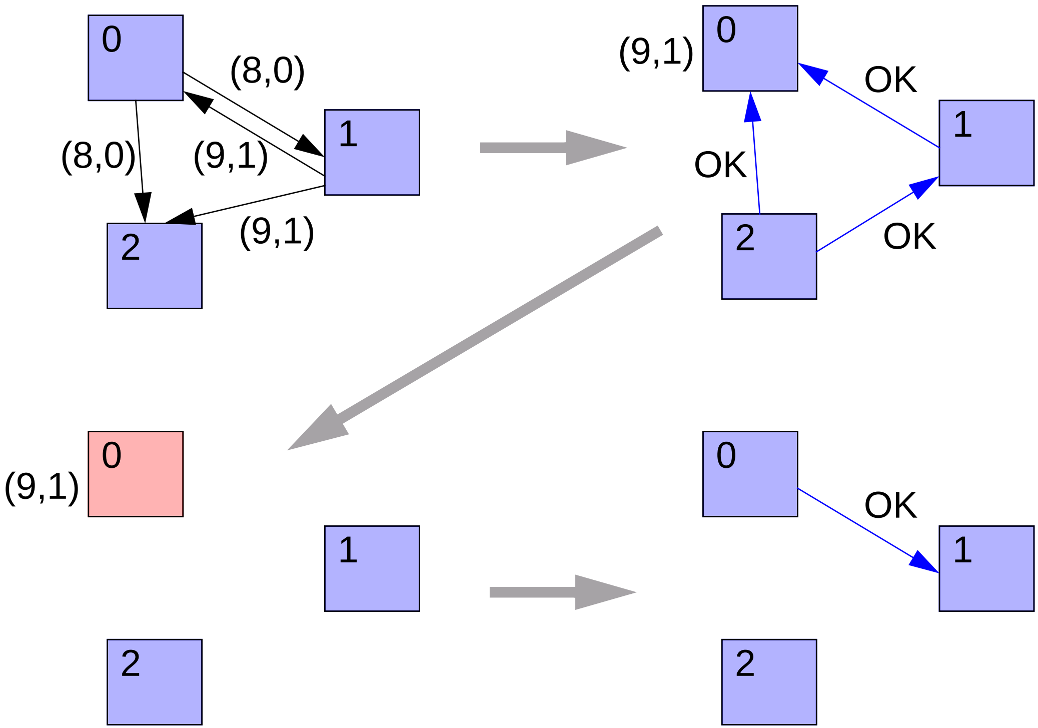
- No conflict: it clearly works



- ◆ The sender immediately gets OKs
- ◆ No further messages are sent or enqueued

4 Mutual Exclusion (4)

- Two processes want to enter the critical region at the same time



- The process with the lowest timestamp will win

4 Mutual Exclusion (5)

- Is it really better?
  - n points of failures
  - 2(n – 1) messages
  - Group membership must be known to all other processes
- Hardly better than the centralized version
  - Shows that it is possible to solve the problem by a distributed algorithm
  - Good example for distributed algorithms

5 Election Algorithms

- Problem
  - ◆ Find out a (new) coordinator, initiator, sequencer, or something similar
  - ◆ After the run of the algorithm
    - one group member should be the coordinator,
    - all other group member should know who was elected.
  - ◆ Multiple processes may start the election, but only one process will be elected.

6 Deadlock Detection

- Problem
  - ◆ Find out whether some processes are involved in a deadlock
  - ◆ Traversing the distributed dependency graph

7 Distributed Garbage Collection

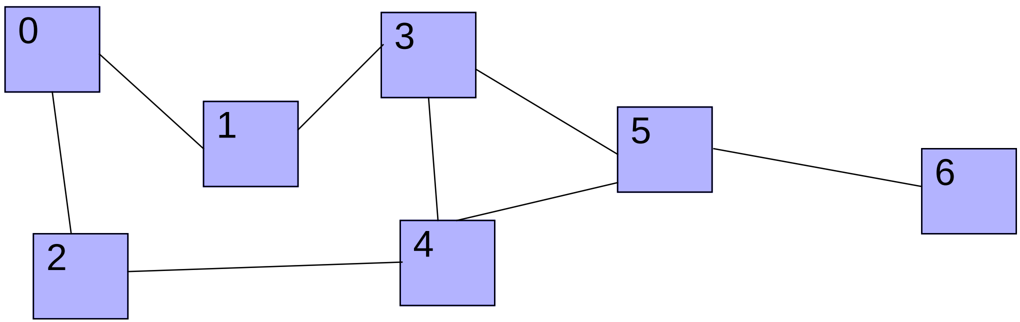
■ Problem

- ◆ Find out data object that are not referenced any more
- ◆ Traversing the distributed reference graph

8 Echo Algorithms

■ Problem

- ◆ Distributed information to all of not fully interconnected processes and compute a function (e.g. maximum of the output of all processes)



D.5 Object-Based Distributed Systems

- So far: processes
  - Processes and message passing
  - Processes and remote procedure calls

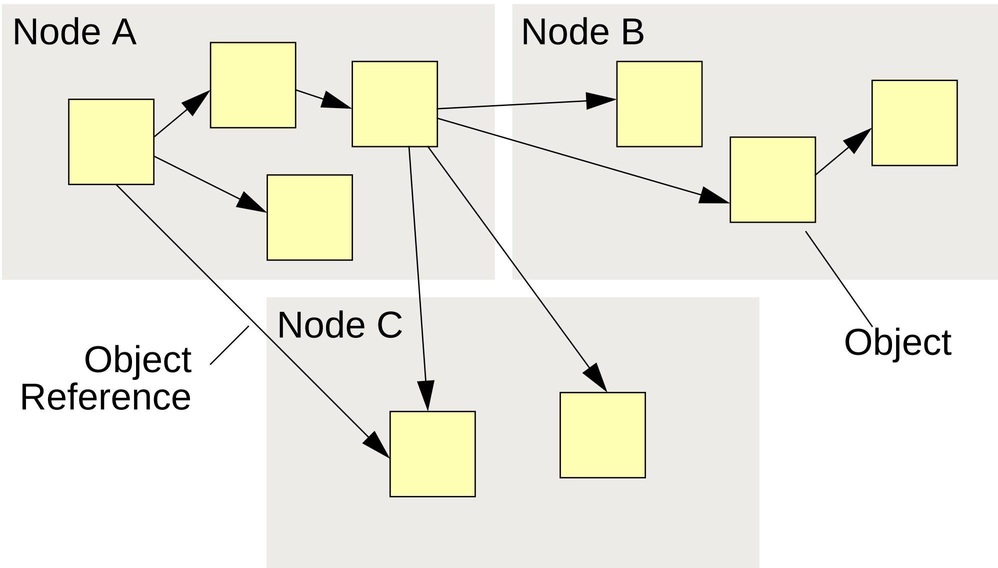
- Object-based programming
  - Objects
  - Classes
  - Methods, method invocation

  - Inheritance (object-oriented programming)

★ Systems that are distributed and object-based

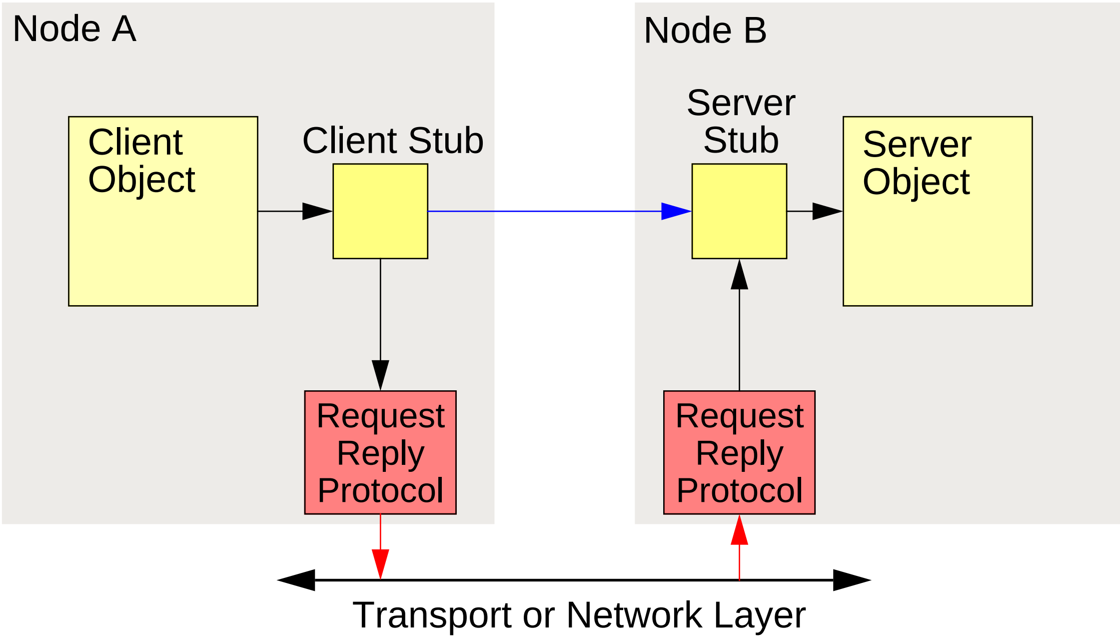
1 Whole Object Approach

- Objects as distributable entities
  - ◆ Objects are distributed on several nodes
  - ◆ Objects communicate with each other
  - ◆ Remote method invocation



1 Whole Object Approach (2)

- Implementing remote method invocation
  - ◆ Stub objects similar to stub procedures

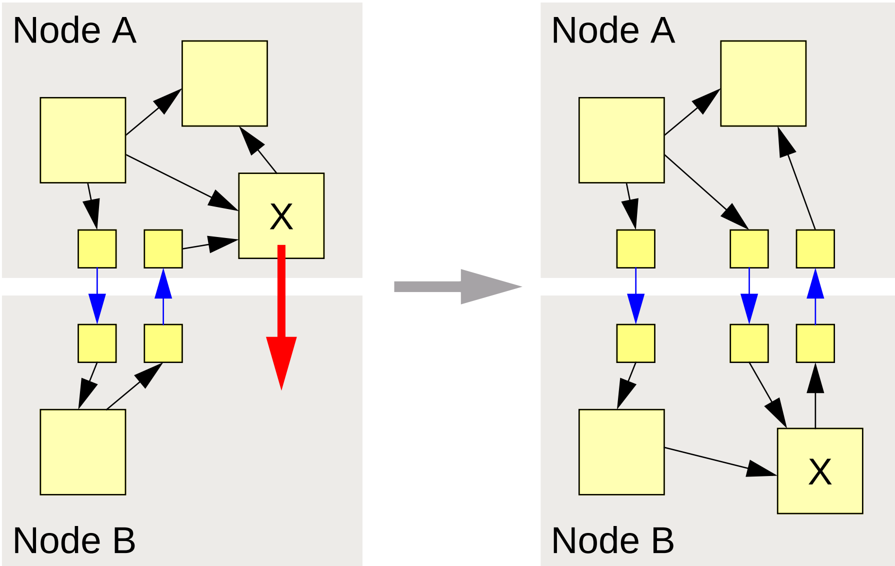


  - ◆ Client-stub object represents server object at client’s node

1 Whole Object Approach (3)

Object mobility

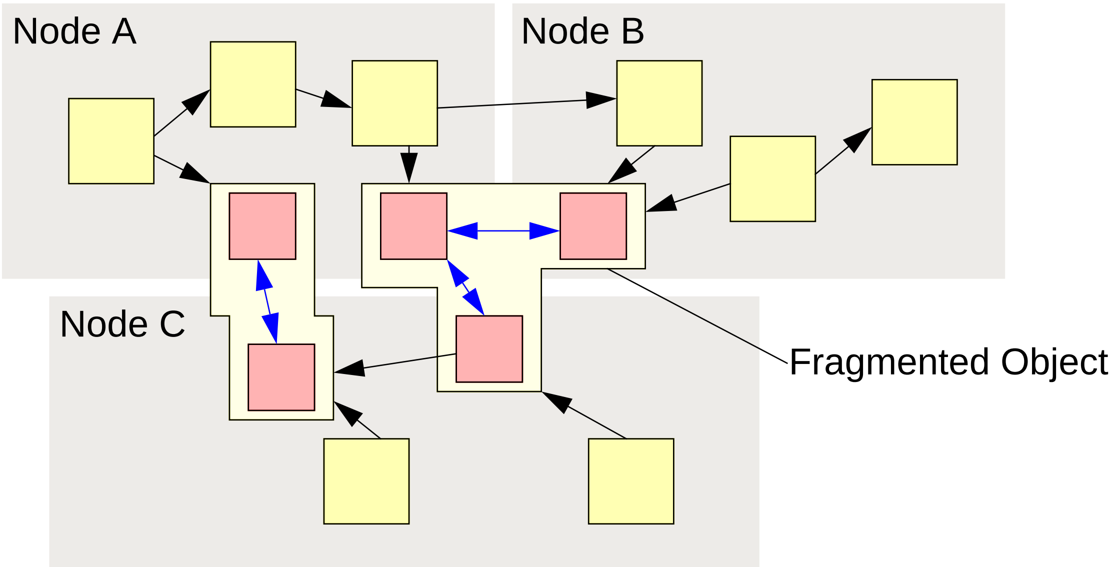
- Objects may migrate from one node to the other



- Stubs are to be created for all references of the moved object

2 Fragmented Object Approach

- Distributed objects consist of fragments that can be spread over multiple nodes
  - ◆ Fragments communicate with each other
  - ◆ Method invocation is always done locally



2 Fragmented Object Approach (2)

★ Advantages

- ◆ More general; includes the whole object approach
  - one fragment is the main object
  - other fragments are stubs
- ◆ Arbitrary communication between fragments
  - group communication for fragments replicating the object’s state
  - real-time or transactional communication
  - communication with the object is always local
- ◆ "Intelligent stubs"
  - local fragment can replicate or cache data of the object
  - local fragment can compute methods that do need little of the object’s data

2 Fragmented Object Approach (3)

▲ Disadvantages

◆ Programmer has to build up the object-internal communication by his own

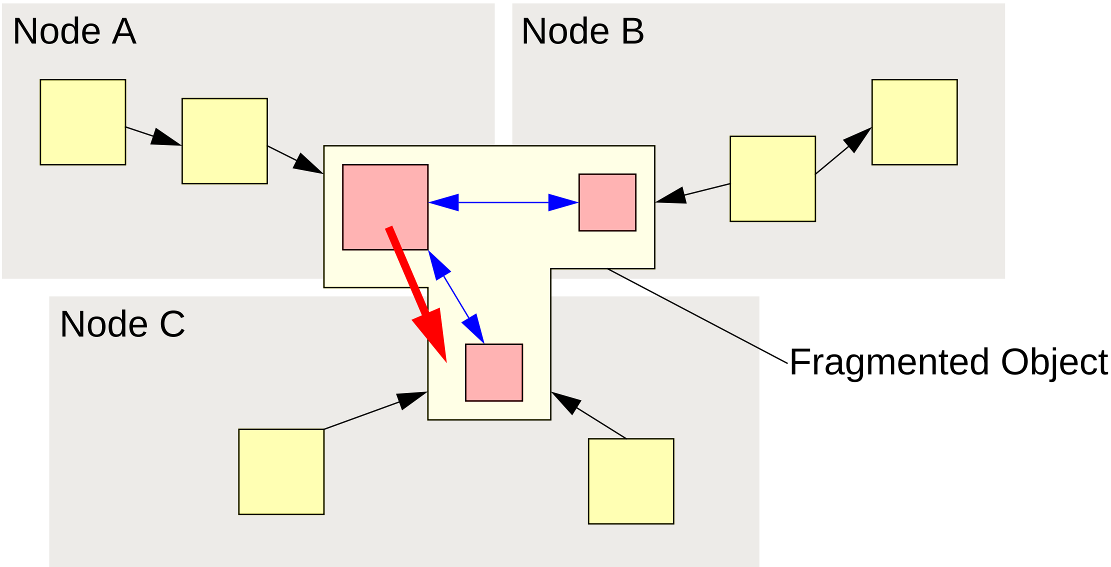
- tools and libraries may help (e.g., stub fragment generator)
- special name services may be needed

◆ System does not know about stubs

- Somehow, the system has to load the fragment code from somewhere whereas it otherwise only has to generate a stub.

2 Fragmented Object Approach (4)

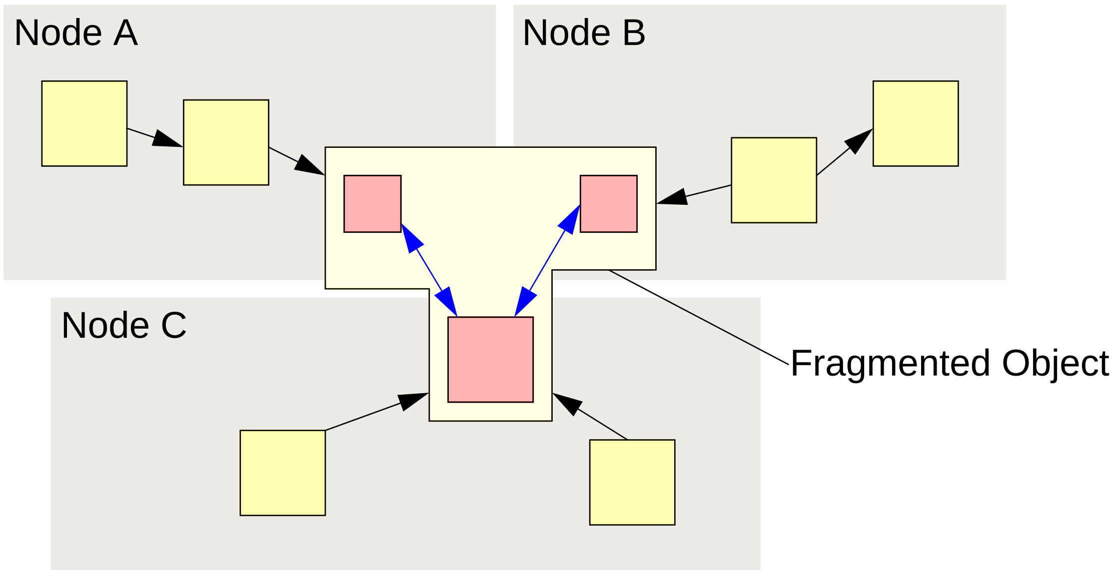
- Object mobility
  - Mobility is relative because the object is always accessed via a local fragment
  - Fragments may be mobile: fragments need to be replaced by one another



2 Fragmented Object Approach (5)

■ Example:

- A new main fragment is built up at the side of stub fragment, takes over the essential data from the old main fragment, and replaces the stub.
- The old main fragment is replaced by a new stub fragment



E Distributed Objects in CORBA

★ Motivation

Location

- Transparency of location

Heterogeneity

- Different hardware,
- different operating systems, and
- different programming languages are used in distributed systems.

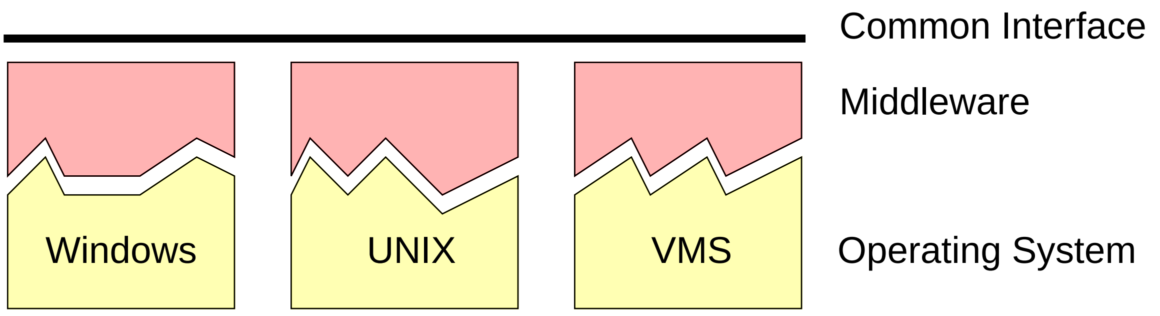
- Transparency of heterogeneity

Services

- Name server
- Time server
- ...

E.1 Middleware Approach

- Middleware: a piece of software between operating system and application



◆ Middleware provides services for distributed programming

- **We need:** middleware for distributed object-based programming

★ CORBA – Common Object Request Broker Architecture,
a standard of the OMG

E.2 CORBA Overview

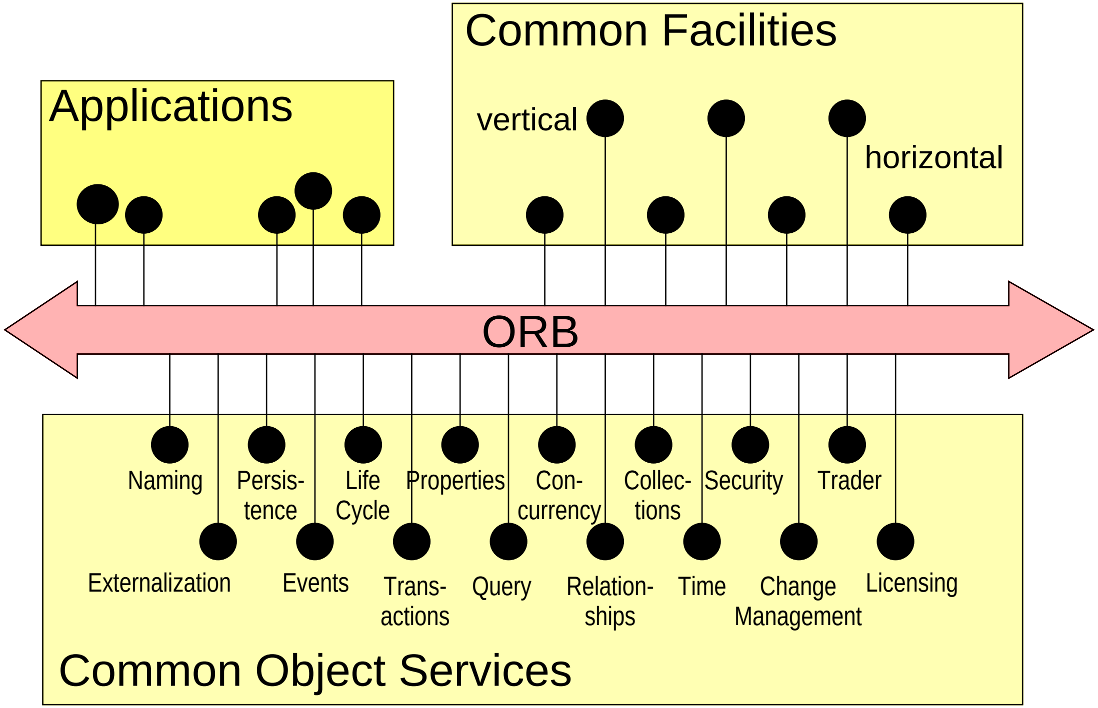
1 Design Goals

- CORBA is a standard
  - There are standard documents
  - Definition of CORBA compliance
  - Vendors build implementations of the standard (e.g., VisiBroker, OmniBroker, Orbix, etc.)
- CORBA wants to abstract from
  - hardware,
  - operating system, and
  - programming language.

1 Design Goals (2)

- Interoperability
  - ◆ An application shall run on different CORBA implementations without major changes
  - ◆ Applications on different CORBA implementations shall be able to communicate
- Embedding of legacy applications
  - ◆ Legacy applications can be encapsulated and can act as CORBA objects
- Vision of business objects or components
  - ◆ CORBA objects represent all kind of data and proceedings of a company

2 Architectural Overview



- Object Management Architecture – OMA

2 Architectural Overview (2)

- Application objects
  - Whole object approach for distributed objects
  - Client/server architecture
- Object Request Broker (ORB)
  - Communication backbone
  - Enables objects to communicate
- CORBA Services
  - Basic services for distributed programming
  - Extends the ORB’s functionality
  - Services look like objects
- CORBA Facilities
  - Application-specific services

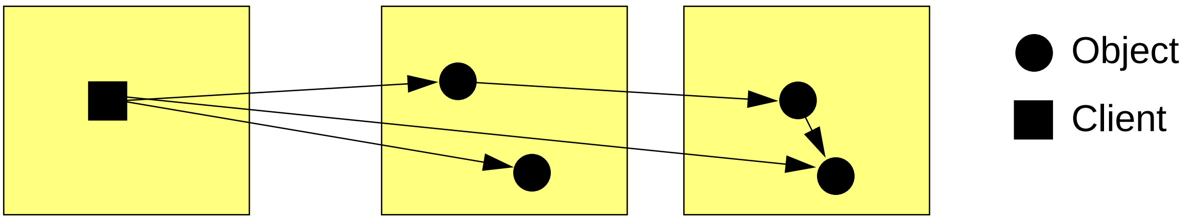
3 CORBA Implementations

- A CORBA implementation must contain
  - the implementation of the core architecture
  - one language mapping (e.g., for C++)
- A CORBA implementation may contain
  - an arbitrary number of services
  - functionality for interoperability with other CORBA implementations (GIOP/IIOP)
- The implementation is free **how** it realizes the requirements of the standard
  - Many different implementations are possible
    - Daemon-based
    - Library-based
    - ...

E.3 Application Objects

1 Distributed Object

- ◆ Identity
- ◆ State
- ◆ Operations (methods)
- ◆ CORBA objects can be invoked (implement server side)
- ◆ CORBA objects may act as clients



■ Clients need not to be CORBA objects

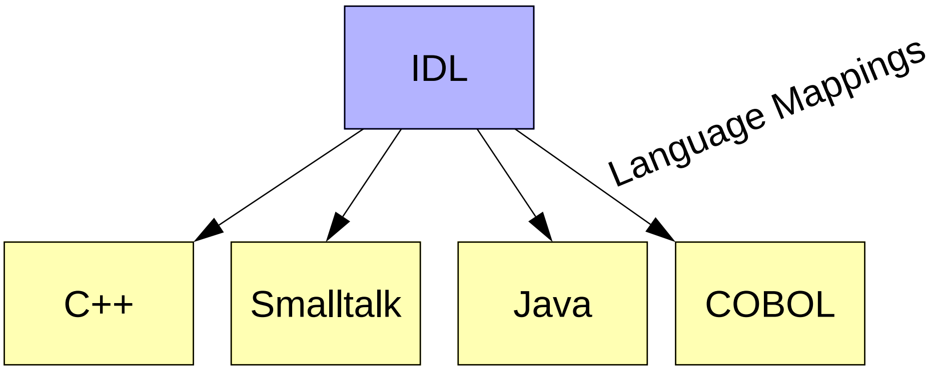
- ◆ CORBA clients can be processes, etc.

2 Distributed Object (2)

- Distributed objects form an application
  - Objects cooperate by communicating with each other
- ★ Example: printer management system
  - Client objects
  - Spooler objects
  - Printer objects
- "Whole object approach"
  - Client-side stub can contact the server object

3 Interface Definition Language (IDL)

- Language for describing object interfaces
  - Independent of implementation language of an object
  - Language mapping defines how IDL primitives are mapped to the concepts of a specific programming language
  - Language mapping is part of the CORBA standard
  - Language mappings exist for C, C++, Smalltalk, COBOL, and Java
  - IDL is similar to C++



3 Interface Definition Language (2)

- Example

```
module MyModule
{
    interface MyInterface
    {
        attribute int lines;
        void printLine( in string toPrint );
    };
```

IDL to C++

```
namespace MyModule {
    class MyInterface : ... {
        public:
         virtual CORBA:Long lines();
         virtual void lines( CORBA::Long _val );
         void printLine( const char *toPrint );
        ...
    };
```

3 Interface Definition Language (3)

- Example

```
module MyModule
{
    interface MyInterface
    {
        attribute int lines;
        void printLine( in string toPrint );
    };
```

IDL to Java

```
public interface MyModule.MyInterface extends ... {
    public int lines();
    public void lines( int lines );
    public void printLine(java.lang.string toPrint );
    ...
};
```

```
nam
        void PrintHello( char *toPrint );
        ...
    };
```

3 Interface Definition Language (4)

★ Advantages

- ◆ Transparency of implementation language
- ◆ Enables interoperability

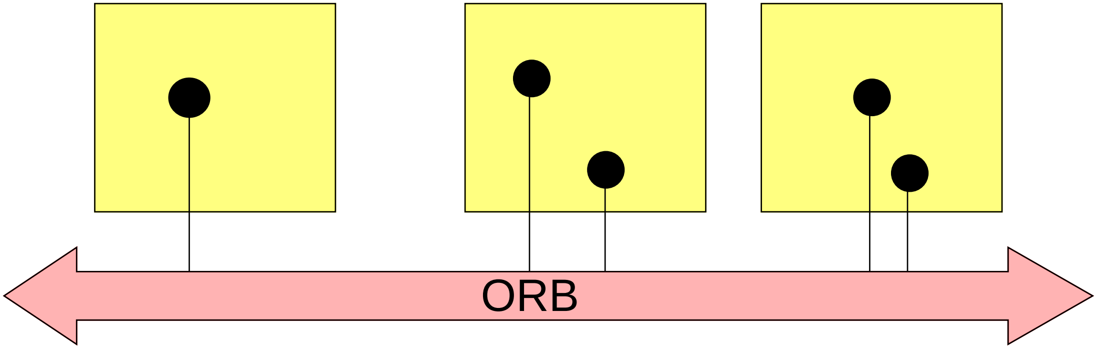
▲ Disadvantages

- ◆ Object interface must be defined in the target language **and** in IDL
- ◆ IDL is quite expressive
  - Language that do not provide the right mechanisms will have a complex language mapping
- ◆ If a language has special features they cannot be used because they are not part of IDL

4 Process of Creation and Binding

- Creation of a server object
  - Description of the object interface in IDL
  - Programming the server object in an implementation language
  - Registration of the object at the ORB (the OA respectively)
    - The ORB creates an Interoperable Object Reference (IOR)
- Binding to the server object at client side
  - Retrieval of an object reference to the server
    - Result of a name-server query
    - Return parameter of a method call
    - Retrieval from outside of the system: user knows the reference as a string (IORs can be converted to strings and back)
  - Creation of the client stub
  - Method invocations using the stub object

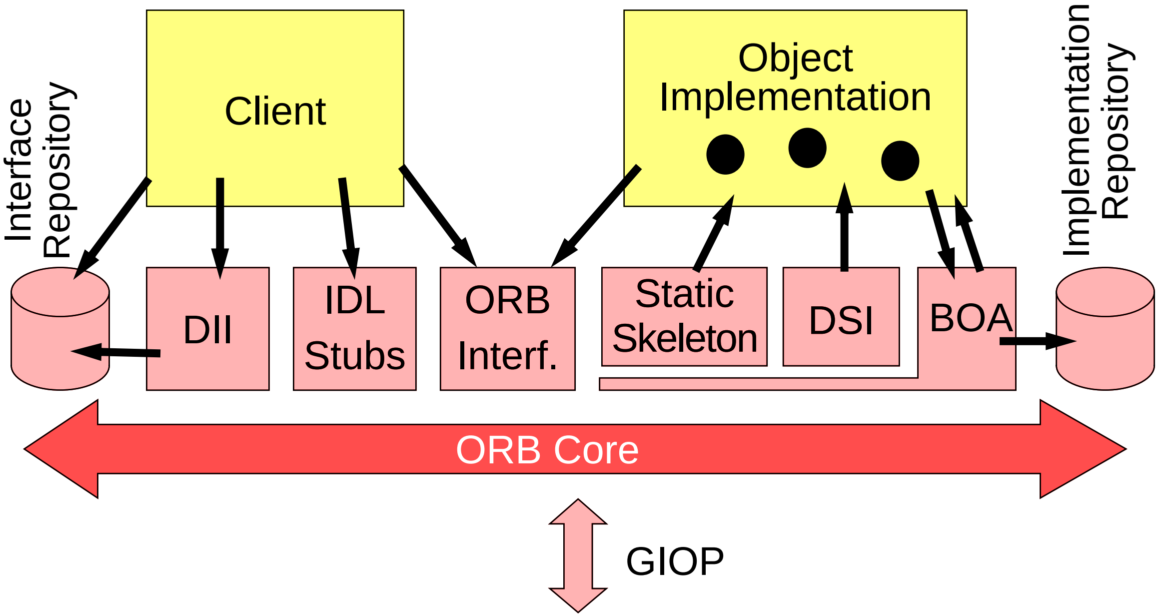
E.4 Object Request Broker – ORB



- The ORB is the communication backbone of a CORBA implementation
- All communication is handled by the ORB
  - ◆ ... within an address space / process
  - ◆ ... between address spaces / processes
  - ◆ ... between address spaces / processes of different ORBs
- The ORB implements location transparency

1 Architecture

- Central components of an ORB



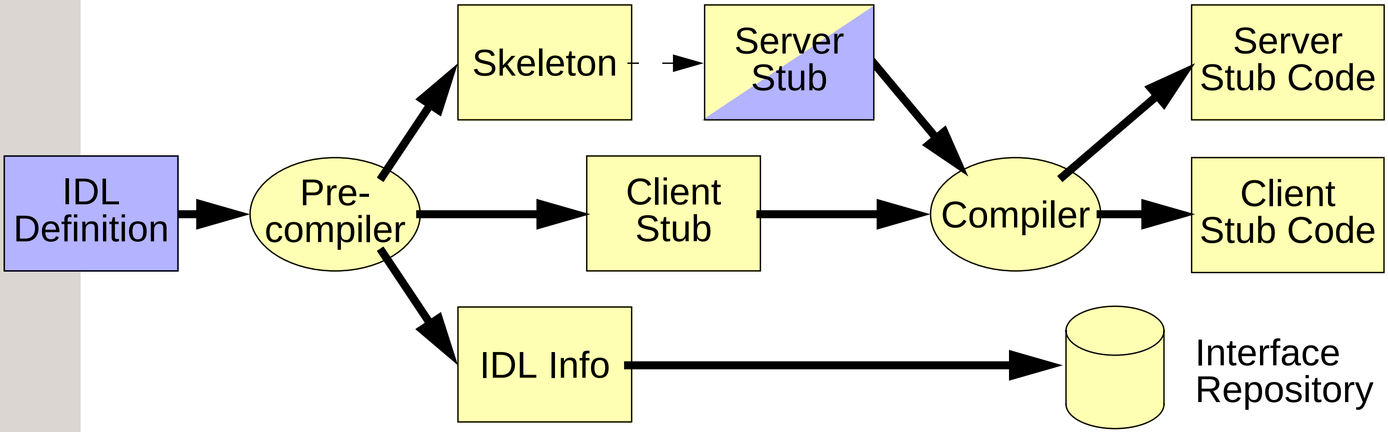
2 Static Stubs

- Stubs on client and server side
  - As soon as we exactly know the interface of an object, we can create static stubs from it.
  - Static stubs are automatically created from the IDL description
  - Server side stubs are called *skeletons*
    - CORBA skeletons have to be filled with the actual implementation

- Tasks of the stubs
  - Marshalling of parameters
  - Send and receive of request and reply messages using the ORB core

2 Static Stubs (2)

- Stub creation



- From IDL, a precompiler generates a number of output files.
- The skeleton contains a frame for filling in the actual methods of the object implementation. The skeleton already contains the server-stub functionality.
- The client-stub file contains the code for the client stub.
- An additional file may contain data for the interface repository.
- Usually the stubs have to be compiled by the target-language compiler.

3 Interface Repository

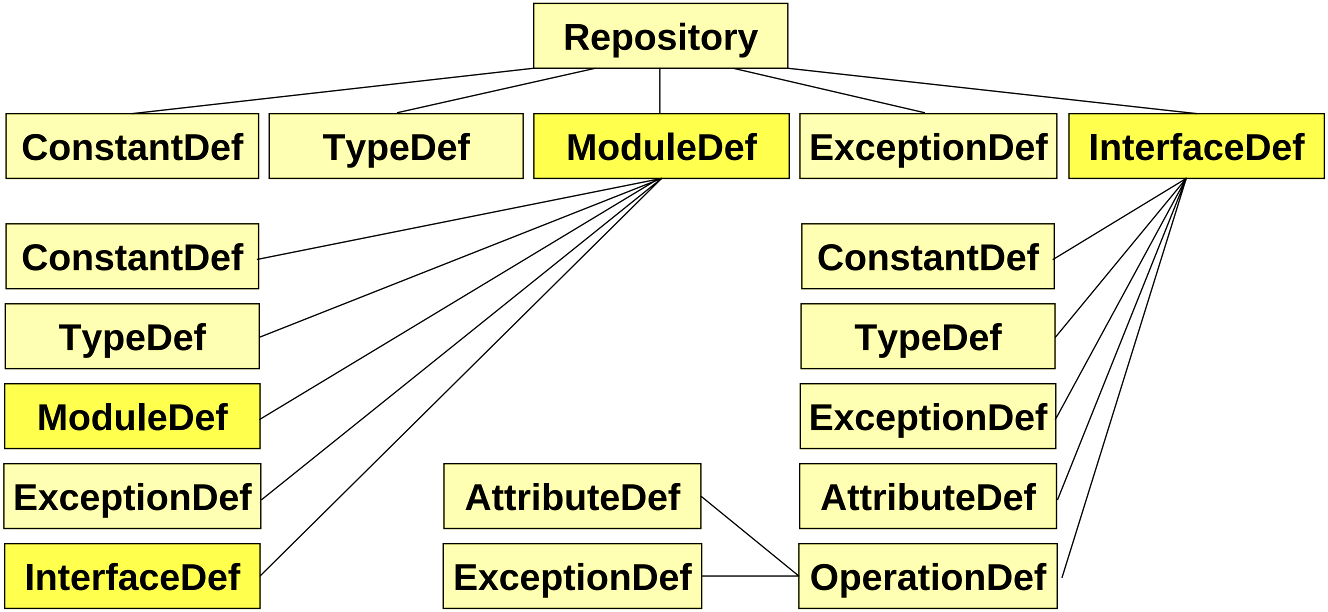
- Data base for interface definitions in IDL

- Query the data base for
  - ◆ Type checking
    - Does the stub's type match the object's type?
    - Inter-ORB operations: insertion into multiple repositories
  - ◆ Retrieval of meta data by clients and tools
    - Dynamic invocations
    - Debugging
    - Class browser
  - ◆ Implementing method *get_interface* of each object

- Writing to the data base using
  - ◆ IDL compiler
  - ◆ Write methods

3 Interface Repository (2)

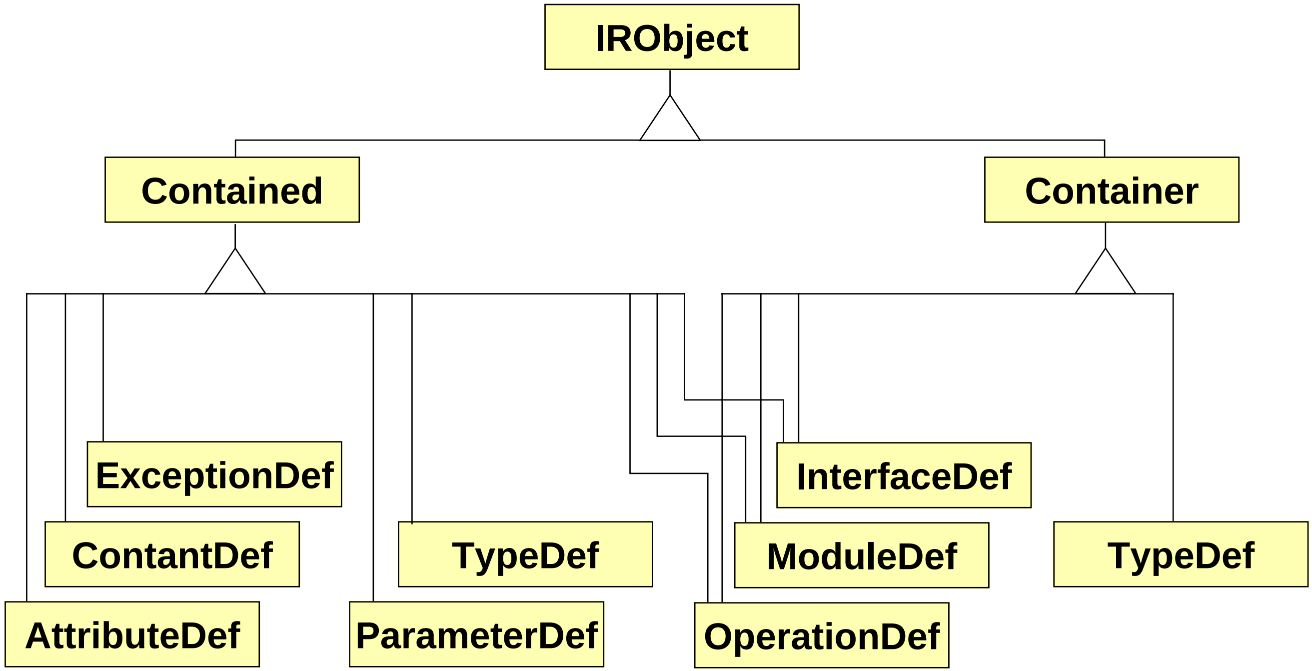
- Stored information / composition of an IDL file



◆ Hierarchy of the components of an IDL file, and of the interface repository respectively

3 Interface Repository (3)

- Inheritance hierarchy of the IDL interfaces of IR objects



3 Interface Repository (4)

- Standard types are represented by so-called *type codes*

TypeCode

  - Representation of basic types: **int, float, boolean**
  - Representation of composite standard types: **union, struct, enum**
  - Representation of template types and complex declarators: **sequence, string, array**

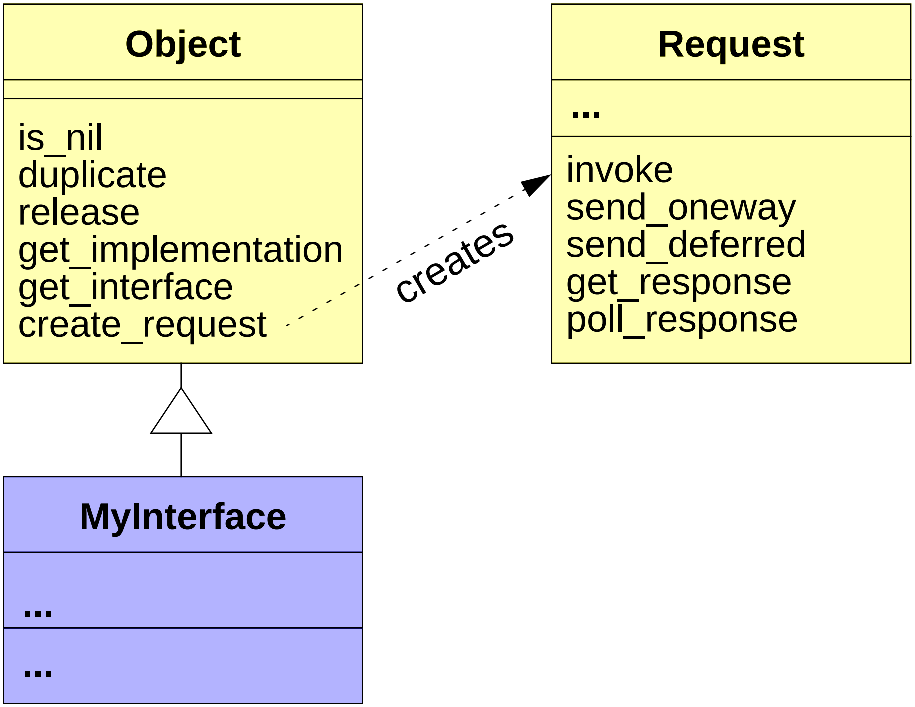
- Type codes represent type and structure as objects with an standard IDL interface
  - Operation for comparing type codes
  - Operation for obtaining the description of the type

4 Dynamic Invocation Interface (DII)

- DII allows the invocation of methods whose object interface was not known at compile time
  - ◆ Interface description can be retrieved using the interface repository
- Single steps of an invocation (which have to be mapped by the language binding)
  - ◆ Retrieve signature of the method from the interface repository
  - ◆ Create list of parameters
  - ◆ Compose request
  - ◆ Invoke method
    - – with RPC (synchronous call)
    - – with asynchronous RPC (asynchronous call)
    - – with datagramm message (no reply, no reliability)

4 Dynamic Invocation Interface (2)

- IDL definitions for the DII



5 Dynamic Skeleton Interface (DSI)

- DSI allows a CORBA server to accept method invocations for objects whose interfaces are only dynamically retrievable, e.g., in
  - Bridges to other ORBs
  - CORBA-encapsulated data bases
  - Dynamically created objects and interfaces
- The DSI identifies the object for which the invocation is made for
  - A call-back function is invoked that gets all the necessary information
  - The call-back function has to invoke the corresponding object

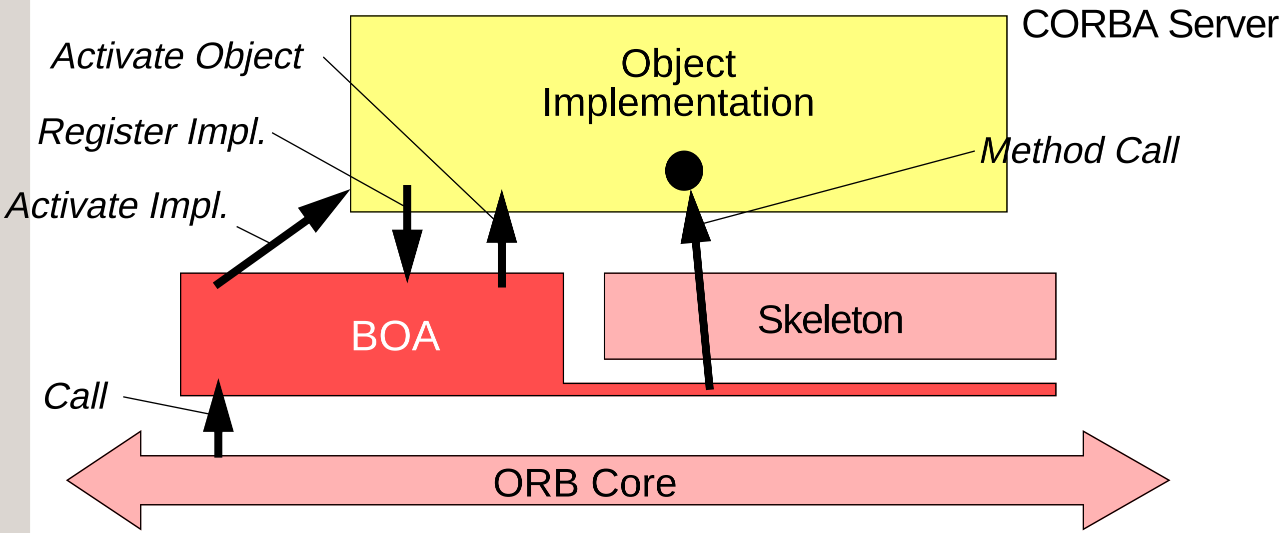
★ **Note:**
DII and DSI are not distinguishable from static stubs, i.e., they are fully interoperable with static stubs.

6 Object Adaptor

- The object adaptor is a local representative of ORB services
  - ◆ It generates object references (for new objects)
  - ◆ It maps object references to implementations
  - ◆ It forwards incoming method calls
  - ◆ It authenticates the caller (implementing CORBA's security functionality)
  - ◆ It activates and deactivates the object and its implementation
  - ◆ It registers server classes within the implementation repository
- CORBA defines the mandatory Basic Object Adaptor
  - ◆ Basic functionality
- Another example: an OODB Adaptor
  - ◆ Connecting an object-oriented data base to CORBA
  - ◆ The OODB Adaptor converts all objects in the data base to CORBA objects

Basic Object Adaptor (BOA)

- Activation and deactivation of CORBA servers
  - Implementation of an objects is not always active in the sense of the operating system



- **Note:** Do not mix up server objects with CORBA servers.

7 Basic Object Adaptor (2)

- Activation strategies

◆ **Shared Server**

A server process contains multiple objects of an implementation

◆ **Persistent Server**

Activation strategy is realized outside of the BOA; server is always running (persistent); server process hosts multiple objects of an implementation

◆ **Unshared Server**

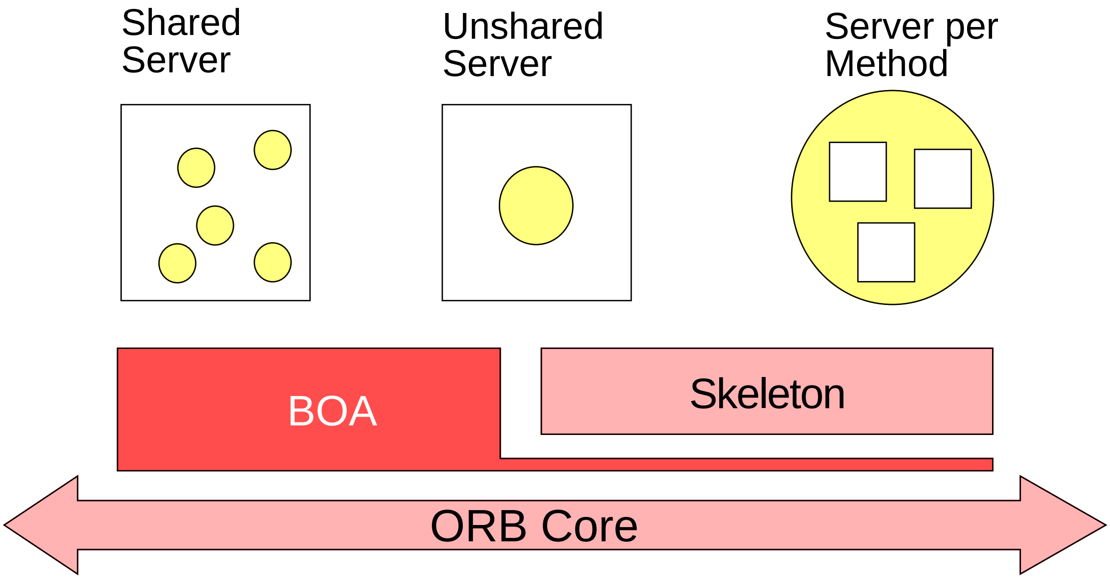
One server process per object instance

◆ **Server per Method**

One server process per method invocation

7 Basic Object Adaptor (3)

- Activation strategies



- Circles are objects
- Squares are CORBA servers

8 Implementation Repository

- Data base for object implementations
  - Data to localize and activate object implementations
  - Information for debugging and administration
  - ...
- Implementation repository is highly implementation dependent.
- Query the data base
  - Implementation of the method *get_implementation* of each object
- Writing the data base by
  - External tools
  - The BOA at the start of a server

9 Inter-ORB Communication

- GIOP – General Inter-ORB Protocol
  - Basic protocol for the interaction of two ORBs
  - Common Data Representation (CDR) converts IDL-compliant parameters into a serial byte stream
  - IIOP – Internet Inter-ORB Protocol
(GIOP over TCP/IP; implementation is mandatory)
  - Other implementations of GIOP are possible
- ESIOP – Environment Specific Inter-ORB Protocols, e.g., DCE/ESIOP
  - Inter-ORB protocol on the basis of DCE RPC
- IOR – Interoperable Object Reference
  - String representation
  - Several profiles (one of it is IIOP)

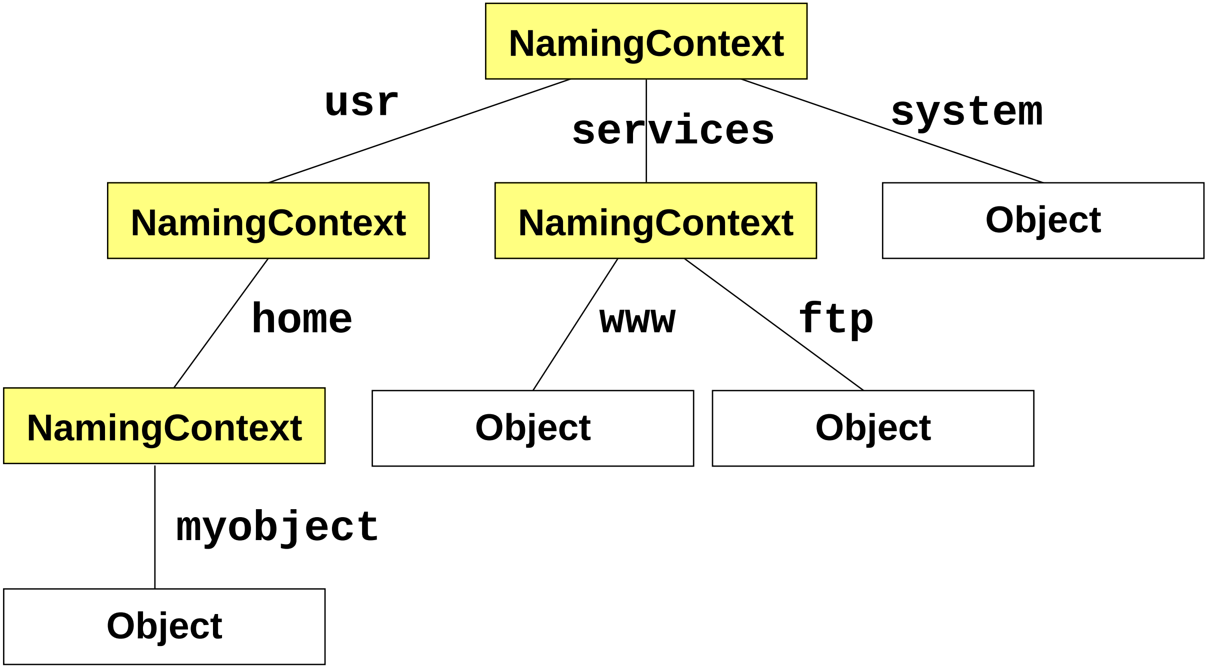
E.5 CORBA Services

- Basic services of a distributed system as extensions of the ORB
  - Naming service
  - Transaction service
  - Persistent object service
  - Life cycle service
  - ...
- Services are accessible by IDl interfaces
  - No additional primitives necessary; method invocation is enough

1 Naming Service

- CORBA defines an hierarchical naming service similar to the UNIX file system
  - Name space forms a tree
  - Names consist of multiple components (syllables)
    e.g., < "usr"; "home"; "myobject" >
    (UNIX: "/usr/home/myobject")
  - Interface to the naming service is defined by IDL

- Example of a name tree



1 Naming Service (3)

- Interface of a Naming Context and the Binding Iterator

NamingContext
resolve list destroy new_context unbind bind rebind bind_context rebind_context bind_new_context

BindingIterator
next_one next_n destroy

- ◆ Naming Context acts like a directory
- ◆ Objects or other contexts can be bound to a name
- ◆ Listing the context returns an array and an iterator that allows you to look at the next entries

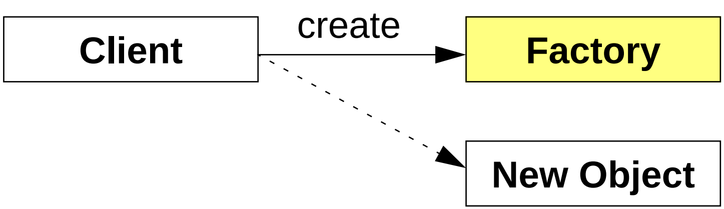
2 Life Cycle Service

- Controls the life cycle of an object
  - Creation
  - Copying, Migration
  - Deletion

- Life Cycle Service defines the common interface for controllable objects

★ Modell des Lebenszyklus

- Creation of an Object:

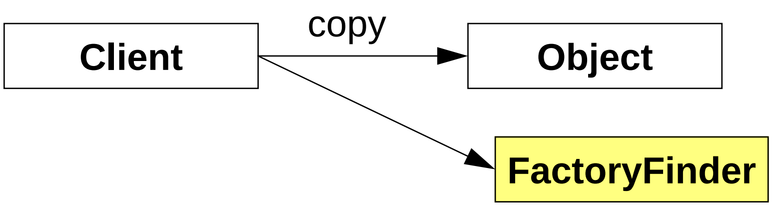


Call of a create method at a factory object

  - Protocol and interface of the factory is not defined and probably dependent on the kind of object to create

Life Cycle Service (2)

◆ Copying and migration:



FactoryFinder determines the location

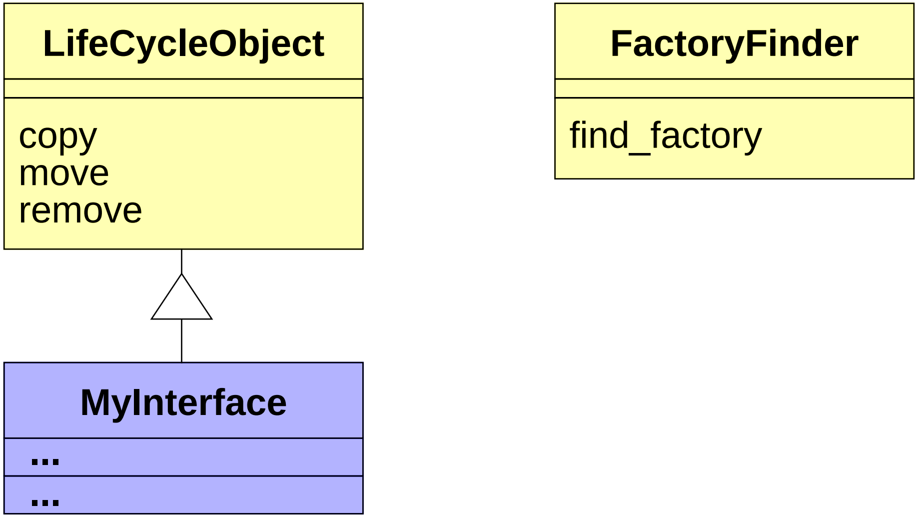
- Locations can be a specific computer or a group of computers, etc.

◆ Deletion:

- With the call of remove the object is deleted.

2 Life Cycle Service (3)

- Objects need to implement the LifeCycleObject interface



- copy and move need a FactoryFinder object to find a factory object which in turn will create the copy or migrated object

2 Life Cycle Service (4)

- Example: implementation of the copy meethod
  - ◆ Client invokes copy method and passes a reference to a FactoryFinder
  - ◆ The copy method is provided by the programmer **or** by the CORBA implementation
  - ◆ The copy methode calls the FactoryFinder and selects a suitable factory that is used to create a new object
  - ◆ The new object will be initialized with the data of the existing object

▲ To the outside: clearly defined interface

▲ To the inside: open and implementation dependent,
e.g., protocol between copy method and factory

- In CORBA 2.0 relationship between objects is considered for copying and mobility: part-of relationship, etc.
  - ◆ Parts of the object are also copied or migrated
(deep copy vs. shallow copy)

F OLE — COM & DCOM

But the darkest of dark sides is a thing Microsoft hastily concoted called OLE ... OLE is catching on like e-mail at Al Gore's house, but unfortunately it is the deadliest possible blow to OOT — mainly because it isn't object-oriented, it isn't fully specified, it's inferior to OpenDoc, and there is little anyone can do to stop it. OLE will be synonymous with objects.
(Ted Lewis, IEEE Computer Magazine, Dez. 1994)

F.1 References

Wan+97. Y. M. Wang et al.: DCOM and CORBA Side by Side, Step By Step, and Layer by Layer, http://akpublic.research.att.com/~ymwang/papers/HTML/DCOMnCORBA/S.html, C++ Report Magazine, September 1997.

MS97. The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm.

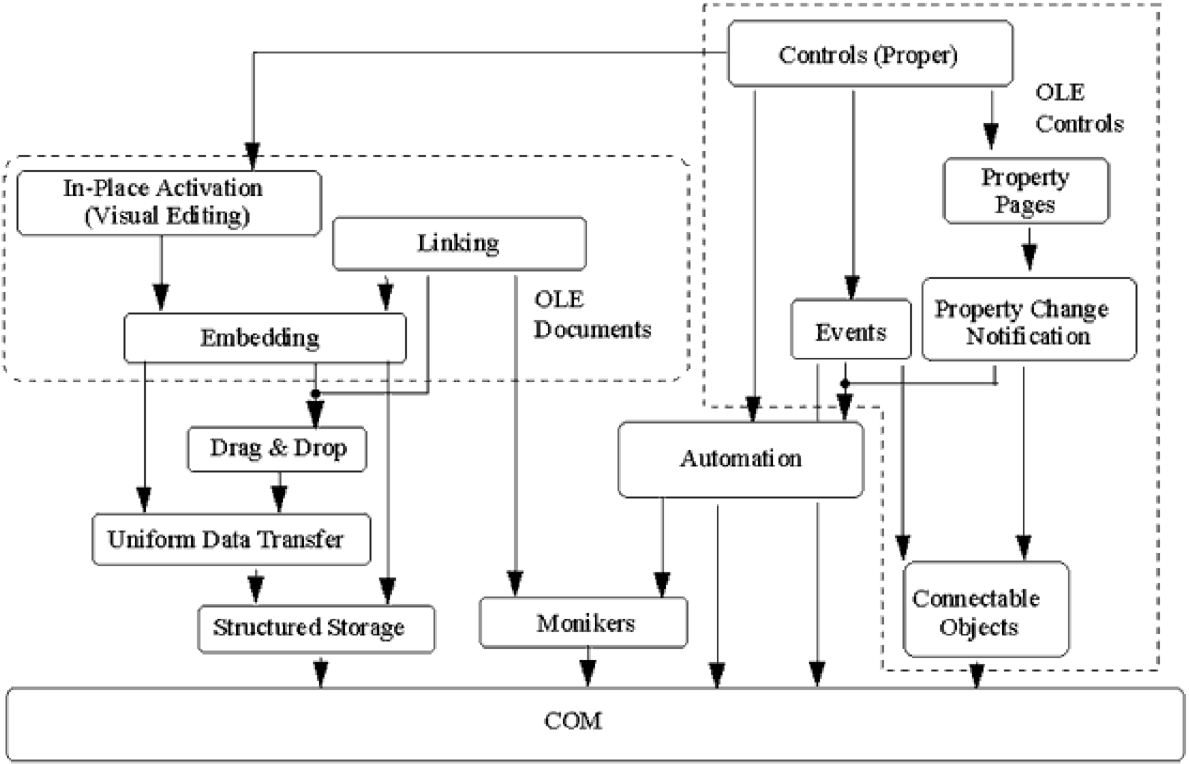
BrKi96. N. Brown, C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

Kirt97. M. Kirtland, The COM+ Programming Model Makes it Easy to Write Components in Any Language, Microsoft Systems Journal, December 1997, http://www.microsoft.com/msj/1297/complus2/complus2.htm.

F.2 Overview

- OLE: Object Linking and Embedding
  - ➥ Microsoft's "standard" for collaboration of software components
    - ➤ Text processor & spreadsheet calculator & drawing tool
    - ➤ Spreadsheet calculator & database
    - ➤ Scripts with applications
    - ➤ …
- COM: Component Object Model
  - ➥ An Object Request Broker for a single-machine environment
    - ➤ Object Bus & Object Services
- DCOM: Distributed COM
- ActiveX: COM enabled for the Internet (what ever that means)

F.3 OLE Architecture



from: Microsoft: COM Techncical Overview

F.4 COM IDL & COM ODL

1 IDL

- Interface definition language, used to create
  - ➤ client proxies
  - ➤ server stubs
  - ➤ C code for parameter marshalling
- MIDL Compiler
  - ➤ stub generator
- Comparison to CORBA IDL
  - ◆ no language binding
    - ➤ binary "standard" for vtable with pointers to methods
  - ◆ no multiple inheritance on interfaces

2 ODL

- Object description language
  - ◆ used to describe object interfaces for
    - ➥ Type Library (= Interface Repository)
- MKTYPLIB
  - ◆ utility to generate
    - ➤ interface metadate for type library
    - ➤ header file with function prototypes

F.5 COM: OLE's Object Bus

1 !!! "COM objects" are not objects

In COM, an object is a piece of compiled code that provides some service to the rest of the system. To avoid confusion, it is probably best to refer to an object used in COM as a COM component or simply as a component. This avoids confusing COM components with source-code OOP objects such as those defined in C++. (Microsoft: COM Tehnical Overview)

- As CORBA, COM separates the object interface from its implementation
  - ➤ separate interface description in IDL
  - ➤ one COM object can support multiple interfaces

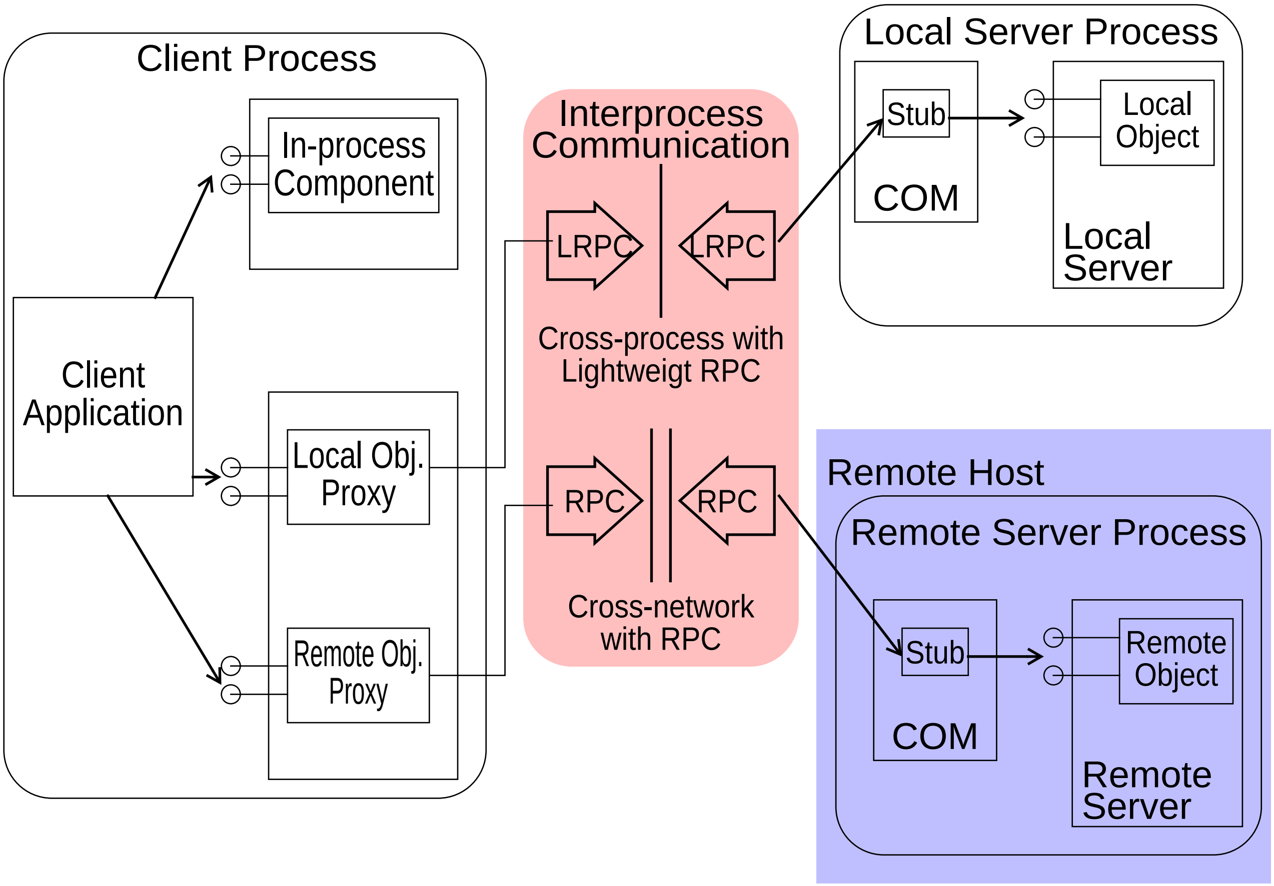
- COM interfaces are interfaces to services, not to "real objects"
  - ◆ COM objects have no unique id
    - ➥ COM objects are collections of functions
    - ➥ COM objects do not maintain state

2 COM server

- Like CORBA sever
  - creation and execution environment for COM objects
  - object creation via factory interface

- Flavors
  - ➤ In-process servers
  - ➤ Local servers
  - ➤ Remote servers

3 COM Client/Server Interaction



F.6 COM/OLE Object Services

- Interface negotiations
  - ➤ client initially gets only primary interface
  - ➤ client may request reference to another interface
  - ➤ if supported the target object returns reference to that interface
- Life cycle management
  - ➤ factories
  - ➤ reference counting & deletion of unused objects
- Component licensing
  - ➤ object instantiation depending on license files
- Event service (connectable objects)

F.7 OLE Automation

- Dynamic Invocation Interface = OLE Automation
  - OLE Automation allows a single program to control automation servers residing in many applications
  - automation server: interface objects of a scriptable application

F.8 OLE Uniform Data Transfer

- Clipboard transfer
- Drag&drop
- Links
  - ➤ notification mechanism

F.9 OLE: Structured Storage

➥ Persistence service

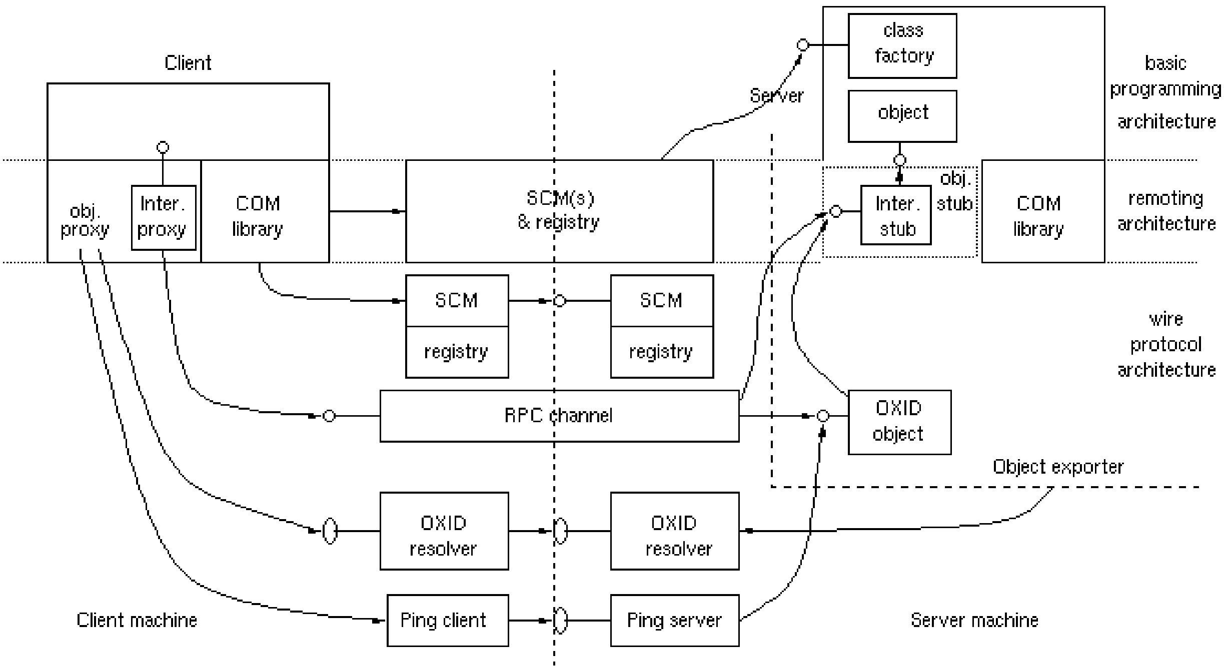
- Compound Files (future MS file system?)
  - ➤ transactional storage
- Persistent objects
  - ➤ can read and write themselves to storage
- Monikers
  - ◆ persistent intelligent names
    - ➤ naming service + interface to the objects behind it
  - ◆ implement name-related interfaces
  - ◆ names for files, tables, table-cells, ...
  - ◆ work-around for persistent objects
    (COM provides references to interfaces of services, not to objects,
    solution: moniker can instantiate an object an reestablish it's former
    state)

F.10 DCOM

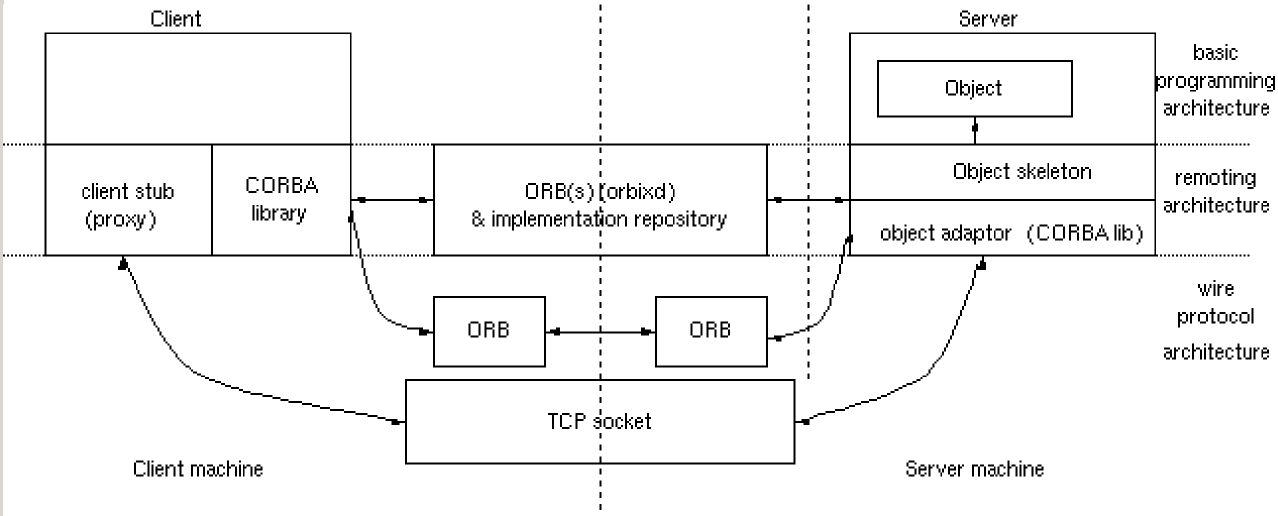
- Distributed extension to COM
- Remote method invocation based on DCE RPC
- Open technology
  - ➤ DCOM submitted to IETF to become an Internet standard
  - ➤ Available for Windows NT 4.0, Windows 95, MacOS
  - ➤ Software AG builds port for Solaris

F.11 DCOM vs. CORBA Architecture

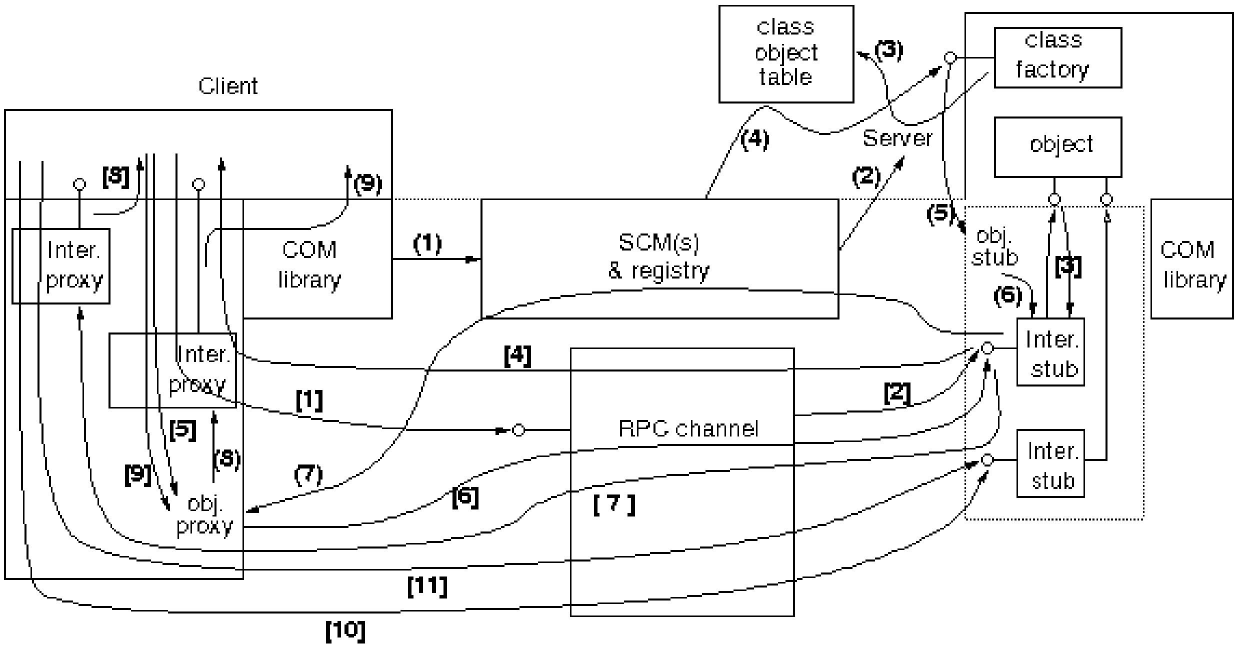
1 Overall Architecture — DCOM

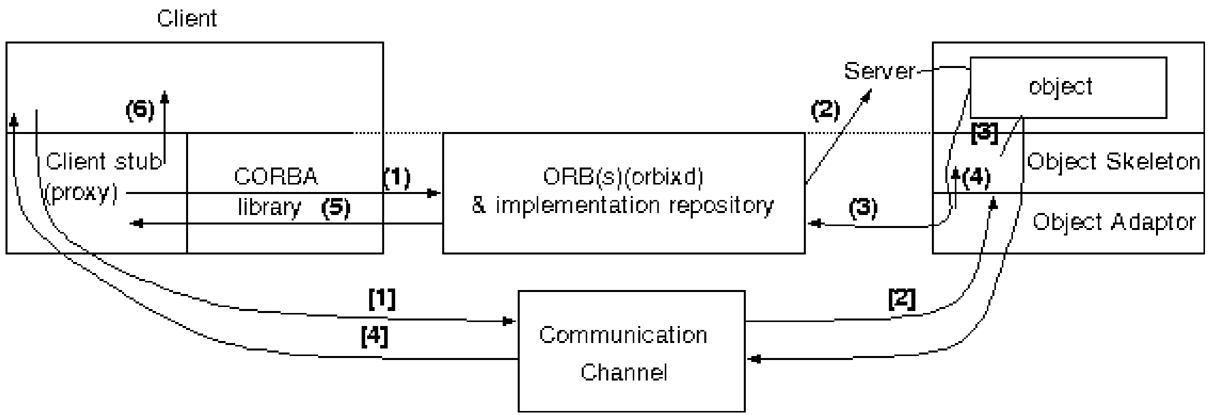


F.11 DCOM vs. CORBA Architecture

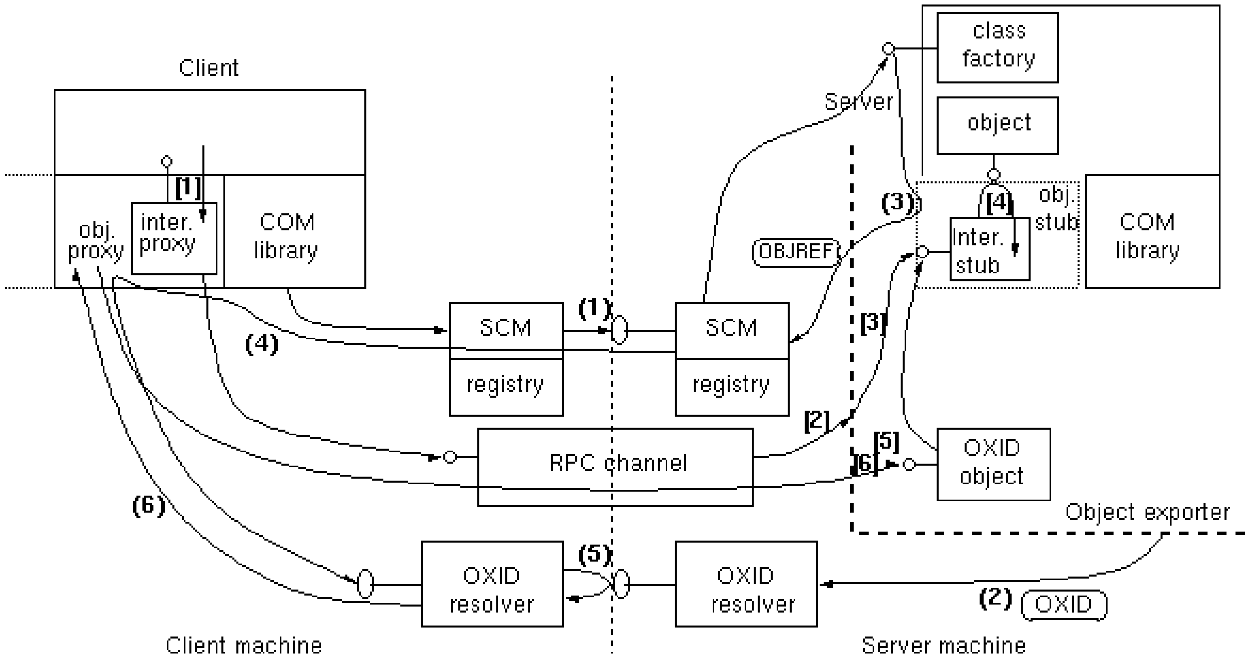


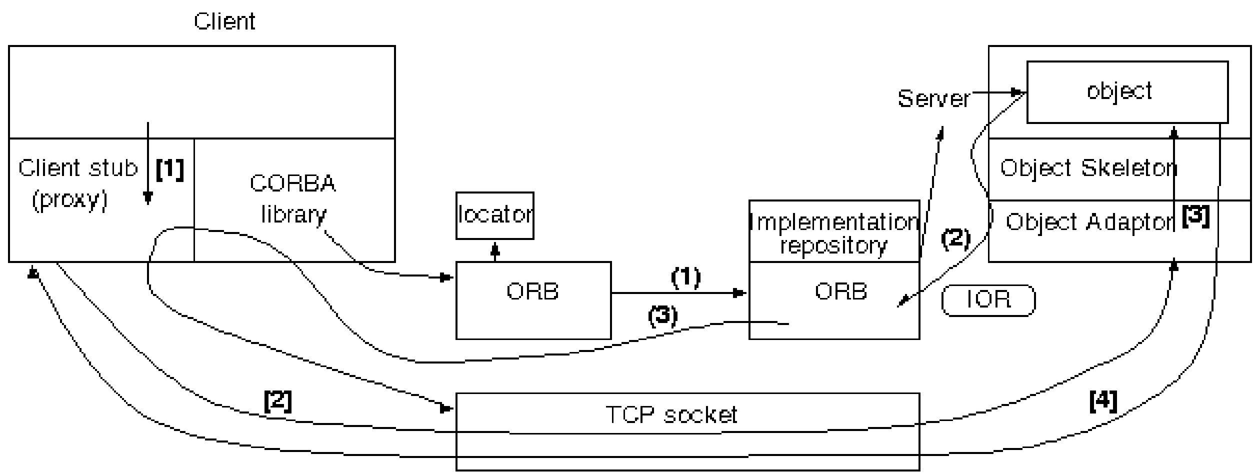
F.11 DCOM vs. CORBA Architecture




Client
Client stub
(proxy)
CORBA
library
(1)
(5)
(6)
ORB(s)(orbixd)
& implementation repository
(2)
(3)
Server
object
[3]
(4)
Object Skeleton
Object Adaptor
[1]
[4]
[2]
Communication
Channel

F.11 DCOM vs. CORBA Architecture




Client
Client stub
(proxy)
[1]
CORBA
library
locator
ORB
(1)
(3)
Implementation
repository
ORB
(2)
IOR
Server
object
Object Skeleton
Object Adaptor
[3]
[2]
TCP socket
[4]

G Frameworks

G.1 Overview

Design is hard.
One way to avoid the act of design is to reuse existing designs
[Ralph Johnson]

- Class libraries
- Frameworks — What they are, How they work, Benefits
- Types of Frameworks
- CORBA & Frameworks
- Java Frameworks

G.2 References

JoF88. R.E Johnson, B. Foote: Designing Reusable Classes, *Journal of Object-Oriented Programming*, June 1988, ftp://st.cs.uiuc.edu/pub/papers/frameworks/designing-reusable-classes.ps.

Joh91. R.E Johnson, Reusing Object-Oriented Design, University of Illinois, Technical Report UIUCDCS 91-1696, 1991, ftp://st.cs.uiuc.edu/pub/papers/frameworks/reusable-oo-design.ps.

Joh92. R.E Johnson, Documenting Frameworks Using Patterns, OOPSLA '92 Proceedings, 1992, ftp://st.cs.uiuc.edu/pub/patterns/papers/documenting-frameworks.ps

Joh93. R.E Johnson, How to Design Frameworks, Tutorial Notes, OOPSLA '93, Washington, 1993, ftp://st.cs.uiuc.edu/pub/papers/frameworks/OOPSLA93-frmwk-tut.ps

OHE96. Robert Orfali, Dan Harkey, Jeri Edwards: *The Essential Distributed Objects Survival Guide*, John Wiley, New York, 1996.

SFO97. San Francisco Project Technical Summary, http://www.javasoft.com/javareel/isv/IBM/SanFrancisco/prod_summary.html

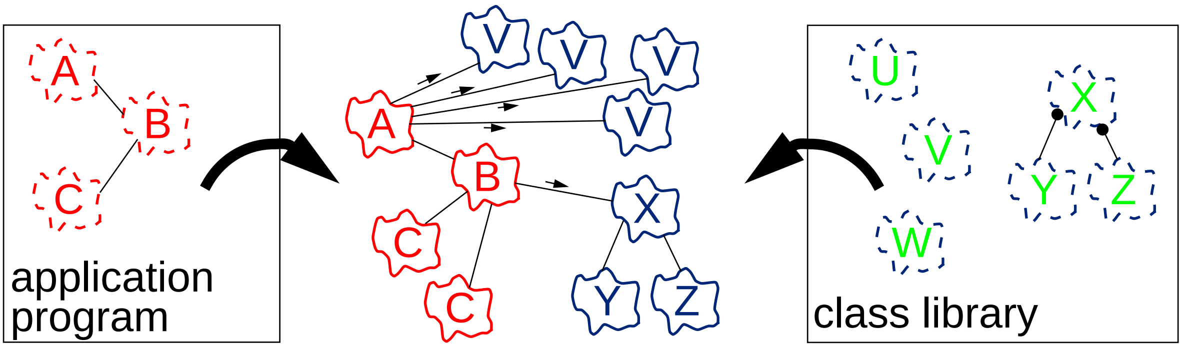
Tal93. Taligent Inc., Leveraging Object-Oriented Frameworks, A Taligent White Paper, 1993

Tal94. Taligent Inc.,Building Object-Oriented Frameworks, A Taligent White Paper, 1994, http://www.taligent.com/Technology/WhitePapers/BuildingFwks/BuildingFrameworks.html

Tal95. Taligent Inc, The Power of Frameworks - For windows and OS/2 developers, Addison-Wesley, 1995

1 Classes and Class Libraries

- Class = design for a set of objects
- Class library
  - ◆ Collection of classes
  - ◆ Flow of control: application objects → library objects



2 Frameworks (What)

- Framework = design for a set of applications
  - ➥ design of a set of objects that collaborate to carry out a set of responsibilities
  - ➥ a way to reuse high-level design
- Framework =

  set of classes

  \+ rules how the objects play together

  \+ definition of the interfaces in the game (how can I join)

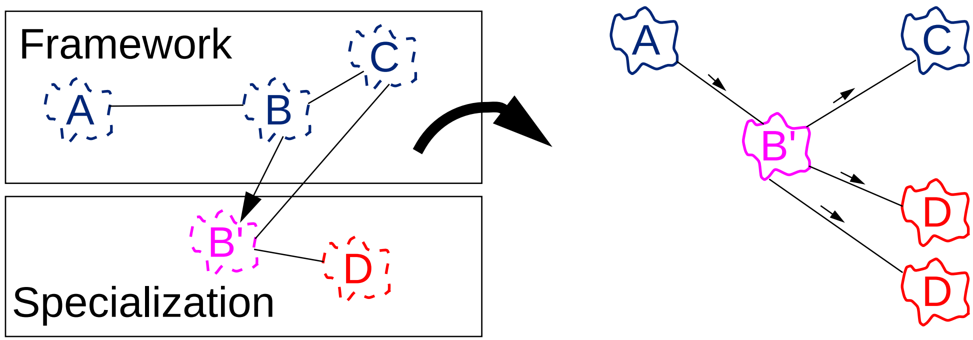
  \+ definition of interfaces to the game (interaction with the outside world)

  \+ definition of the goals of the game
- Compared with a hardware board
  - ◆ the board = instance of the Framework
  - ◆ ICs = objects
  - ◆ backplane = ORB

2 Frameworks (What - 2)

Frameworks

- are an application or application skeleton
- application developer may
  – add – substitute – modify
  components
- Flow of control:
  framework → application object → framework
  "Don't call us, we'll call you" (Hollywood principle)



3 Frameworks (How)

- Two sorts of interfaces, two ways for customization

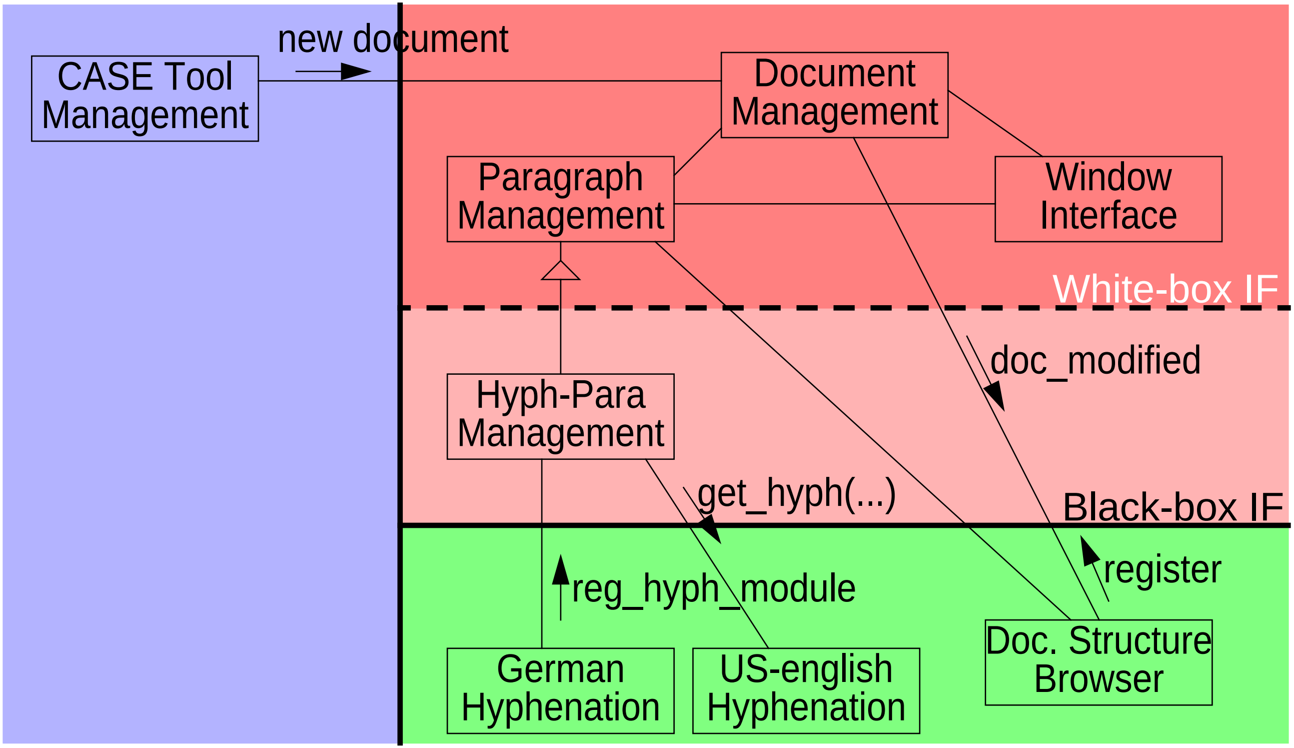
- Client API
  - ➤ external interface of the framework
    (how can other applications interact with the framework)
  - ➤ described in IDL

- Framework API
  (Black-box Interface for customization)
  - ➤ internal interface of the framework
    (how can new components interact with the rest of the framework
    + how does the framework interact with the new components)
  - ➤ interface described in IDL, protocol for registration & notification

- Subclasses of framework components
  (White-box Interface for customization)
  - ➤ customization + replacement of components of the framework
  - ➤ polymorphism guarantees interoperation

4 Example

- Framework for document processing



5 Benefits

- Prefabricated infrastructure
  - ➤ reduces coding, debugging & testing
- Architectural guidance
  - ➤ software is wired and ready to go
  - ➤ you just have to plug in your extensions
- Less monolithic applications
  - ➤ small pieces of applications are plugged into existing frameworks
  - ➤ existing frameworks are plugged together
- Foundation for a software components industry
  - ➤ Well-designed general frameworks are the basis for problem-specific solutions
- Reduced maintenance
  - ➤ Frameworks provide the bulk of (hopefully well-tested) code

G.4 Types of Frameworks

1 Application Frameworks

- Expertise applicable to a wide variety of programs
  - ➤ graphical user interfaces

2 Domain Frameworks

- Expertise in a particular problem domain
  - ➤ manufacturing control
  - ➤ document processing

3 Support Frameworks

- System-level services
  - ➤ file systems, device interaction, ...

G.5 CORBA & Frameworks

- Main goal of CORBA
  - ➥ infrastructure for business objects

★ Business objects

- ➤ a representation of a thing active in the business domain
- ➤ includes
  - – business name and definition
  - – attributes, behavior, relationship, constraints
- ➤ examples:
  - – a person (customer), a place, a concept (invoice, contract), …

➥ may be used in unpredictable combinations

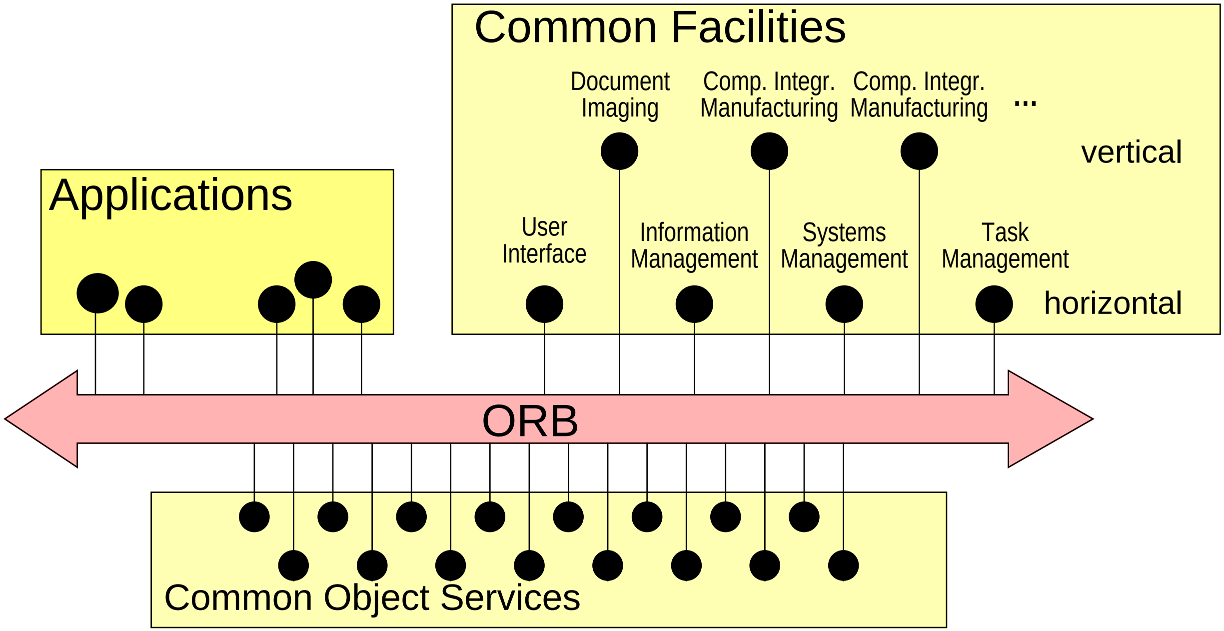
➥ is independent of specific applications

◆ represents a "everyday life entity" $\rightarrow$ exists in the "end user's world"

◆ in contrast: entities that make sense only to information systems

G.5 CORBA & Frameworks (2)

- CORBA Facilities
  - ➤ provide services for business objects
  - ➤ are built on top of Common Object Services
  - ➤ extend the "primitive" COS



G.6 Java Frameworks

- Java Media Framework
  - ➤ audio and video device control
- Lightweight UI Framework
  - ➤ Customizable user interface environment
- General Administrative Framework

★ Fundamental concepts for building Frameworks with Java:

- ➥ Interfaces to describe type conformance
- ➥ Component technology: Java Beans

H Java & Component Models

H.1 Overview

- Component models
- Java — Design goals & key properties
- JavaBeans
  - ➤ Architecture
  - ➤ Properties
  - ➤ Events
  - ➤ Introspection

H.2 Component Models

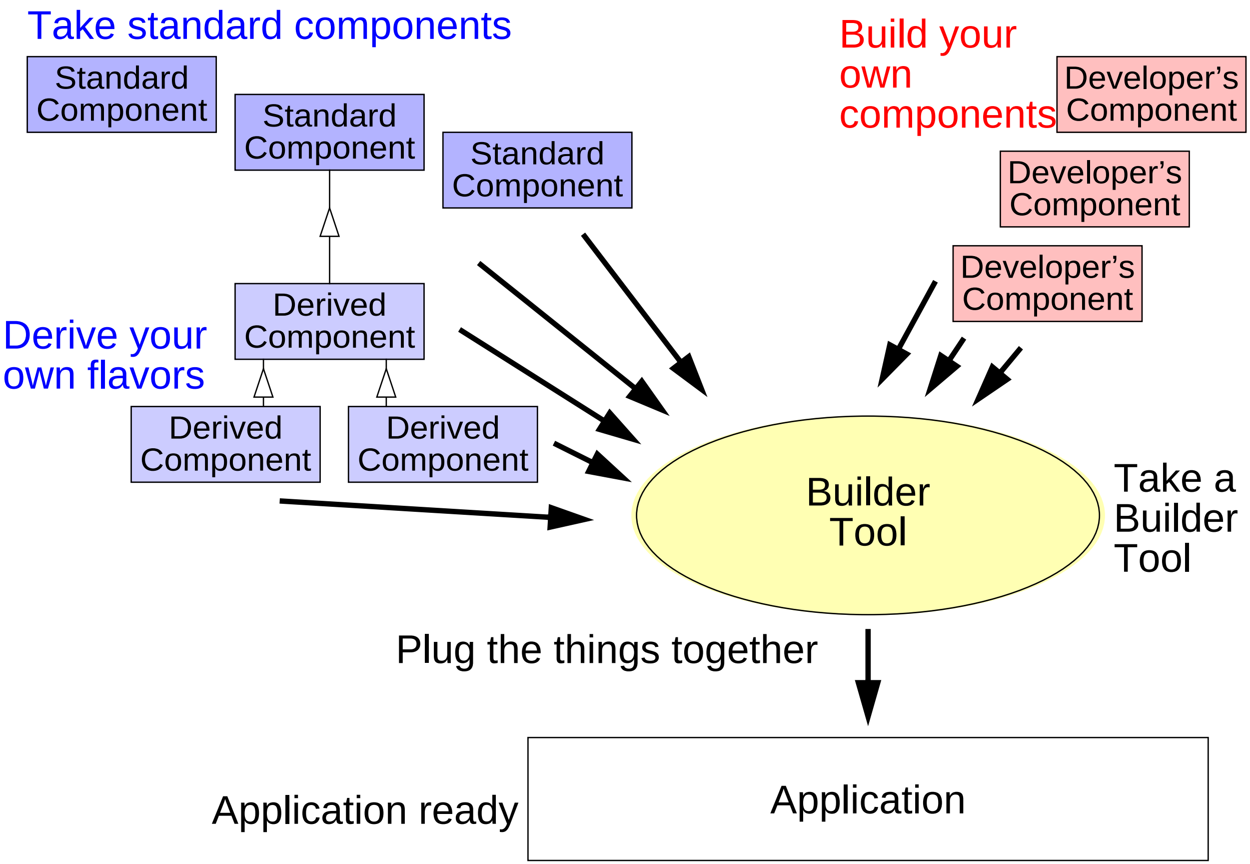
1 What is a Software Component?

- A reusable piece of software that:
  - ➤ has a well-specified public interface
  - ➤ can be used in unpredictable combinations
  - ➤ is a stand-alone, marketable entity
- Software components achieve reuse by following standard conventions
- Software components can be combined (visually) to complex applications
  - ➤ software kit
  - ➤ visual programming with builder tools
- Self-describing
  - ➤ automatic analysis of interface and properties possible

2 Rival Component Architectures

- JavaBeans
- ActiveX
  - ➤ not portable, proprietary
- OpenDOC
  - ➤ pretty much dead
- Proprietary Solutions
  - ➤ GUI builder class libraries

3 Philosophy



H.3 Java — Design Goals?

- Solve today's problems with development and distribution of software
  - various operating systems (Unix, Windows, MacOS, …)
  - various hardware architectures
- Java: language and environment for *secure*, *high performance*, and highly *robust* applications on *multiple platforms* in *heterogeneous, distributed networks*

H.4 Java — Key Properties for Components

- Object oriented
- Polymorphism based on class/interface conformance
- Highly dynamic (loading & linking)
- Reflection/introspection mechanisms

H.5 JavaBeans

1 Definition

- JavaBeans is an API specification for creating reusable software components using Java
  - ➤ defines the Java software components
  - ➤ and how they fit together
- A Bean is any Java class that follows the JavaBeans conventions
- The official definition:

  A Java Bean is a reusable software component that can be visually manipulated in builder tools.

2 Beans — Architecture

- Properties
  - ➤ allows customization of the Bean
- Methods
- Events
  - ➤ the wiring that allows Beans to be interconnected
- Adapters
  - ➤ if Beans do not fit together
- Introspection
  - ➤ instead of a repository — just look into the beans

3 Example Beans

- Visual Beans
  - ➤ Custom GUI components
  - ➤ HTML Rendering Bean
  - ➤ OpenGL Canvas
- Non-visual Beans
  - ➤ database connectivity
  - ➤ timer bean
    - – triggers events at certain time intervals
    - – may encapsulate complex date/time logic

4 Properties

- Describe properties of components
- Each property has
  - ➤ a **name** — symbolic description of the property (e. g. Color, Font …)
    `private Color color;`(instance variable of the Bean class)
  - ➤ a **type** — Java class, encapsulating the value(s)
  - ➤ constraints (optional) — e. g. read-only or write-only
- Naming convention
  - ➤ get method for reading

```
public Color getColor(){ return color; }
```

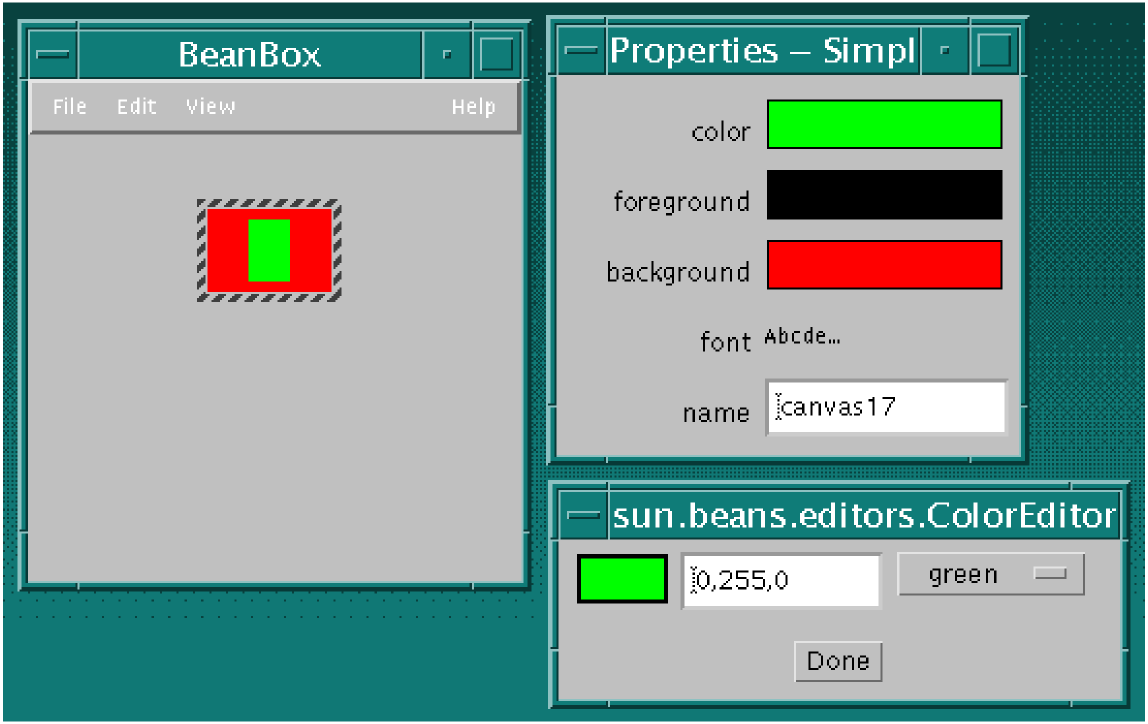
  - ➤ set method for modification

```
public void setColor(Color newColor){
        color = newColor;
        repaint();
}
```

- Examination & modification in property dialogue

4 Properties (2)

- Example: Property Color as property of Bean "SimpleBean"
  - ➤ SimpleBean loaded in BeanBox, clicking on color opens ColorEditor



4 Properties (3)

- Simple Properties
  - ➤ set/get methods
- Bound Properties
  - ➤ fire off a notification when they are changed
- Constrained Properties
  - ➤ proposed changes may be rejected if invalid
- Indexed Properties
  - ➤ used for arrays of properties

5 Events

- Builds on Source-Listener Pattern
  - ➤ EventSource, EventObject, EventListener framework
- PropertyChangeSupport
  - ➤ Allows a Bean to send out notifications whenever a property value changes
- VetoableChangeSupport
  - ➤ Allows Beans to reject property values that are out of range
- Interested Beans register as EventListener

5 Events (2)

- Bean events are not limited to properties.
- You can define your own events using a simple naming convention:
  - ➤ Example

same name

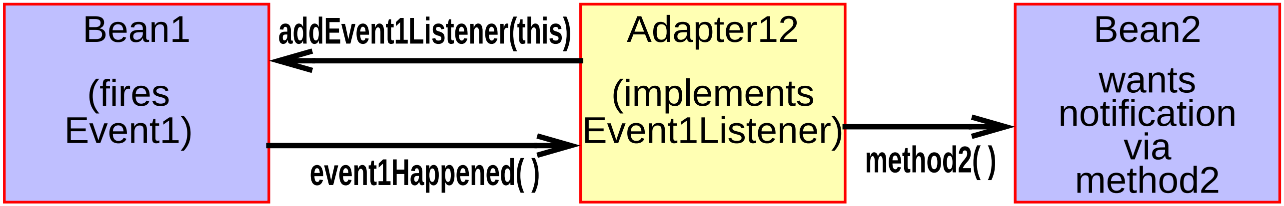
```
public void addTimerListener(TimerListener l)
public void removeTimerListener(TimerListener l)
```

add/remove

TimerListener must be
a subclass of
java.util.EventListener

6 Adaptors

- Adaptation of events of one bean to methods of another bean



  - ➤ Adapter12 implements appropriate Event-Listener interface
  - ➤ Adapter12 registers for Event1
  - ➤ Event1 happens, Bean1 invokes the method, which was defined in the event interface (**`event1Happend()`**), at all registered event listeners
  - ➤ **`event1Happend()`** in Adapter12 invokes **`method2()`** at Bean2

- Adapter may manipulate event data

- Automatic generation of simple adapters possible

7 Introspection

- Allows automatic analysis of beans
- Java1.1 Reflection API
  - ➤ analysis of Java classes at runtime
  - ➤ members: name & type
  - ➤ methods: name, parameters & return type
- JavaBeans naming conventions
  - ➤ get/set methods -> properties
  - ➤ add/remove methods -> events
  - ➤ other methods -> ordinary methods
- Alternative: Information in BeanInfo class

◆ some sort of interface repository

  - ➤ developer may explicitly specify properties and events