INTRODUCTION TO SOFTWARE ENGINEERING

# 6. SOFTWARE DEVELOPMENT METHODS

Bui Thi Mai Anh

anhbtm@soict.hust.edu.vn

ĐẠI HỌC BÁCH KHOA

## Content

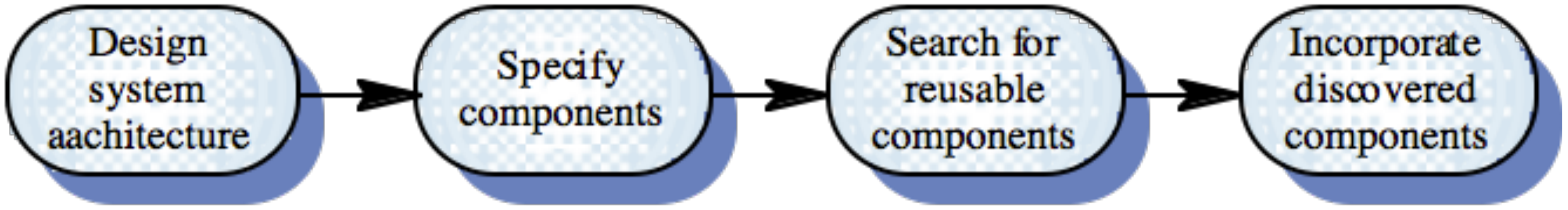
## Software Reuse

- In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems)
- Software engineering has focused on custom development of components
- To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process

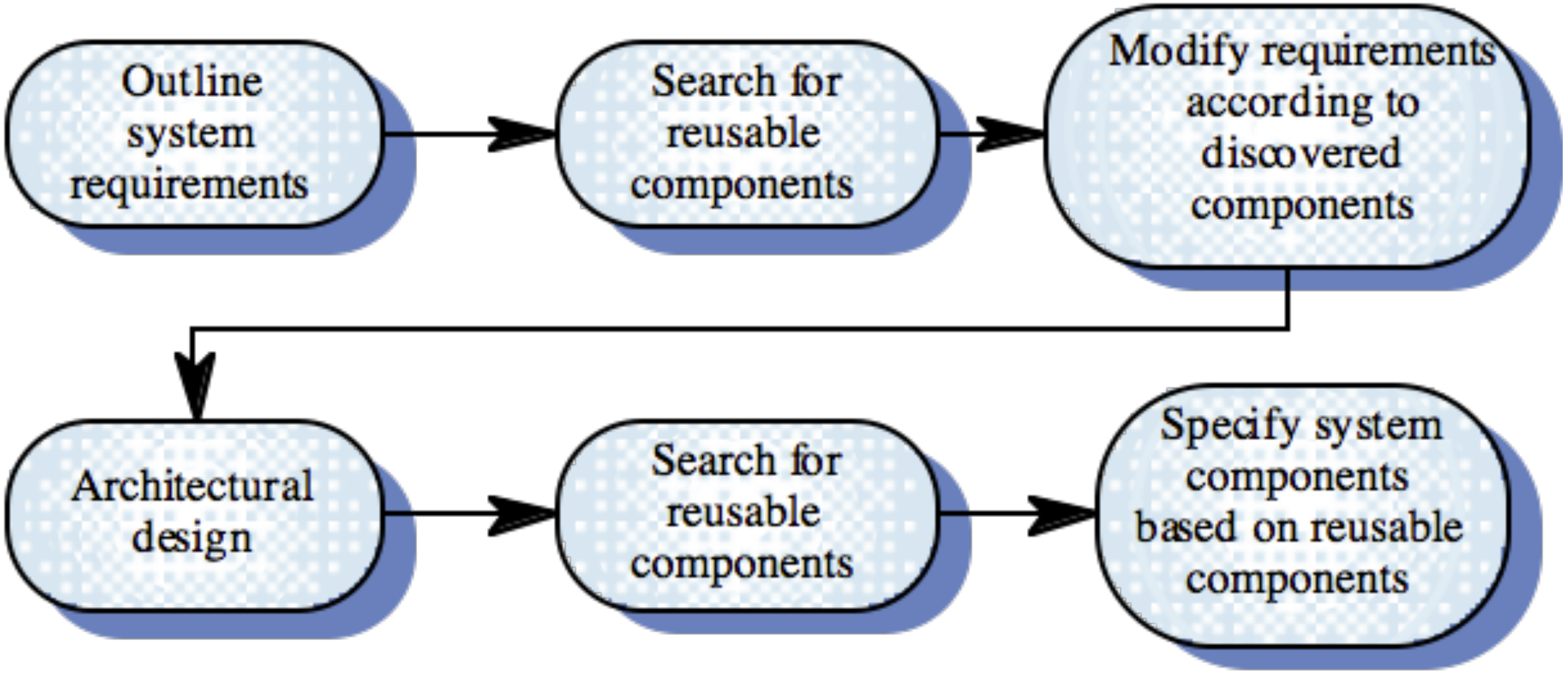
# Types of Software Reuse

- Application System Reuse
  - reusing an entire application by incorporation of one application inside another (COTS reuse)
  - development of application families (e.g. MS Office)
- Component Reuse
  - components (e.g. subsystems or single objects) of one application reused in another application
- Function Reuse
  - reusing software components that implement a single well-defined function

## Opportunistic Reuse


Design system aachitecture
Specify components
Search for reusable components
Incorporate discovered components

## Development Reuse as a Goal


Outline system requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Specify system components based on reusable components

## Benefits of Reuse

- Increased Reliability
  - components already exercised in working systems
- Reduced Process Risk
  - less uncertainty in development costs
- Effective Use of Specialists
  - reuse components instead of people
- Standards Compliance
  - embed standards in reusable components
- Accelerated Development
  - avoid custom development and speed up delivery

## Requirements for Design with Reuse

- You need to be able to find appropriate reusable components
- You must be confident that that component you plan to reuse is reliable and will behave as expected
- The components to be reused must be documented to allow them to be understood and modified (if necessary)

## Reuse Problems

- Increased maintenance costs
- Lack of tool support
- Pervasiveness of the “not invented here” syndrome
- Need to create and maintain a component library
- Finding and adapting reusable components

## Economics of Reuse - part 1

- Quality
  - with each reuse additional component defects are identified and removed which improves quality.
- Productivity
  - since less time is spent on creating plans, models, documents, code, and data the same level of functionality can be delivered with less effort so productivity improves.

## Economics of Reuse - part 2

- Cost
    - savings projected by estimating the cost of building the system from scratch and subtracting the costs associated with reuse and the actual cost of the software as delivered.
- Cost analysis using structure points
    - can be computed based on historical data regarding the costs of maintaining, qualification, adaptation, and integrating each structure point.

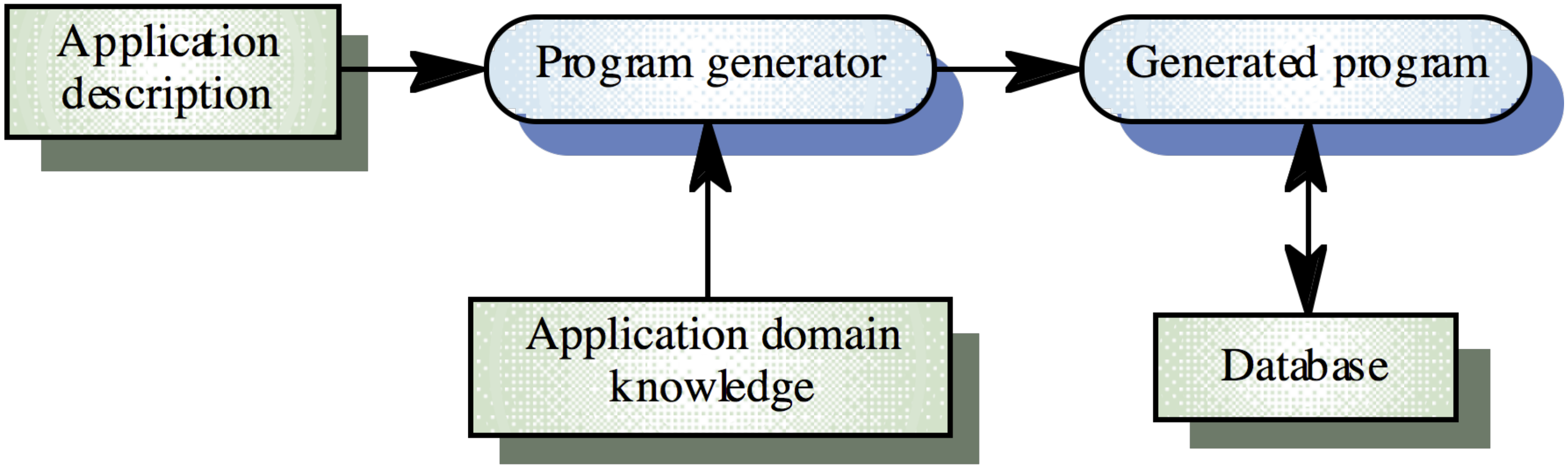
## Generator-Based Reuse

- Program generators reuse standard patterns and algorithms
- Programs are automatically generated to conform to user defined parameters
- Possible when it is possible to identify the domain abstractions and their mappings to executable code
- Domain specific language is required to compose and control these abstractions

## Types of Program Generators

- Applications generators for business data processing
- Parser and lexical analyzers generators for language processing
- Code generators in CASE tools
- User interface design tools

# Program Generation



## Assessing Program Generator Reuse

- Advantages
  - Generator reuse is cost effective
  - It is easier for end-users to develop programs using generators than other CBSE techniques
- Disadvantages
  - The applicability of generator reuse is limited to a small number of application domains

## Content

# Procedural programming

- Often thought as a synonym for **imperative programming**.
- Specifying the **steps** the program must take to reach the desired **state**.
- Based upon the concept of the **procedure call**.
- Procedures, also known as routines, subroutines, methods, or functions that contain a series of computational steps to be carried out.
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

- A procedural programming language provides a programmer a means to define precisely each step in the performance of a task. The programmer knows what is to be accomplished and provides through the language step-by-step instructions on how the task is to be done.
- Using a procedural language, the programmer specifies **language statements** to perform a **sequence of algorithmic steps**.

## Procedural programming

- Possible benefits:
  - Often a better choice than simple sequential or unstructured programming in many situations which involve moderate complexity or require significant ease of maintainability.
  - The ability to re-use the same code at different places in the program without copying it.
  - An easier way to keep track of program flow than a collection of "GOTO" or "JUMP" statements (which can turn a large, complicated program into spaghetti code).
  - The ability to be strongly modular or structured.
- The main benefit of procedural programming over first- and second-generation languages is that it allows for **modularity**, which is generally desirable, especially in large, complicated programs.
- Modularity was one of the earliest **abstraction** features identified as desirable for a programming language.

## Procedural programming

- **Scoping** is another abstraction technique that helps to keep procedures strongly modular.

- It prevents a procedure from accessing the variables of other procedures (and vice-versa), including previous instances of itself such as in recursion.

- Procedures are convenient for making pieces of code written by different people or different groups, including through programming **libraries**.
  - specify a simple interface
  - self-contained information and algorithmics
  - reusable piece of code

## Procedural programming

- The focus of procedural programming is to break down a programming task into a collection of **variables**, **data structures**, and **subroutines**, whereas in object-oriented programming it is to break down a programming task into objects with each "object" encapsulating its own data and methods (subroutines).

- The most important distinction is whereas procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together so an "object" operates on its "own" data structure.

## Procedural programming

- The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs.
- FORTRAN (1954) was the first major programming language to remove through **abstraction** the obstacles presented by machine code in the creation of complex programs.
- FORTRAN was a compiled language that allowed named variables, complex expressions, subprograms, and many other features now common in imperative languages.
- In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed.
- In the 1970s, Pascal was developed by Niklaus Wirth, and C was created by Dennis Ritchie.
- For the needs of the United States Department of Defense, Jean Ichbiah and a team at Honeywell began designing Ada in 1978.

# Content

# Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures encapsulating data fields and procedures together with their interactions – to design applications and computer programs.

- Associated programming techniques may include features such as data **abstraction**, **encapsulation**, **modularity**, **polymorphism**, and **inheritance**.

- Though it was invented with the creation of the **Simula** language in 1965, and further developed in **Smalltalk** in the 1970s, it was not commonly used in mainstream software application development until the early 1990s.

- Many modern programming languages now support OOP.

## OOP concepts: class

- A class defines the abstract characteristics of a thing (object), including that thing's **characteristics** (its attributes, fields or properties) and the thing's **behaviors** (the operations it can do, or methods, operations or functionalities).

- One might say that a class is a blueprint or factory that describes the nature of something.

- Classes provide **modularity** and **structure** in an object-oriented computer program.

- A class should typically be recognizable to a non-programmer familiar with the **problem domain**, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation).

- Collectively, the properties and methods defined by a class are called its **members**.

## OOP concepts: object

- An object is an individual of a class created at run-time trough object **instantiation** from a class.
- The set of values of the attributes of a particular object forms its **state**. The object consists of the **state** and the **behavior** that's defined in the object's class.
- The object is instantiated by implicitly calling its constructor, which is one of its member functions responsible for the creation of instances of that class.

## OOP concepts: attributes

- An **attribute**, also called data member or member variable, is the data encapsulated within a class or object.
- In the case of a regular field (also called **instance variable**), for each instance of the object there is an instance variable.
- A static field (also called **class variable**) is one variable, which is shared by all instances.
- Attributes are an object's variables that, upon being given values at instantiation (using a **constructor**) and further execution, will represent the state of the object.
- A class is in fact a data structure that may contain different fields, which is defined to contain the procedures that act upon it. As such, it represents an **abstract data type**.
- In pure object-oriented programming, the attributes of an object are local and cannot be seen from the outside. In many object-oriented programming languages, however, the attributes may be accessible, though it is generally considered bad design to make data members of a class as externally visible.

## OOP concepts: method

- A **method** is a subroutine that is exclusively associated either with a class (in which case it is called a **class method** or a static method) or with an object (in which case it is an **instance method**).

- Like a subroutine in procedural programming languages, a method usually consists of a sequence of programming statements to perform an action, a set of input parameters to customize those actions, and possibly an output value (called the return value).

- Methods provide a mechanism for accessing and manipulating the encapsulated state of an object.

- Encapsulating methods inside of objects is what distinguishes object-oriented programming from procedural programming.

# OOP concepts: method

- ***instance*** methods are associated with an object
- ***class*** or ***static*** methods are associated with a class.
- The object-oriented programming paradigm intentionally favors the use of methods for each and every means of access and change to the underlying data:
  - **Constructors:** Creation and initialization of the state of an object. Constructors are called automatically by the run-time system whenever an object declaration is encountered in the code.
  - **Retrieval and modification of state:** accessor methods are used to access the value of a particular attribute of an object. Mutator methods are used to explicitly change the value of a particular attribute of an object. Since an object's state should be as hidden as possible, accessors and mutators are made available or not depending on the information hiding involved and defined at the class level
  - **Service-providing:** A class exposes some "service-providing" methods to the exterior, who are allowing other objects to use the object's functionalities. A class may also define private methods who are only visible from the internal perspective of the object.
  - **Destructor:** When an object goes out of scope, or is explicitly destroyed, its destructor is called by the run-time system. This method explicitly frees the memory and resources used during its execution.

# OOP concepts: method

- The difference between procedures in general and an object's method is that the method, being associated with a particular object, may access or modify the data private to that object in a way consistent with the intended behavior of the object.
- So rather than thinking "a procedure is just a sequence of commands", a programmer using an object-oriented language will consider a method to be **"an object's way of providing a service**“. A method call is thus considered to be a request to an object to perform some task.
- Method calls are often modeled as a means of passing a message to an object. Rather than directly performing an operation on an object, a message is sent to the object telling it what it should do. The object either complies or raises an exception describing why it cannot do so.
- Smalltalk used a real “**message passing**” scheme, whereas most object-oriented languages use a standard “function call” scheme for message passing.
- The message passing scheme allows for **asynchronous** function calls and thus **concurrency**.

## OOP concepts: inheritance

- Inheritance is a way to compartmentalize and **reuse** code by creating collections of attributes and behaviors (classes) which can be based on previously created classes.
- The new classes, known as **subclasses** (or derived classes), inherit attributes and behavior of the pre-existing classes, which are referred to as superclasses (or ancestor classes). The inheritance relationships of classes gives rise to a hierarchy.
- **Multiple inheritance** can be defined whereas a class can inherit from more than one superclass. This leads to a much more complicated definition and implementation, as a single class can then inherit from two classes that have members bearing the same names, but yet have different meanings.
- **Abstract inheritance** can be defined whereas abstract classes can declare member functions that have no definitions and are expected to be defined in all of its subclasses.

## OOP concepts: abstraction

- **Abstraction** is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

- Object-oriented programming provides **abstraction** through **composition** and **inheritance**.

## OOP concepts: encapsulation and information hiding

- **Encapsulation** refers to the bundling of data members and member functions inside of a common “box”, thus creating the notion that an object contains its state as well as its functionalities
- **Information hiding** refers to the notion of choosing to either expose or hide some of the members of a class.
- These two concepts are often misidentified. Encapsulation is often understood as including the notion of information hiding.
- Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface — those members accessible to that class.
- The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients.
- It also aims at preventing unauthorized objects to change the state of an object.

## OOP concepts: encapsulation and information hiding

- Members are often specified as **public**, **protected** or **private**, determining whether they are available to all classes, sub-classes or only the defining class.

- Some languages go further:
  - Java uses the default access modifier to restrict access also to classes in the same package
  - C# and VB.NET reserve some members to classes in the same assembly using keywords **internal** (C#) or **friend** (VB.NET)
  - Eiffel and C++ allow one to specify which classes may access any member of another class (C++ friends)
  - Such features are basically overriding the basic information hiding principle, greatly complexify its implementation, and create confusion when used improperly

## OOP concepts: polymorphism

- Polymorphism is the ability of objects belonging to **different typ**es to respond to method, field, or property calls of the **same name**, each one according to an appropriate **type-specific behavior**.
- The programmer (and the program) does not have to know the exact type of the object at compile time. The exact behavior is determined at run-time using a run-time system behavior known as **dynamic binding**.
- Such polymorphism allows the programmer to treat derived class members just like their parent class' members.
- The different objects involved only need to present a **compatible interface** to the clients. That is, there must be public or internal methods, fields, events, and properties with the same name and the same parameter sets in all the superclasses, subclasses and interfaces.
- In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same superclass.

## OOP concepts: polymorphism

- A method or operator can be abstractly applied in many different situations. If a Dog is commanded to speak(), this may elicit a bark(). However, if a Pig is commanded to speak(), this may elicit an oink(). They both inherit speak() from Animal, but their derived class methods override the methods of the parent class. This is **overriding polymorphism**.

- **Overloading polymorphism** is the use of one method signature, or one operator such as "+", to perform several different functions depending on the implementation. The "+" operator, for example, may be used to perform integer addition, float addition, list concatenation, or string concatenation. Any two subclasses of Number, such as Integer and Double, are expected to add together properly in an OOP language. The language must therefore overload the addition operator, "+", to work this way. This helps improve code readability. How this is implemented varies from language to language, but most OOP languages support at least some level of overloading polymorphism.

## OOP concepts: polymorphism

- Many OOP languages also support **parametric polymorphism**, where code is written without mention of any specific type and thus can be used transparently with any number of new types. C++ templates and Java Generics are examples of such parameteric polymorphism.

- The use of pointers to a superclass type later instantiated to an object of a subclass is a simple yet powerful form of polymorhism, such as used un C++.

## OOP: Languages

- Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation.

- Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed.

## OOP: Languages

- Concerning the degree of object orientation, following distinction can be made:
  - Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Smalltalk, Eiffel, Ruby, JADE.
  - Languages designed mainly for OO programming, but with some procedural elements. Examples: C++, C#, Java, Scala, Python.
  - Languages that are historically procedural languages, but have been extended with some OO features. Examples: VB.NET (derived from VB), Fortran 2003, Perl, COBOL 2002, PHP.
  - Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2).
  - Languages with abstract data type support, but not all features of object-orientation, sometimes called object-based languages. Examples: Modula-2, Pliant, CLU.

# OOP: Variations

- There are different ways to view/implement/instantiate objects:

- **<u>Prototype-based</u>**
  - objects - classes + delegation

  - no classes
  - objects are a set of members
  - create ex nihilo or using a prototype object (“cloning”)

- Hierarchy is a "containment" based on how the objects were created using prototyping. This hierarchy is defined using the **delegation principle** can be changed as the program executes prototyping operations.

- examples: ActionScript, JavaScript, JScript, Self, Object Lisp

## OOP: Variations

- **object-based**
  - objects + classes - inheritance

  - classes are declared and objects are instantiated
  - no inheritance is defined between classes
  - No polymorphism is possible

- example: VisualBasic

## OOP: Variations

- **object-oriented**
  - objects + classes + inheritance + polymorphism
  - This is recognized as true object-orientation
  - examples: Simula, Smalltalk, Eiffel, Python, Ruby, Java, C++, C#, etc...

## Content

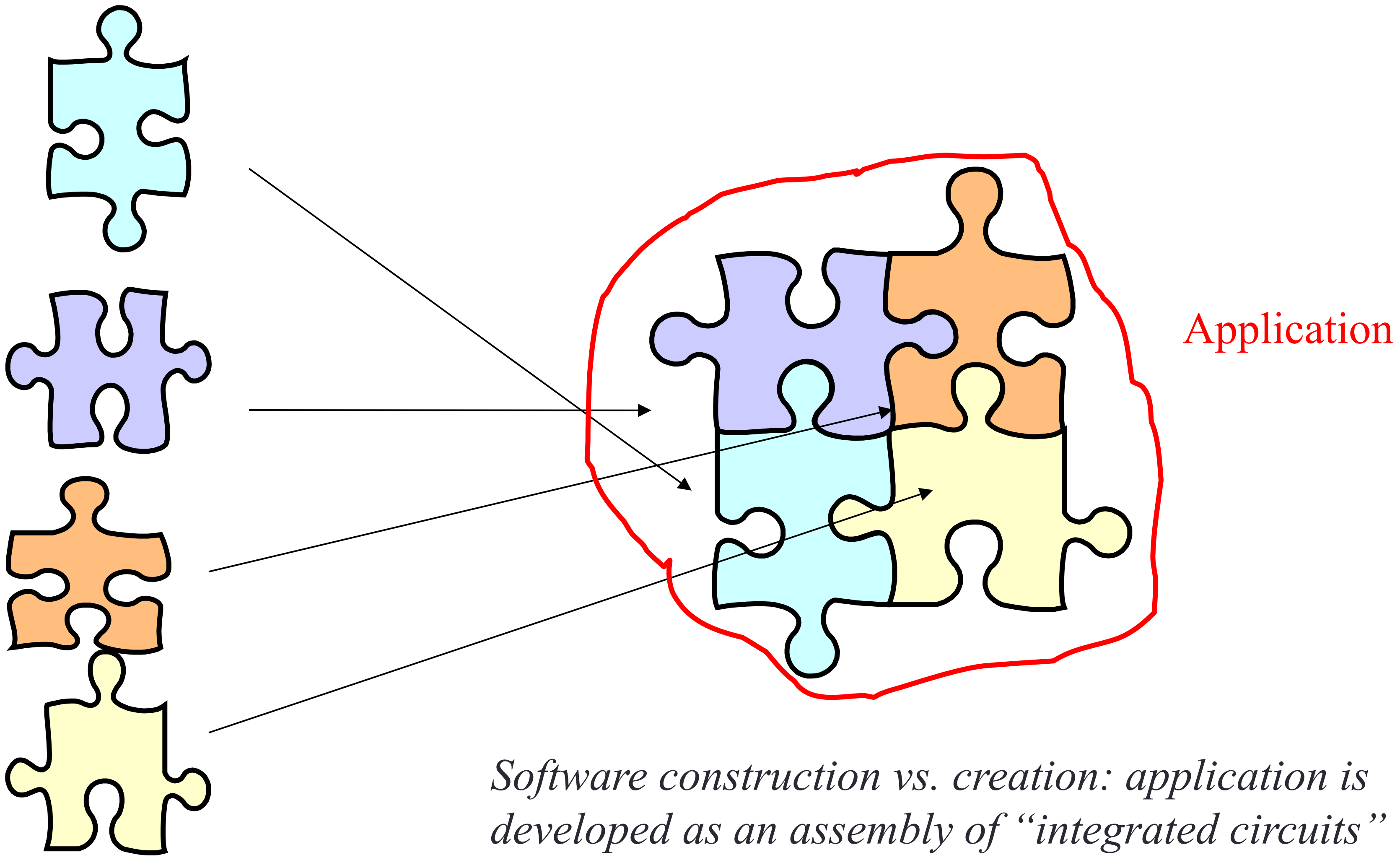
# Component based development

"Systems should be assembled from existing components"

- Idea dates since 1968: Douglas McIllroy: "Mass produced software components"

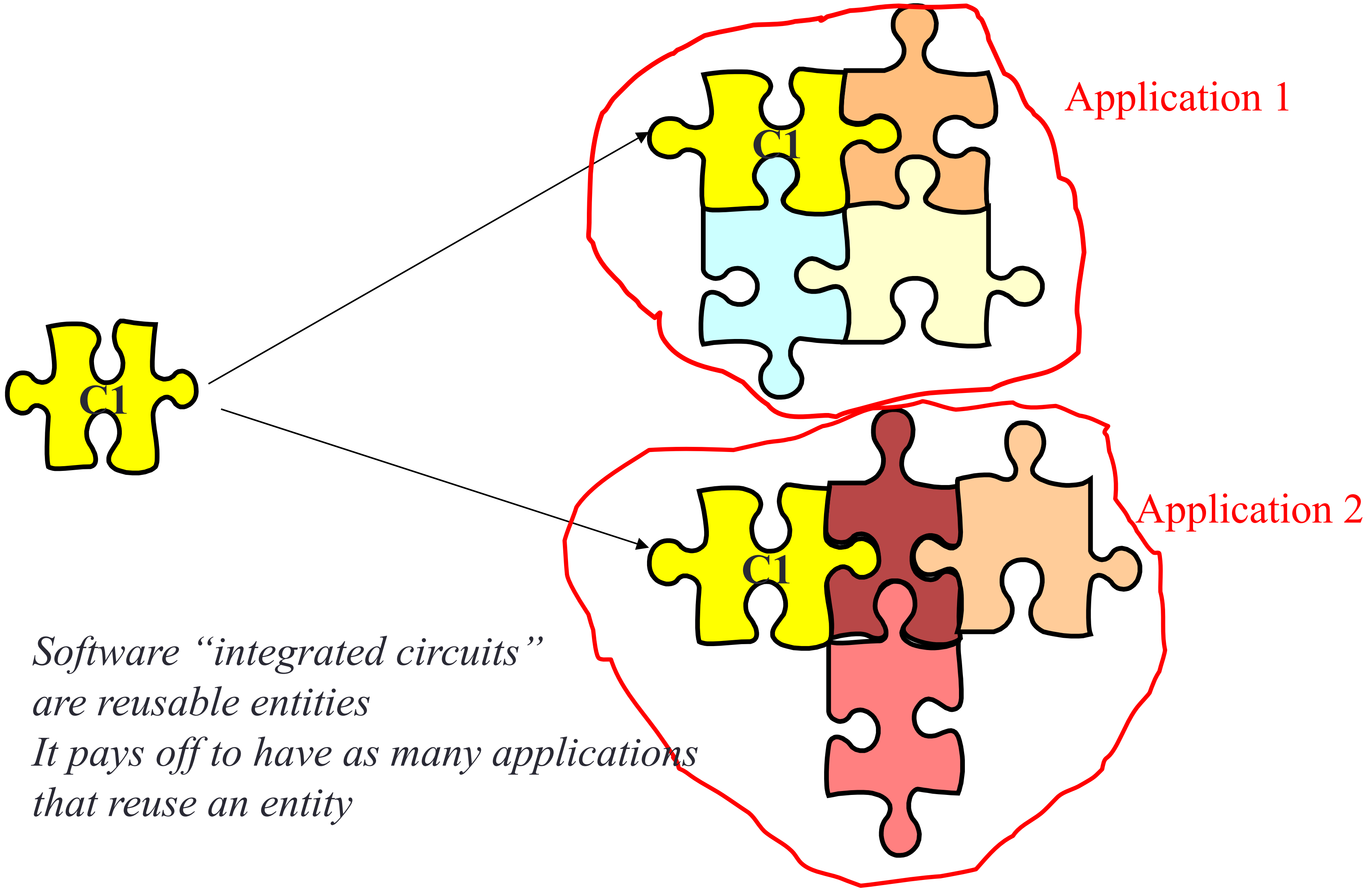
- Component-based software engineering (CBSE) is an approach to software development that relies on software reuse – reusing *artifacts* (*software parts*)
- Advantages of CBSE:
  - **Reuse**: Development of system = assembly of component
  - **Flexibility**: Maintenance,upgrading=customization, replacement of components, extensibility by adding components. His may even happen at run-time with proper infrastructure support !

# Advantage 1: Software construction



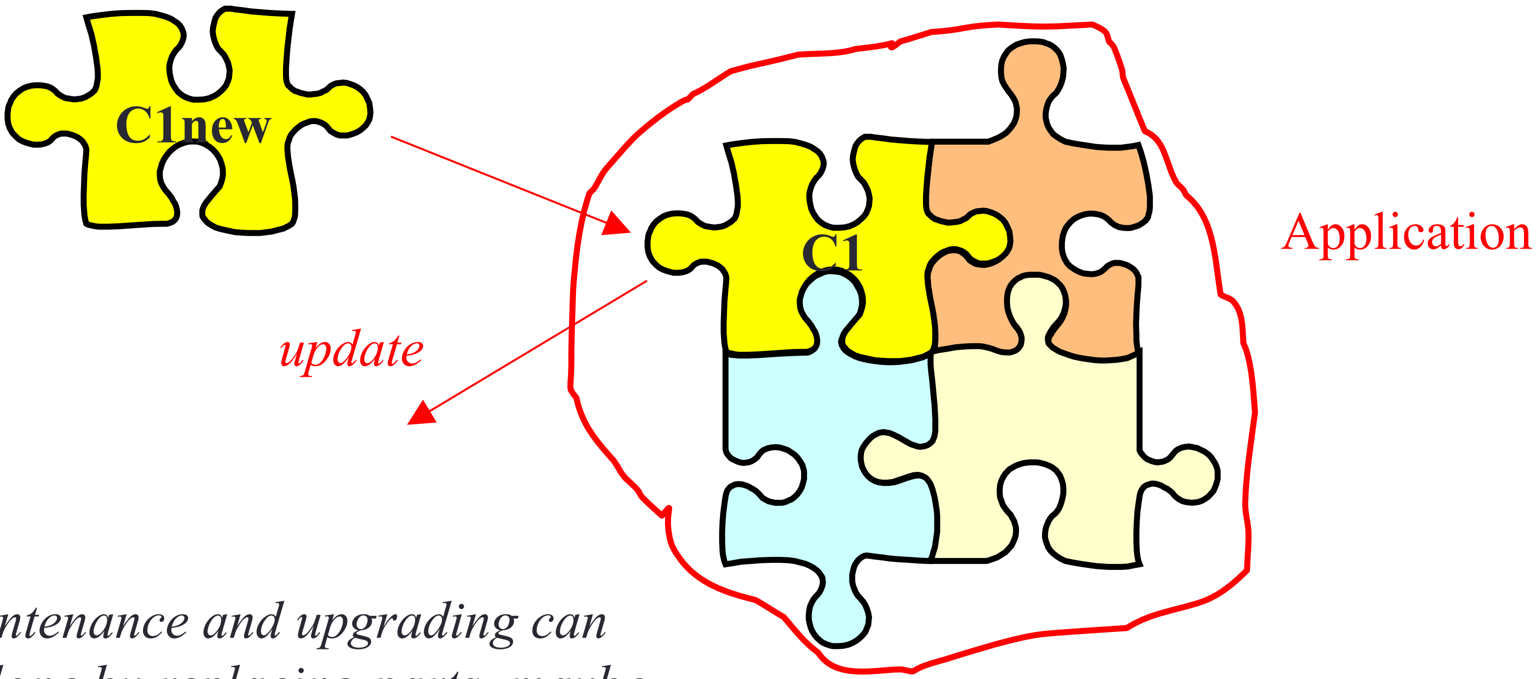
*Software construction vs. creation: application is developed as an assembly of "integrated circuits"*

# Advantage 2: Reuse



*Software "integrated circuits"*
*are reusable entities*
*It pays off to have as many applications*
*that reuse an entity*

## Advantage 3: Maintenance & Evolution



*Maintenance and upgrading can be done by replacing parts, maybe even at runtime*

# What are the "Entities" to compose ?

- Functions
- Modules
- Objects
- Components
- Services
- …

1960

1968: Douglas McIlroy: "*Mass Produc*
*Software Components*"

1970

1980

1990

1998: Clemens Szyperski: "*Componen*
*Software – Beyond Object Oriented*
*Programming*"

2000

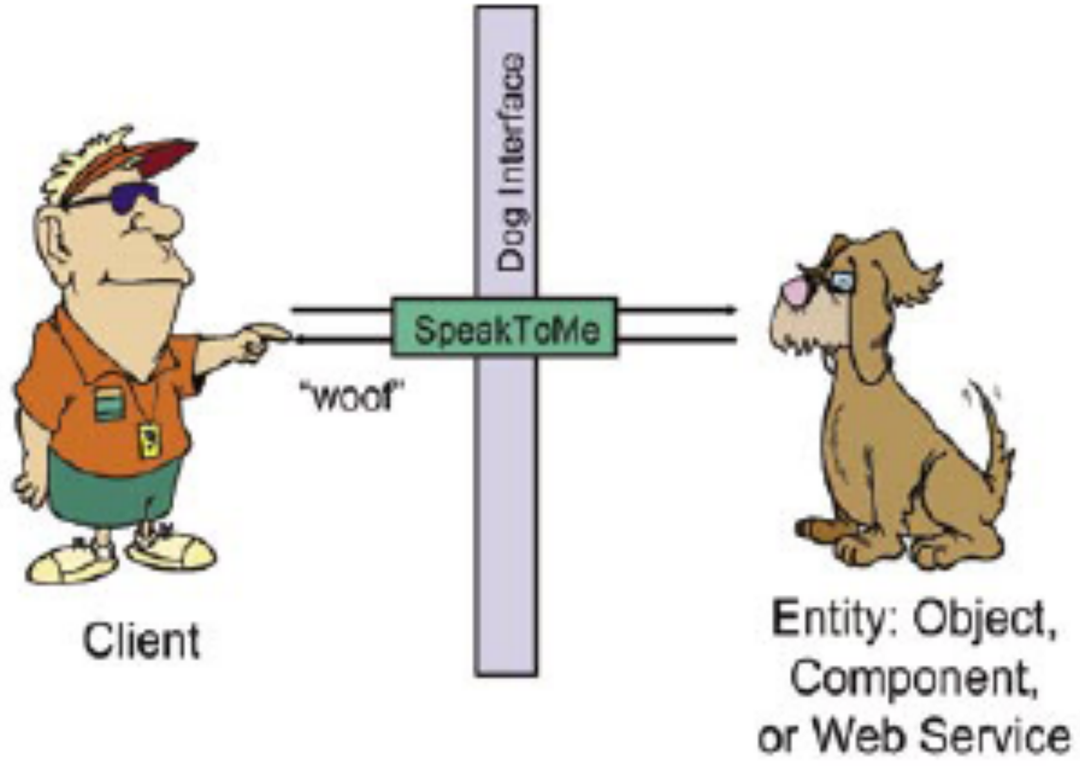
2010

## Principles for reuse by composition

- Key requirements for Black-Box reuse:
  - **Abstraction**: an “Entity” is known by its “interface”
  - **Encapsulation**: the “insides” of an “Entity” are not exposed to the outside

# Commonalities of Reusable Entities

- All are blobs of code that can do something
- All have interfaces that describe what they can do.
- All live in a process somewhere.
- All live to do the bidding of a client.
- All support the concept of a client making requests by "invoking a method."



From [ACM Queue]

# Reusable Entities by Location and Environment

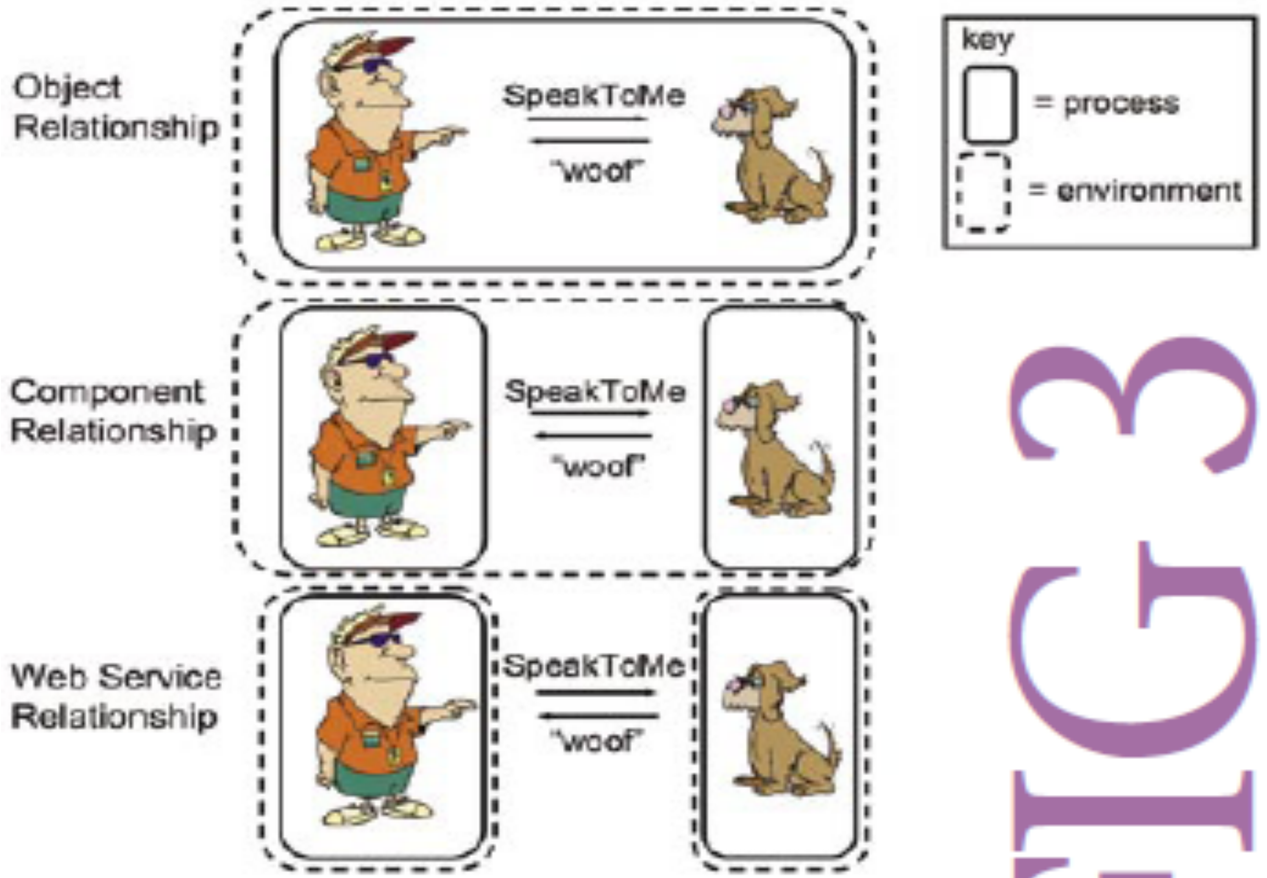
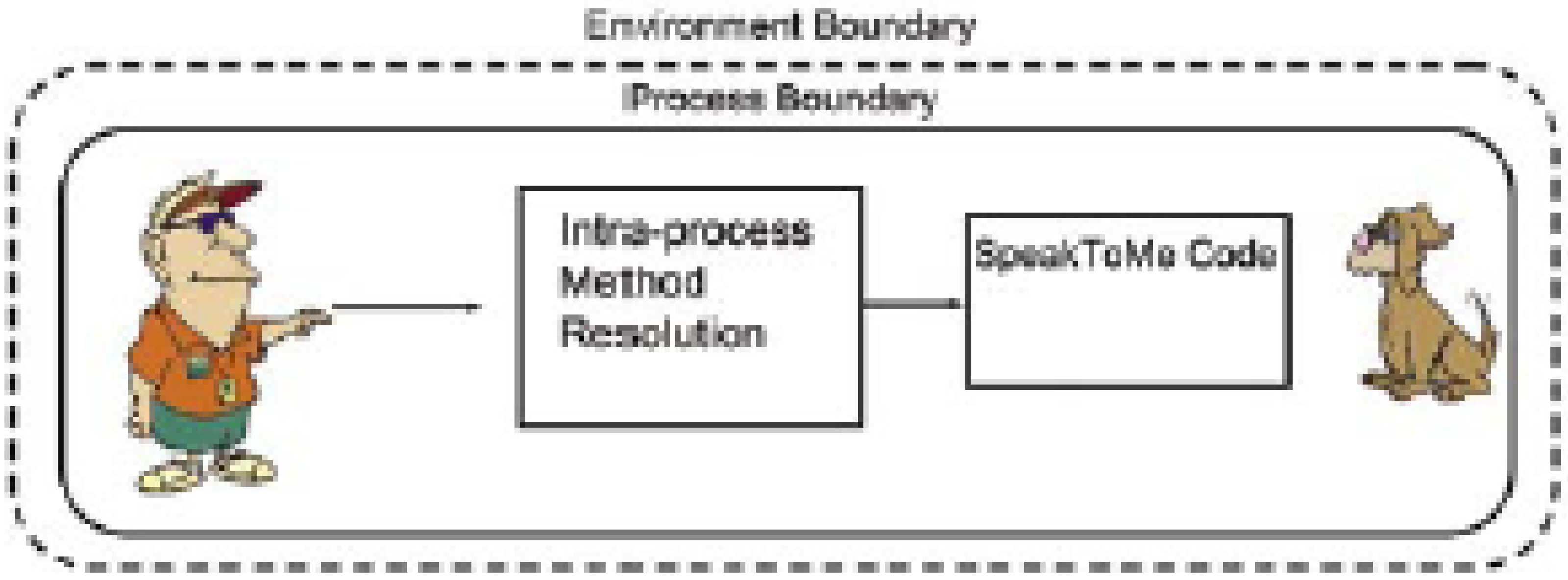


FIG 3

*Environment: the hosting runtime environment for the Entity and the Client (Examples: Microsoft .NET, WebSphere EJB)*

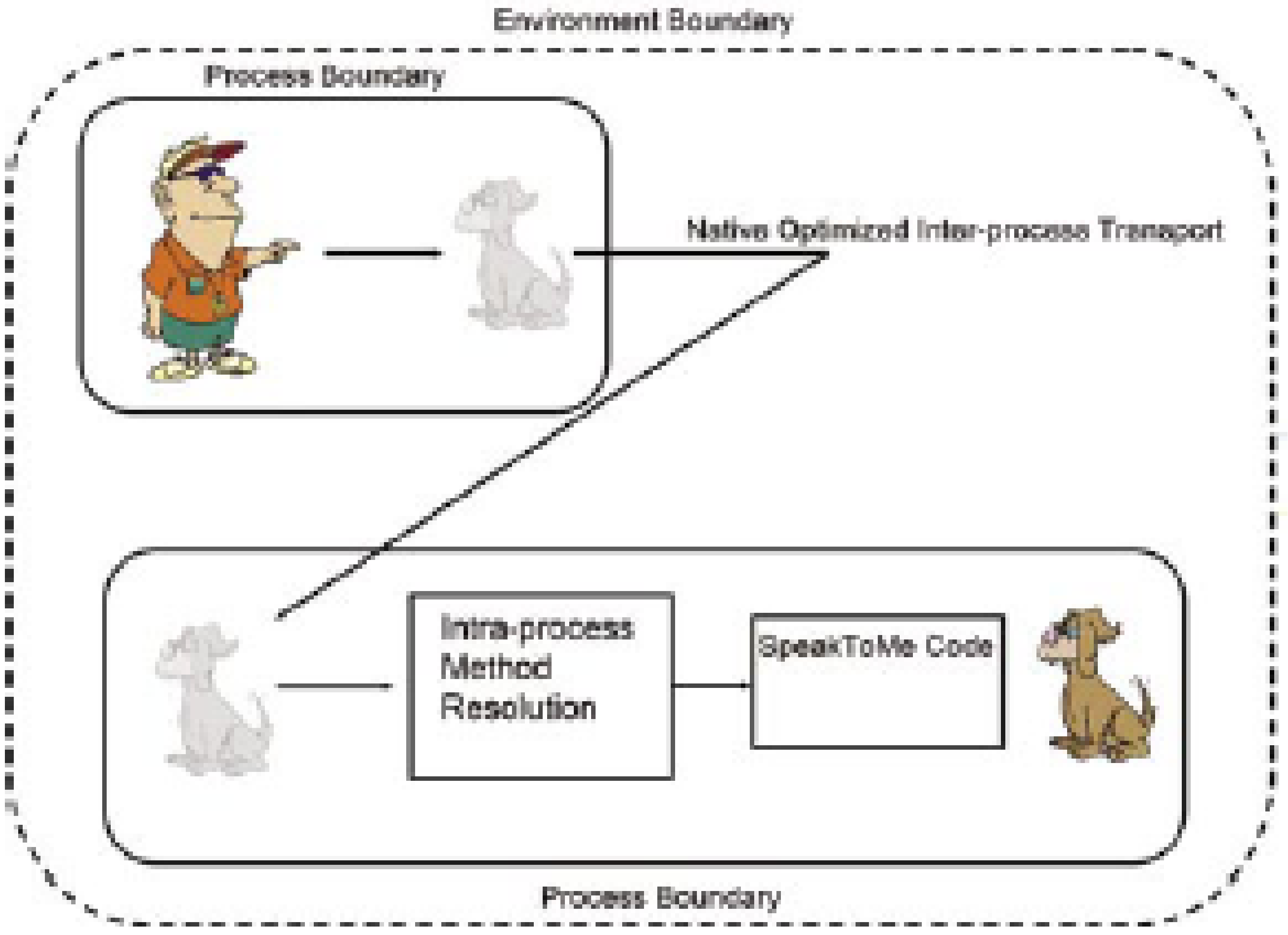
From [ACM Queue]

## Object Method Resolution



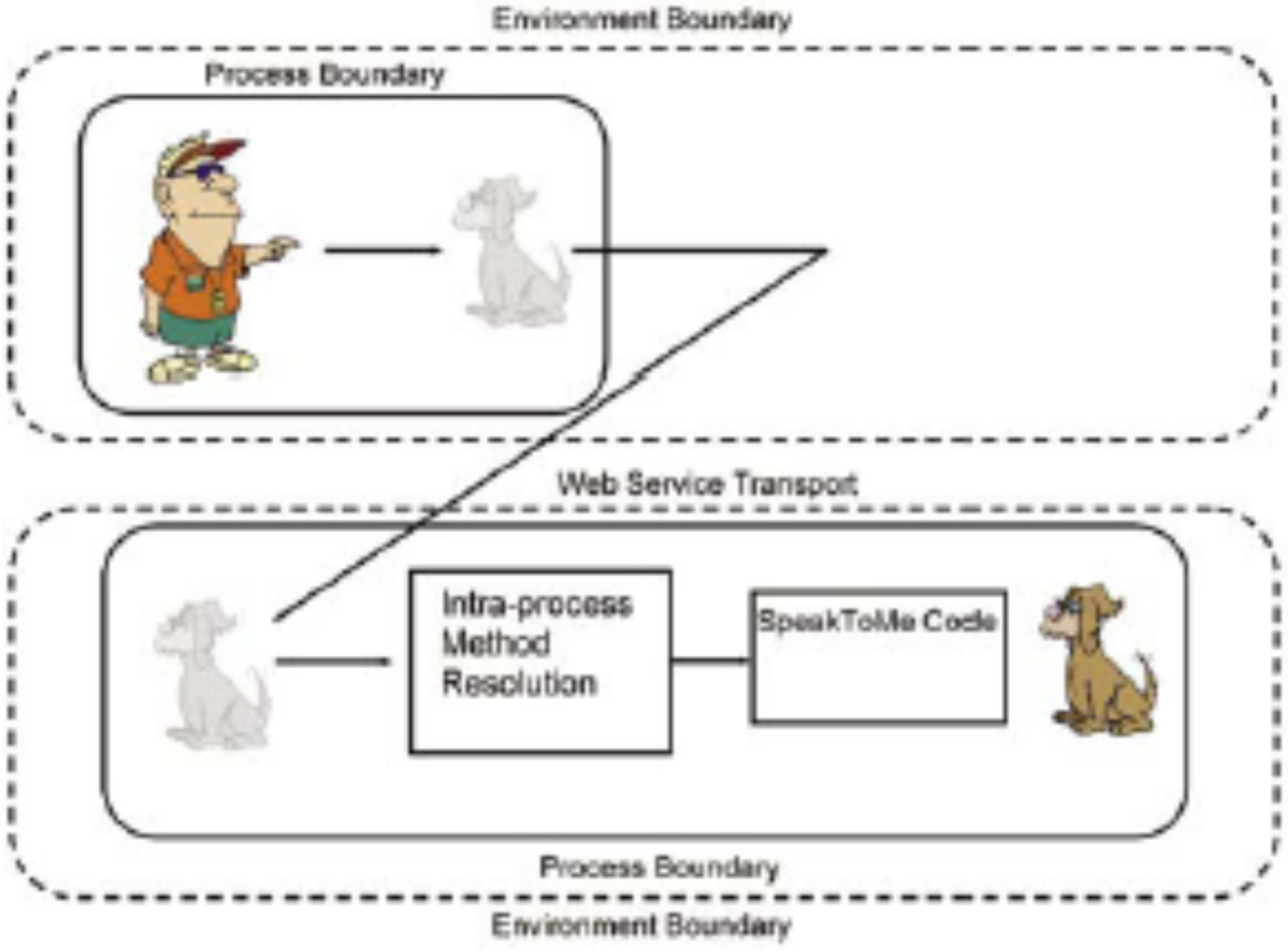
From [ACM Queue]

## Component Method Resolution


Environment Boundary
Process Boundary
Native Optimized Inter-process Transport
Intra-process Method Resolution
SpeakToMe Code
Process Boundary

From [ACM Queue]

## Web Service Method Resolution



From [ACM Queue]

## Objects-Components-Services

| Entities for Reuse and Composition | | |
|---|---|---|
| •Abstraction •Encapsulation | | |
| **Objects** | **Components** | **Services** |
| •Location: same process<br>•Inheritance<br>•Polymorphism | •Location: different processes, same environment<br>•Usually some runtime infrastructure needed<br>•No state<br>•No shared variables | •Location: different environments<br>•More emphasis on interface/contract/service agreement<br>•Mechanisms for dynamic discovery<br>•Dynamically composable |

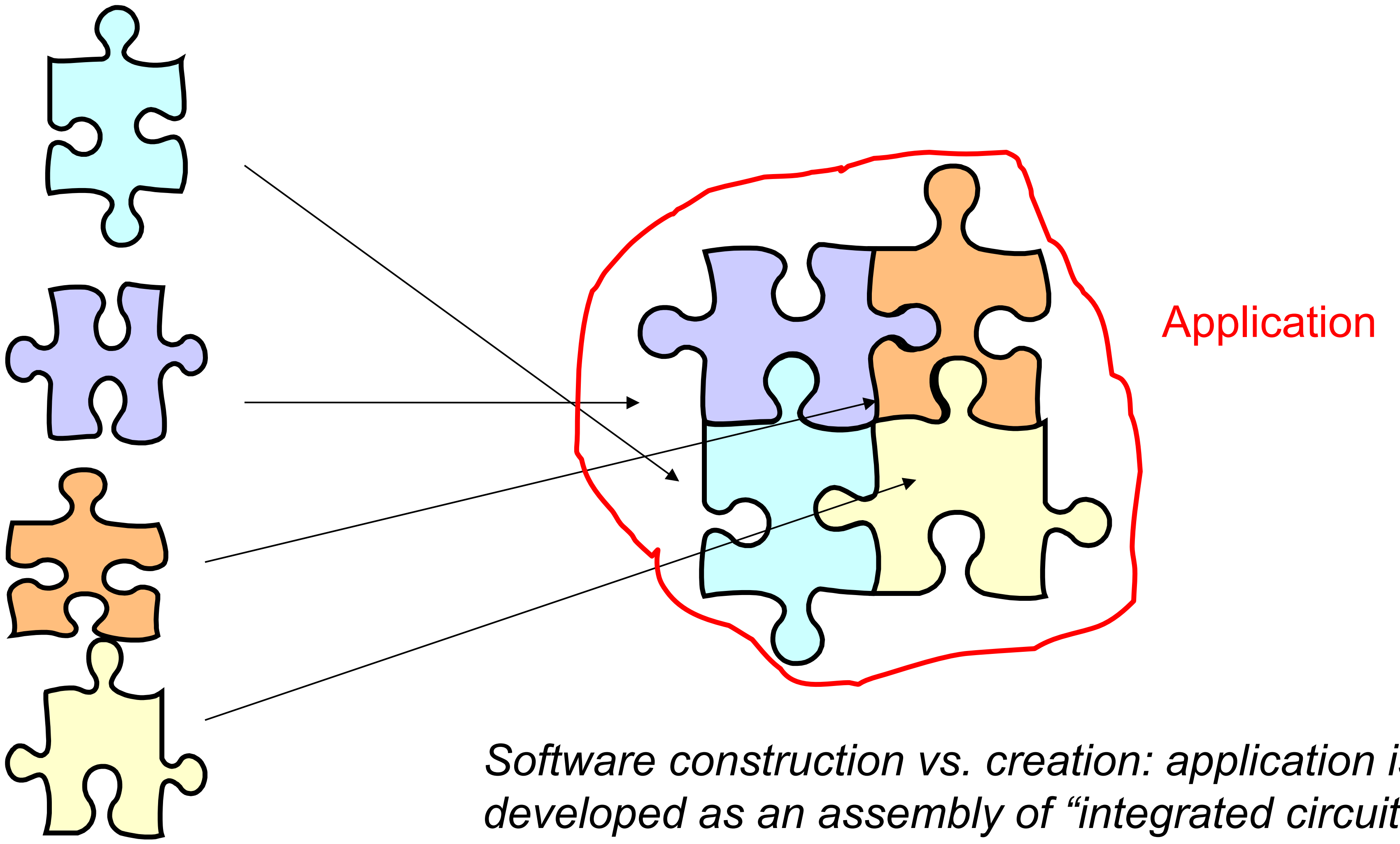
## Reusable Entities made more usable and more composable

- Issues:
    - Interface description – what should contain a complete description ?
    - Composition – how are components glued together ? (do I have to write much glue code ?)
    - Discovery – where and how to find the component/service you need ?
    - Dynamic aspects – when to do discovery/selection/composition
    - Less stress on binary implementation – crossing platform/model boundaries

## CBSE reuse

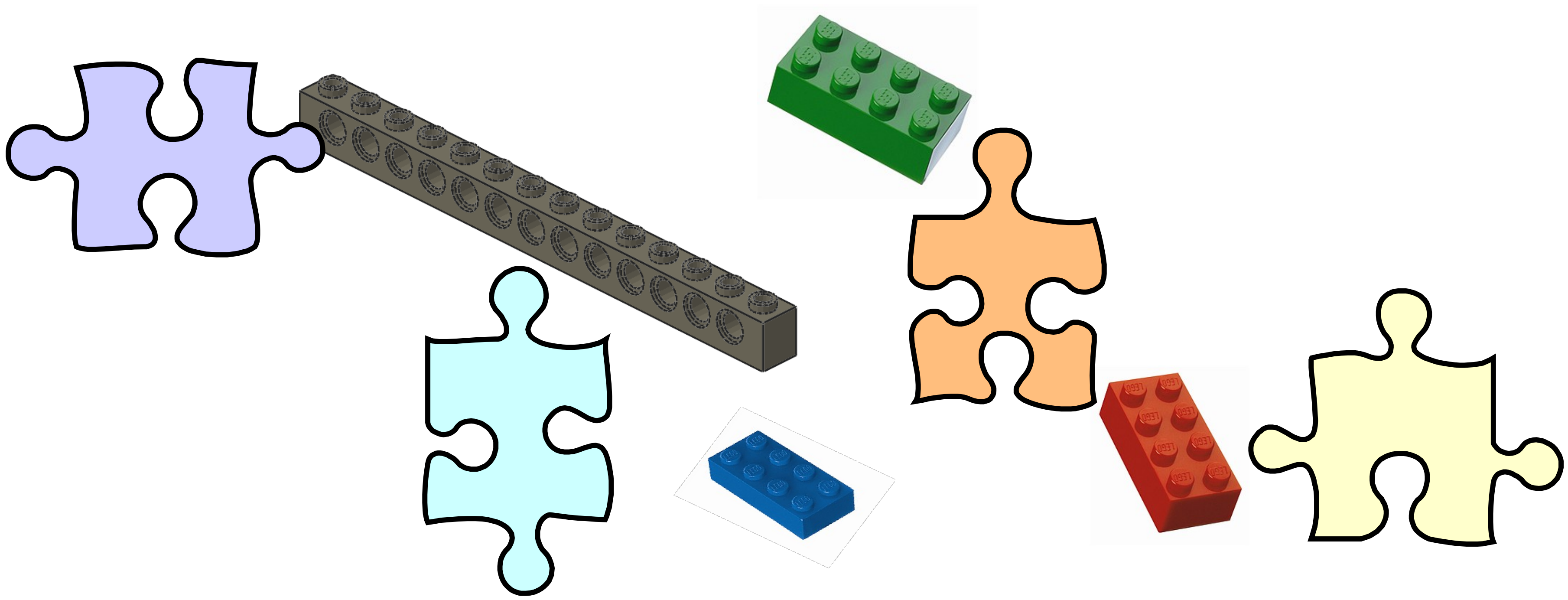
- **Component Based Software Engineering (CBSE) = reuse of:**
  - **Parts (components)**
  - **Infrastructure**

## Component based software construction – the ideal case



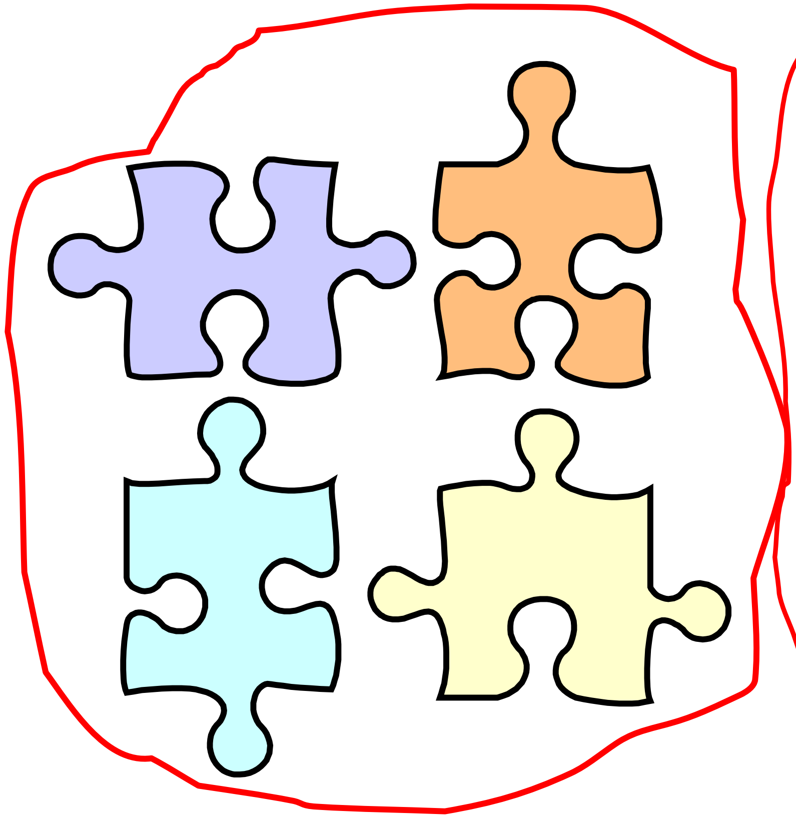
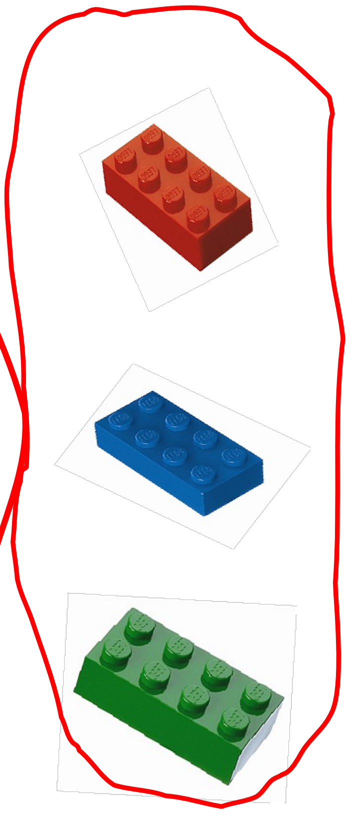
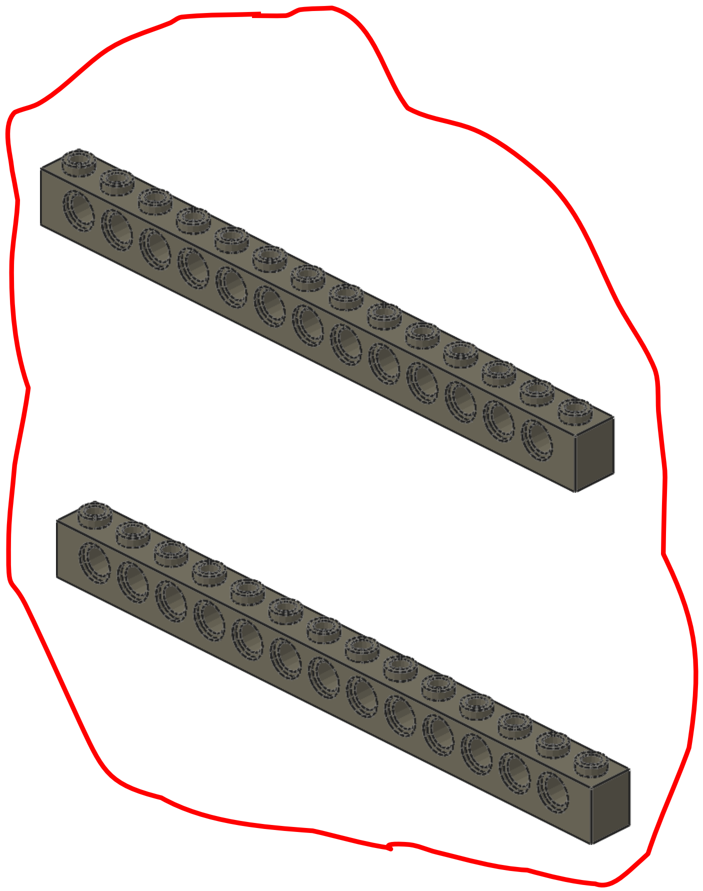
*Software construction vs. creation: application is developed as an assembly of "integrated circuits"*

## Component based software construction – in practice



## Component interactions

Components must obey to common conventions or standards !
Only in this way they will be able to recognise each others interfaces and connect and communicate to each other

## CBSE essentials

- Independent components specified by their interfaces.
  - Separation between interface and implementation
  - Implementation of a component can be changed without changing the system
- Component standards to facilitate component integration.
  - Component models embody these standards
  - Minimum standard operations: how are interfaces specified, how communicate components
  - If components comply to standards, then their operation may be independent of their programming language
- Middleware that provides support for component inter-operability.
  - Provides support for component integration
  - Handles component communication, may provide support for resource allocation, transaction management, security, concurrency
- A development process that is geared to reuse.

## CBSE and design principles

- Apart from the benefits of ***reuse***, CBSE is based on sound software engineering design principles that support the construction of ***understandable*** and ***maintainable*** software:
  - Components are independent so they do not interfere with each other;
  - Component implementations are hidden so they can be changed without affecting others;
  - Communication is through well-defined interfaces so if these are maintained one component can be replaced by another that provides enhanced functionality;
  - Component platforms (infrastructures) are shared and reduce development costs.

## Component definitions - Szyperski

- Szyperski:
- *"A software component is a <u>unit of composition</u> with <u>contractually specified interfaces</u> and <u>explicit context dependencies</u> only. A software component can be <u>deployed independently</u> and is subject to composition by <u>third-parties</u>."*

## Component definitions – Councill and Heinemann

- Councill and Heinmann:
- *"A software component is a software element that conforms to a <u>component model</u> and can be <u>independently deployed</u> and composed without modification according to a <u>composition standard</u>."*

## Component characteristics 1

| | |
|---|---|
| Standardised | Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment. |
| Independent | A component should be independent – it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification. |
| Composable | For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes. |

*Fig. 19.1 from [Sommerville]*

## Component characteristics (cont)

| | |
|---|---|
| Deployable | To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed. |
| Documented | Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified. |

*Fig. 19.1 from [Sommerville]*

## Component interfaces

- An interface of a component can be defined as a specification of its access point, offering no implementation for any of its operations.
- This seperation makes it possible to:
  - Replace the implementation part without changing the interface;
  - Add new interfaces (and implementations) without changing the existing implementation
- A component has 2 kinds of interfaces:
  - Provides interface
    - Defines the services that are provided by the component to the environment / to other components.
    - Essentially it is the component API
    - Mostly methods that can be called by a client of the component
  - Requires interface
    - Defines the services that specifies what services must be made available by the environment for the component to execute as specified.
    - If these are not available the component will not work. This does not compromise the independence or deployability of the component because it is not required that a specific component should be used to provide these services

## Component interfaces

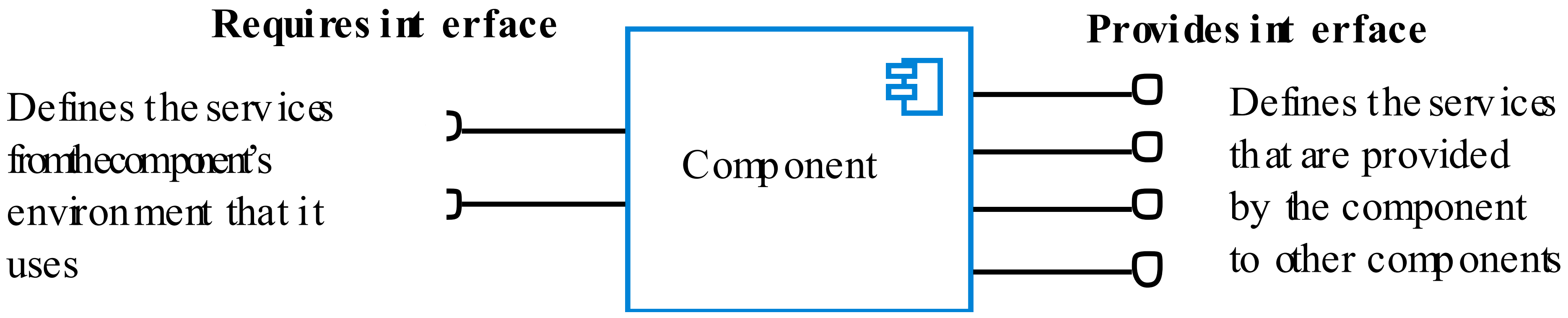

Requires interface
Defines the services from the component's environment that it uses
Component
Provides interface
Defines the services that are provided by the component to other components

*Fig. 19.2 from [Sommerville]*

# Example: A data collector component

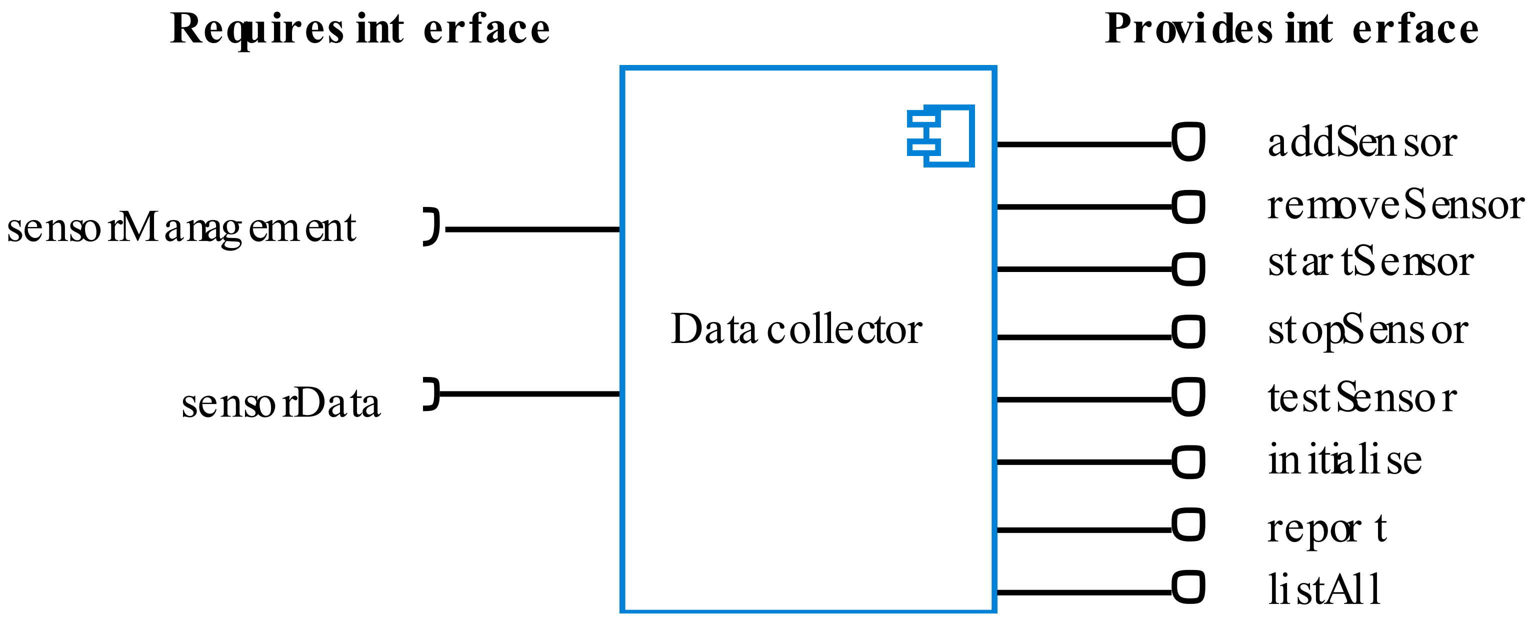


*Fig. 19.3 from [Sommerville]*

## Describing interfaces

- Interfaces defined in standard component technologies using techniques such as Interface Definition Language (IDL) are:
  - Sufficient in describing functional properties.
  - Insuffiecient in describing extra-functional properties such as quality attributes like accuracy, availability, latency, security, etc.
- A more accurate specification of a component's behavior can be achieved through *contracts.*

## Component models

- A component model is a definition of standards for component implementation, documentation and deployment.
- These standards are for:
  - component developers to ensure that components can interoperate
  - Providers of component executioninfrastructures who provide middleware to support component operation
- Examples of component models
  - EJB model (Enterprise Java Beans)
  - COM+ model (.NET model)
  - Corba Component Model
- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

# Elements of a component model

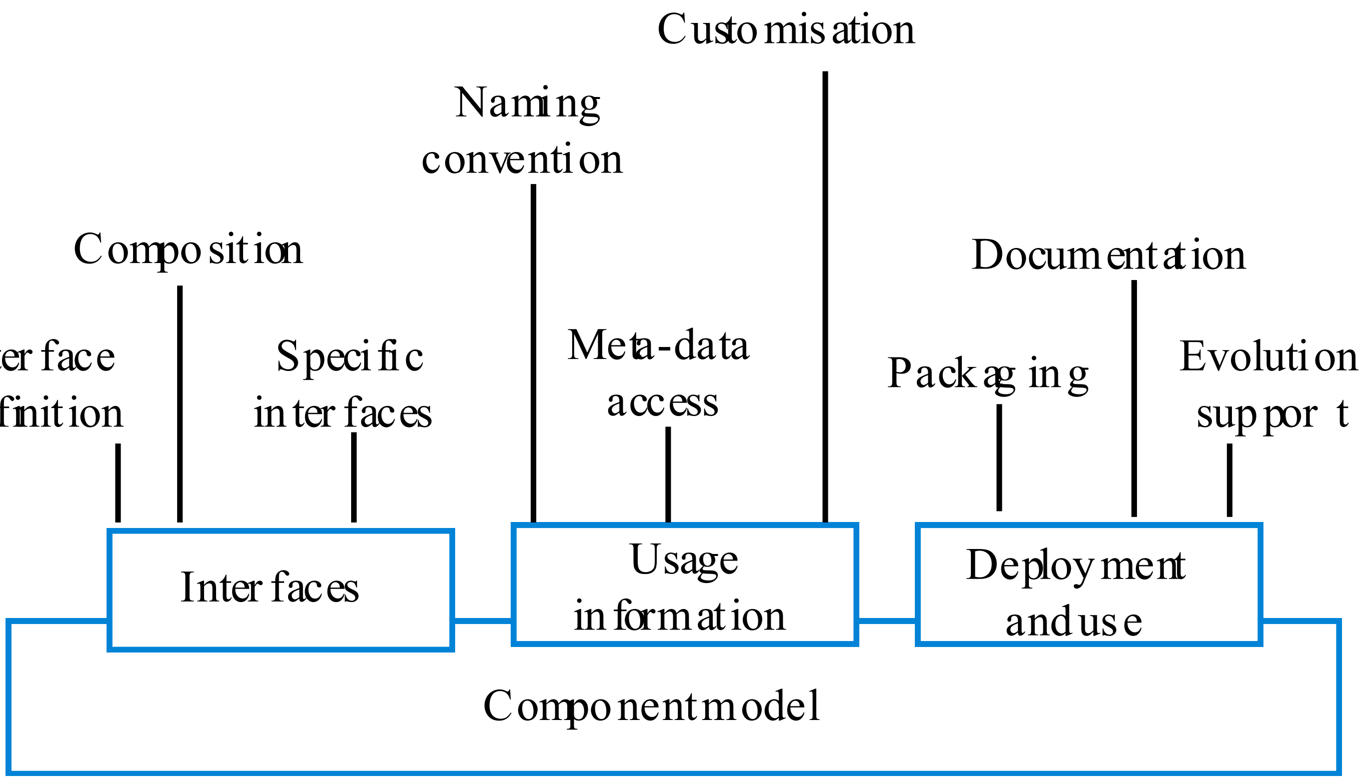


*Fig. 19.4 from [Sommerville]*

## Middleware support

- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide shared services for components:
  - Platform services that allow components written according to the model to communicate;
  - Horizontal services that are application-independent services used by different components.
- To use services provided by a componrnt model infrastructure, components are deployed in a container. This is a set of interfaces used to access the service implementations.

# Component model services

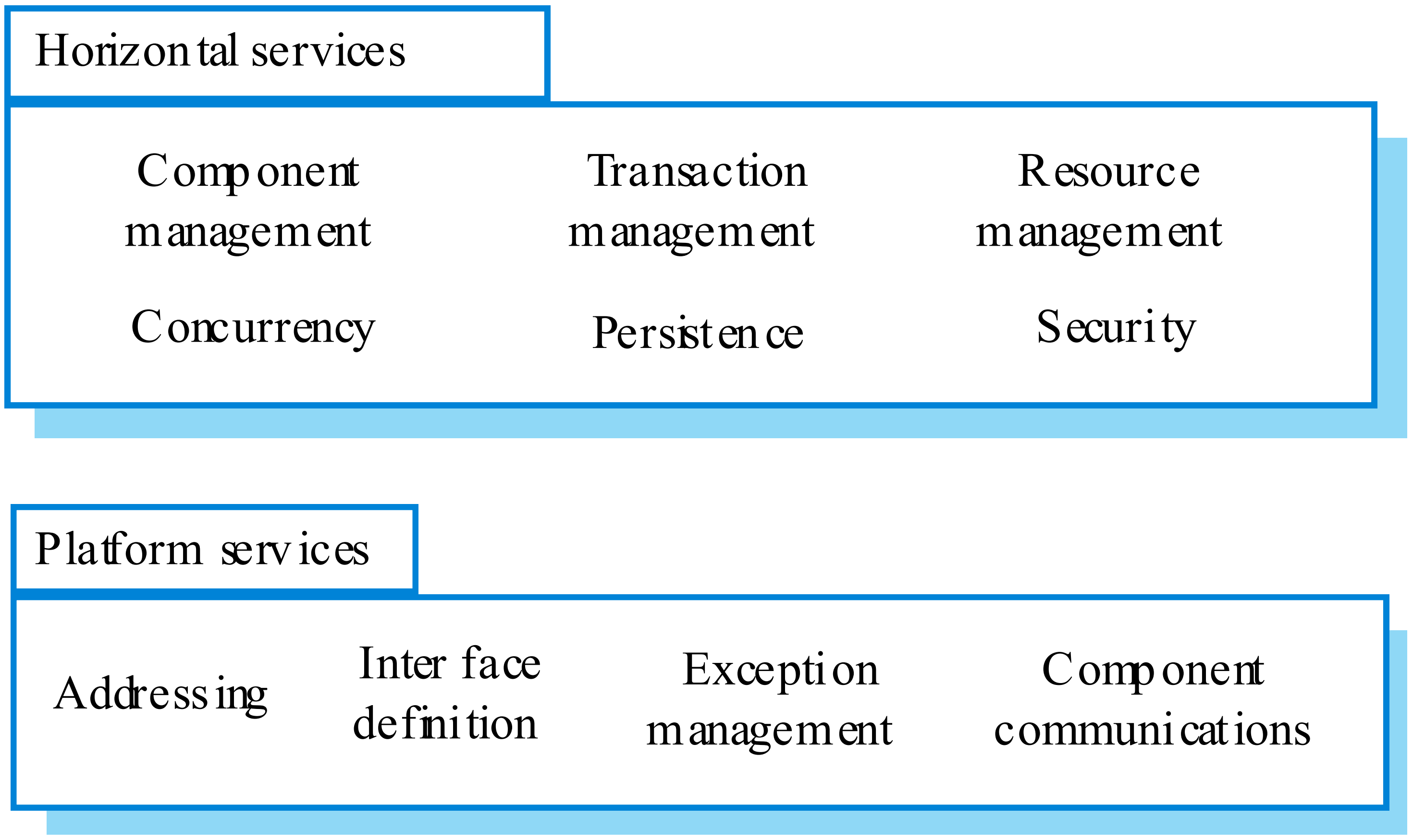
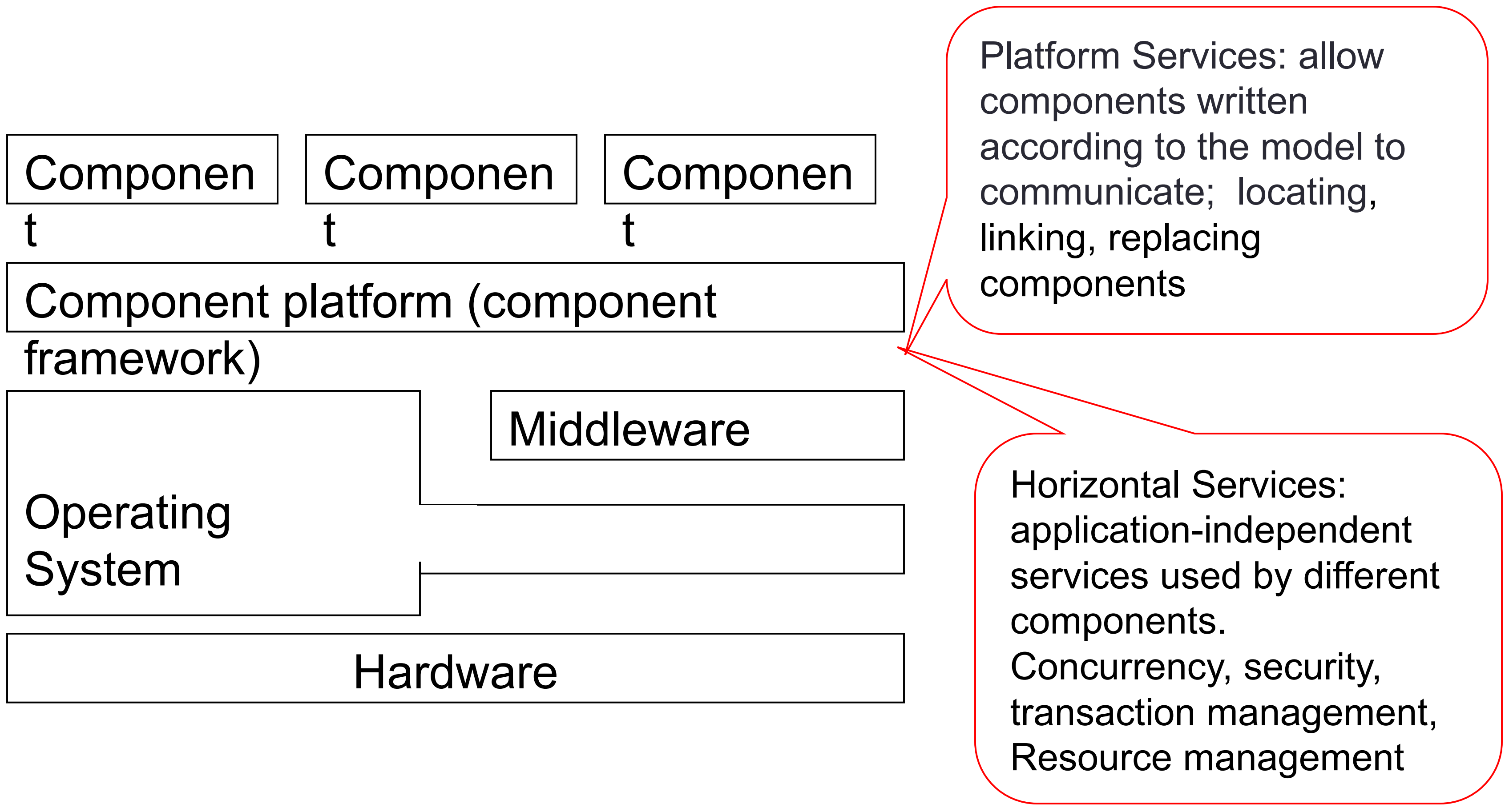


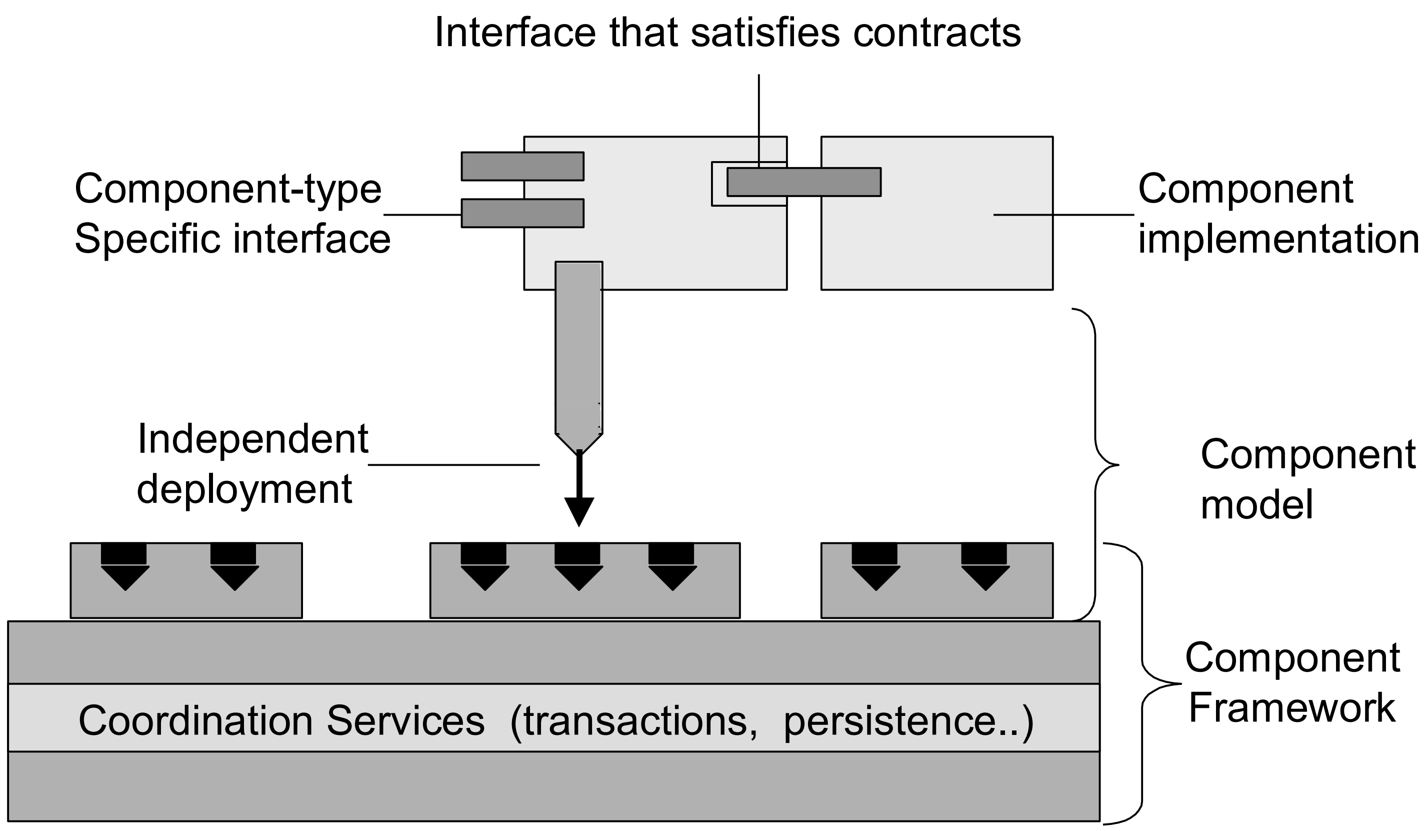
*Fig. 19.5 from [Sommerville]*

# Architecture of Component Models



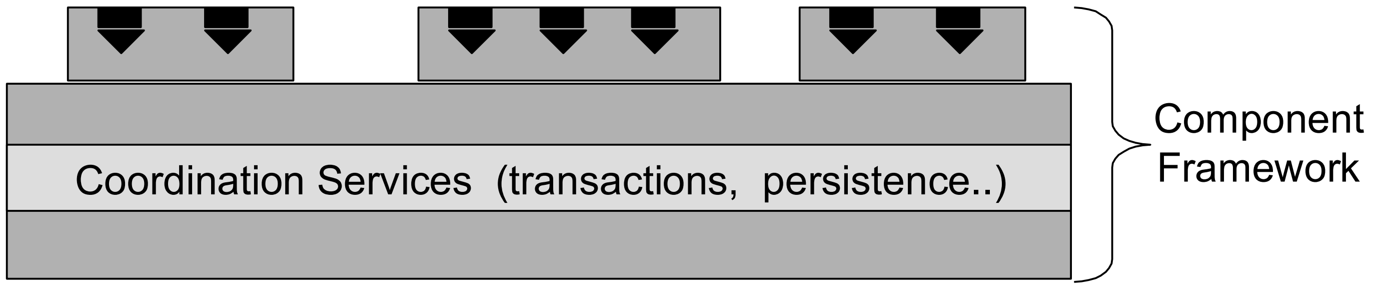
Componen
t
Componen
t
Componen
t
Component platform (component framework)
Operating System
Middleware
Hardware
Platform Services: allow components written according to the model to communicate; locating, linking, replacing components
Horizontal Services: application-independent services used by different components. Concurrency, security, transaction management, Resource management

# Relationships Between Concepts


Interface that satisfies contracts
Component-type Specific interface
Component implementation
Independent deployment
Component model
Component Framework
Coordination Services (transactions, persistence..)

## Component Frameworks

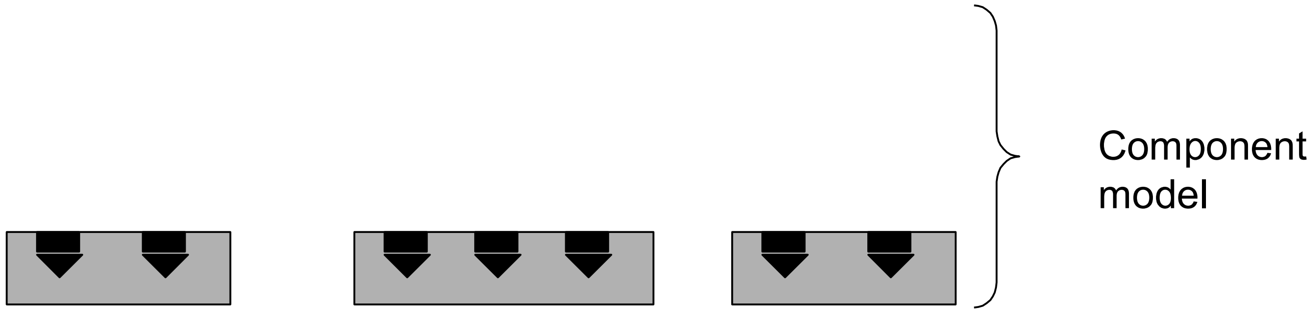
- While frameworks in general describe a typical and reusable situation at a model level, a *component framework* describes a “circuit-board” with empty slots into which components can be inserted to create a working instance.



The component framework forces components to perform their tasks via mechanisms controlle
by the framework

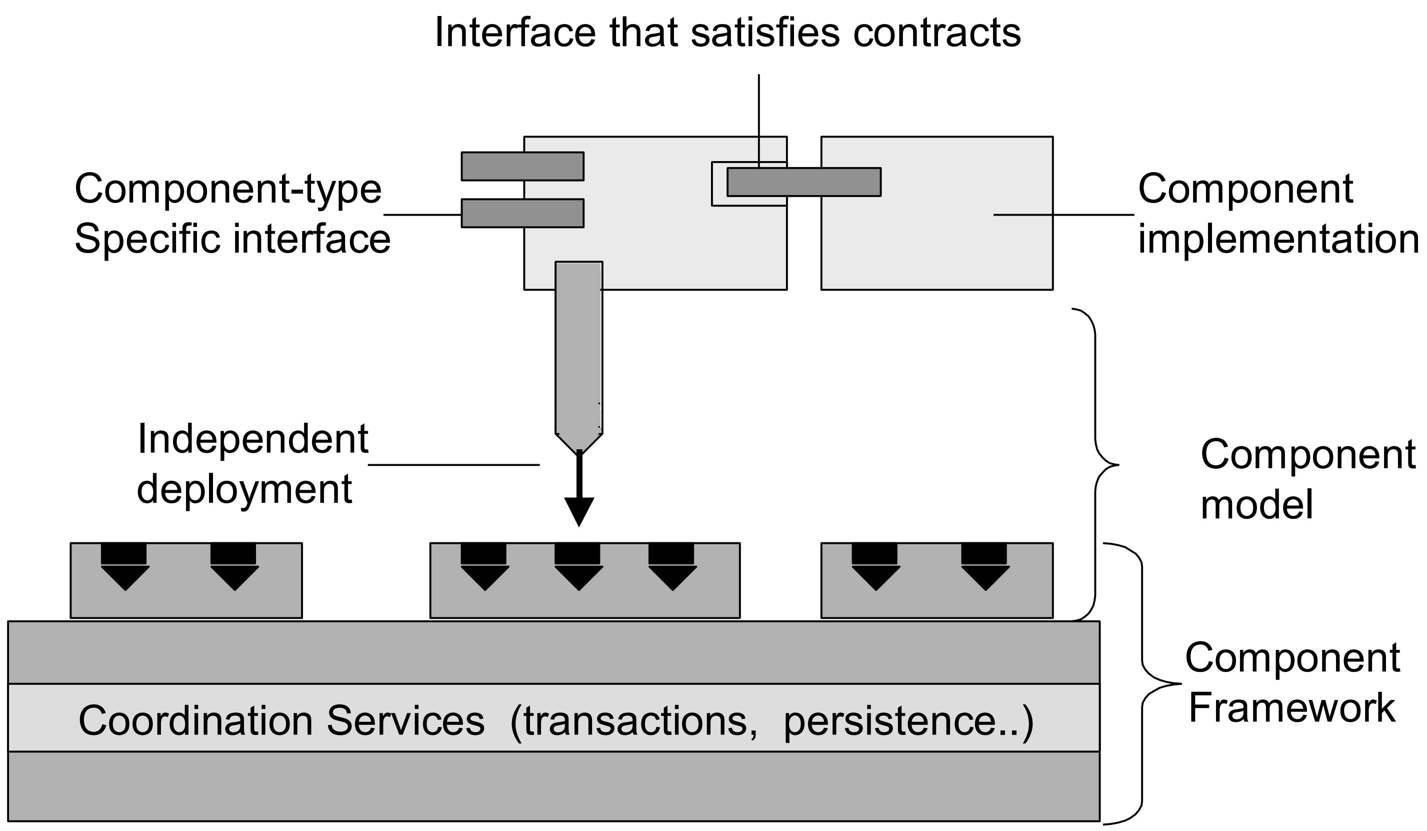
## Component Models

- The two concepts Component Models and Component Frameworks are sometimes intermixed.
- A *component model* defines a set of standards and conventions used by the component developer whereas a component framework is a support infrastructure for the component model.



Standards: component types, interfaces, allowable patterns of interaction

## Relationships Between Concepts


Interface that satisfies contracts
Component-type Specific interface
Component implementation
Independent deployment
Component model
Component Framework
Coordination Services (transactions, persistence..)

## Component Based Development – Summary

- CBSE is about:
  - Building a system by composing “entities”
  - Reusing “entities”
  - Maintaining a system by adding/removing/replacing “entities”
- What are the “entities” ?
  - Functions, modules, objects, components, services, ..
- Reusable “entities” are encapsulated abstractions : provided/required interfaces
- Composition of “entities” has to be supported by
  - Standards (componrnt mofdels)
  - Middleware (component framework)

# Content

## Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)
- Web services
- SOSE: Service-Oriented Software Engineering

## Italian restaurant analogy

- Restaurant provides food: a service
- After the order is taken, food is produced, served, …: service may consist of other services
- The menu indicates the service provided: a service description
- The order is written down, or yelled at, the cook: services communicate through messages

## Main ingredients

- Services
- Service descriptions
- Messages

- Implementation: through web services

## Other example

- Citizen looking for a house:
    - Check personal data ⇒ System X
    - Check tax history ⇒ System Y
    - Check credit history ⇒ System Z
    - Search rental agencies ⇒ System A,B
    - …

## What’s a service

- Platform-independent computational entity that can be used in a platform-independent way
- Callable entities or application functionalities accessed via exchange of messages
- Component capable of performing a task
- Often just used in connection with something else: SOA, Web services, …

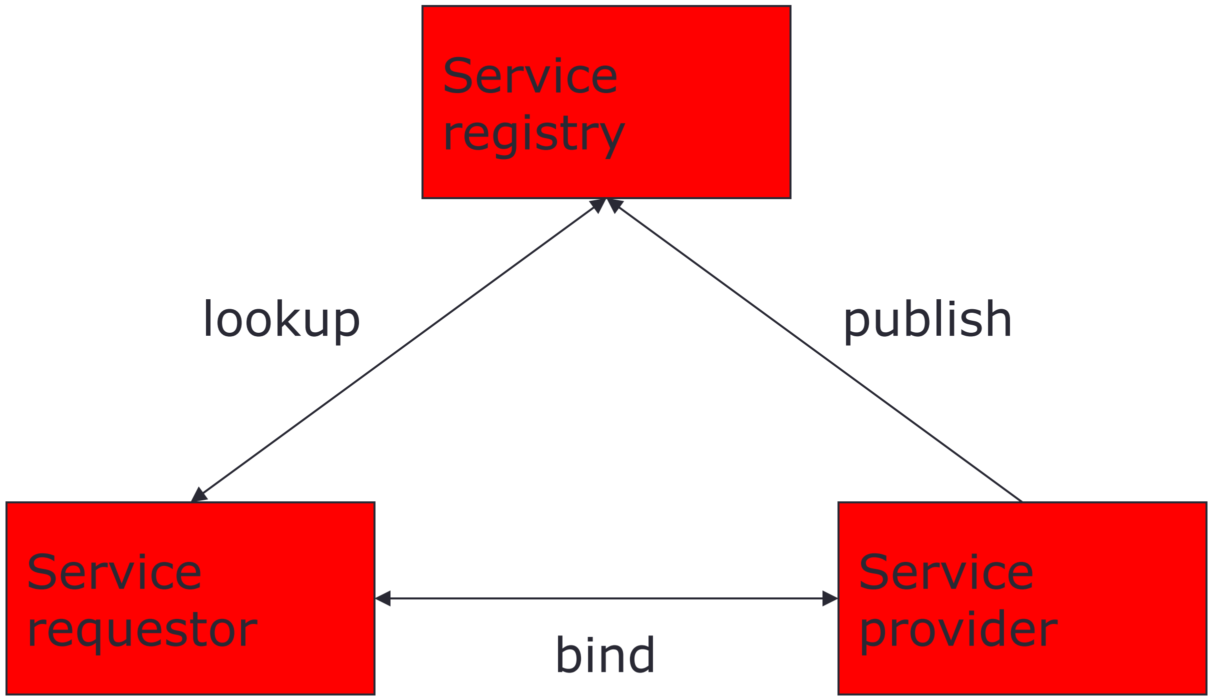
## What’s a service, cnt’d

- Shift from producing software to using software
  - You need not host the software
  - Or keep track of versions, releases
  - Need not make sure it evolves
  - Etc
- Software is “somewhere”, deployed on as-needed basis
- SaaS: Software as a Service

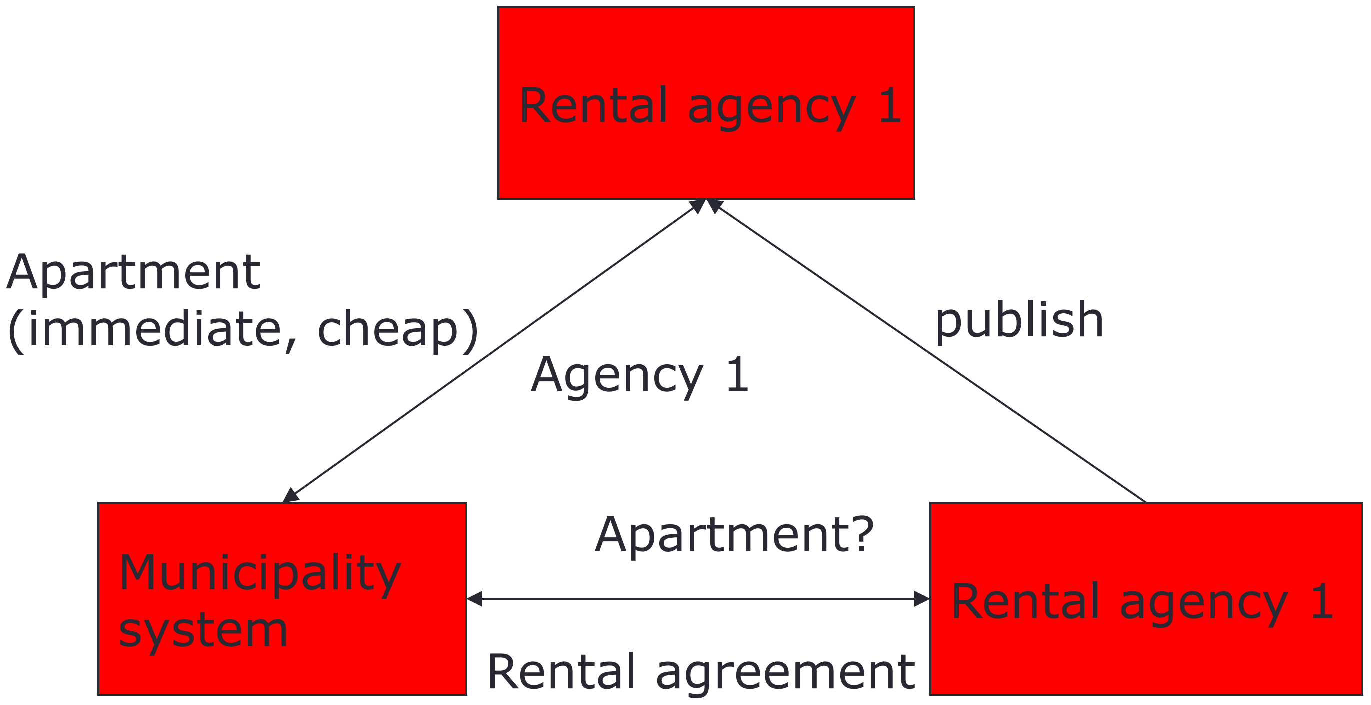
## Key aspects

- Services can be discovered
- Services can be composed to form larger services
- Services adhere to a service contract
- Services are loosely coupled
- Services are stateless
- Services are autonomous
- Services hide their logic
- Services are reusable
- Services use open standards
- Services facilitate interoperability

## Service discovery



# Service discovery



## Service discovery

- Discovery is dynamic, each invocation may select a different one
- Primary criterion in selection: contract
- Selection may be based on workload, complexity of the question, etc $\Rightarrow$ optimize compute resources
- If answer fails, or takes too long $\Rightarrow$ select another service $\Rightarrow$ more fault-tolerance

## Is discovery really new?

- Many design patterns loosen coupling between classes
- Factory pattern: creates object without specifying the exact class of the object.

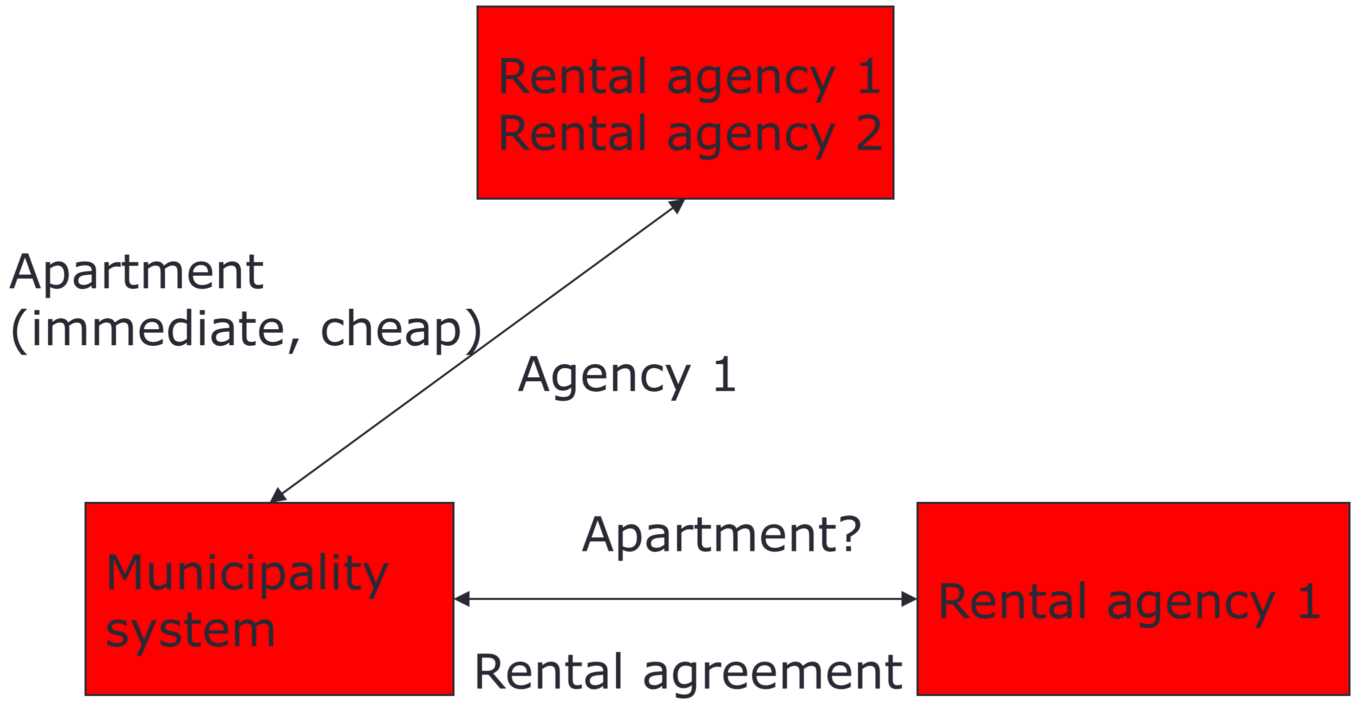
## Services can be composed

- Service can be a building block for larger services
- Not different from CBSE and other approaches

## Services adhere to a contract

- Request to registry should contain everything needed, not just functionality
- For “normal” components, much is implicit:
  - Platform characteristics
  - Quality information
  - Tacit design decisions
- Trust promises?
- Quality of Services (QoC), levels thereof
- Service Level Agreement (SLA)

## Service discovery



## Services are loosely coupled

- Rental agencies come and go
- No assumptions possible
- Stronger than CBSE loose coupling

## Services are stateless

- Rental agency cannot retain information: it doesn’t know if and when it will be invoked again, and by whom

## Services are autonomous, hide their logic

- Rental agency has its own rules on how to structure its process
- Its logic does not depend on the municipality service it is invoked by
- This works two ways: outside doesn’t know the inside, and vice versa

## Services are reusable

- Service models a business process:
    - Not very fine grained
    - Collecting debt status from one credit company is not a service, checking credit status is
- Deciding on proper granularity raises lots of debate

## Service use open standards

- Proprietary standards $\Rightarrow$ vendor lockin

- There are lots of open standards:
  - How services are described
  - How services communicate
  - How services exchange data
  - etc

## Services facilitate interoperability

- Because of open standards, explicit contracts and loose coupling
- Classical CBSE solutions pose problems:
  - Proprietary formats
  - Platform differences
  - Etc
- Interoperability within an organization (EAI) and between (B2B)

## Overview

- Services, service description, service communication

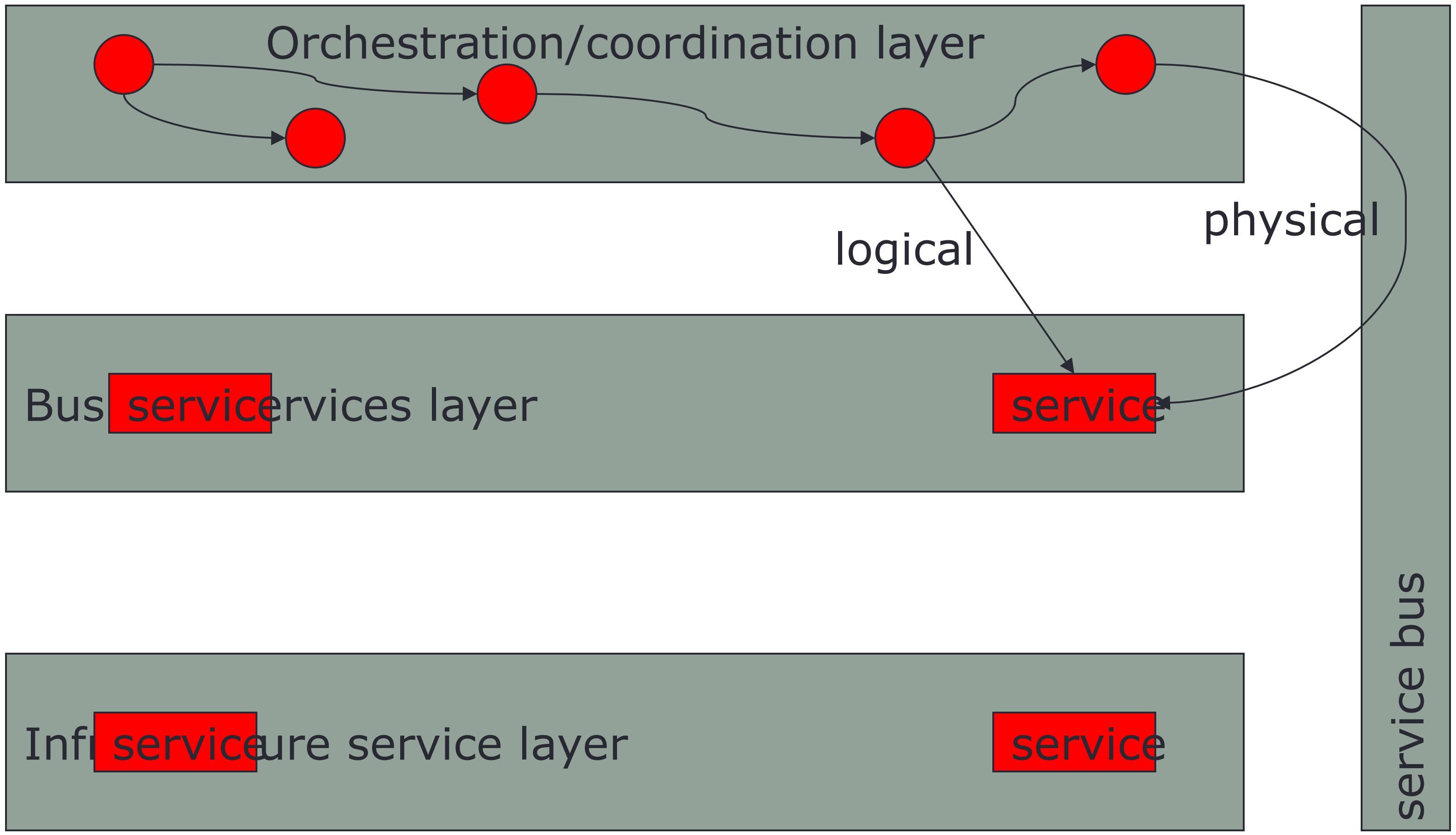
- Service-Oriented Architecture (SOA)

- Web services
- SOSE: Service-Oriented Software Engineering

## Service-Oriented Architecture

- Architecture:
  - the fundamental organization of a system in its components, their relationships to each other and to the environment and the principles guiding its design and evolution
- SOA: Any system made out of services?

## What is SOA?


Orchestration/coordination layer
logical
physical
Business services layer
service
service
Infrastructure service layer
service
service
service bus

## Service bus

- Event-based messaging engine
- Origin: EAI, solve integration problems
- Often takes care of:
  - Mediation: protocol translation, data transformation, etc
  - Quality of Service issues: security, reliable delivery of messages, etc
  - Management issues: logging, audit info, etc.
  - Service discovery
- Can be central (broker, hub), or decentral (smart endpoints)

## Service coordination

- Orchestration: central control
- Choreography: decentral control

## Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)

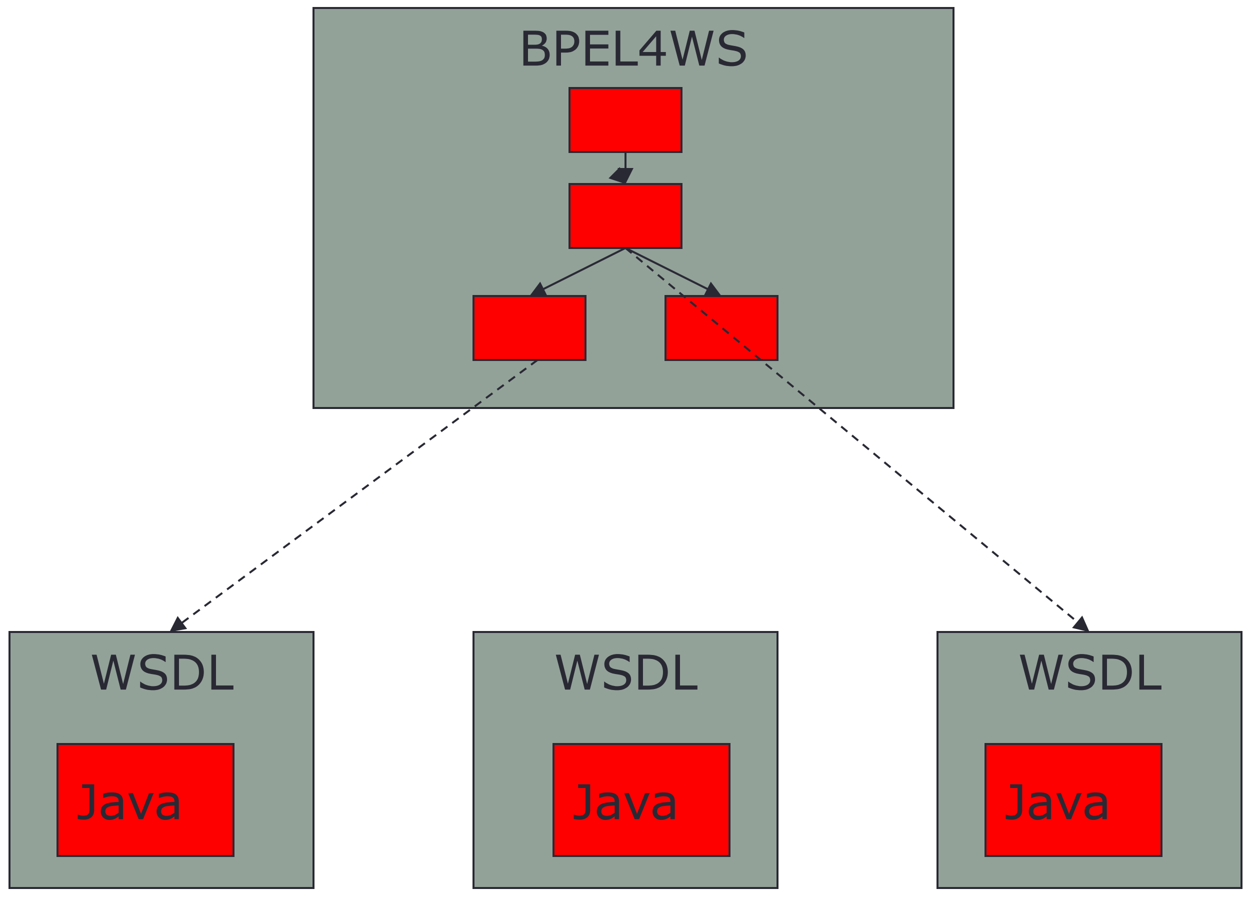
- Web services

- SOSE: Service-Oriented Software Engineering

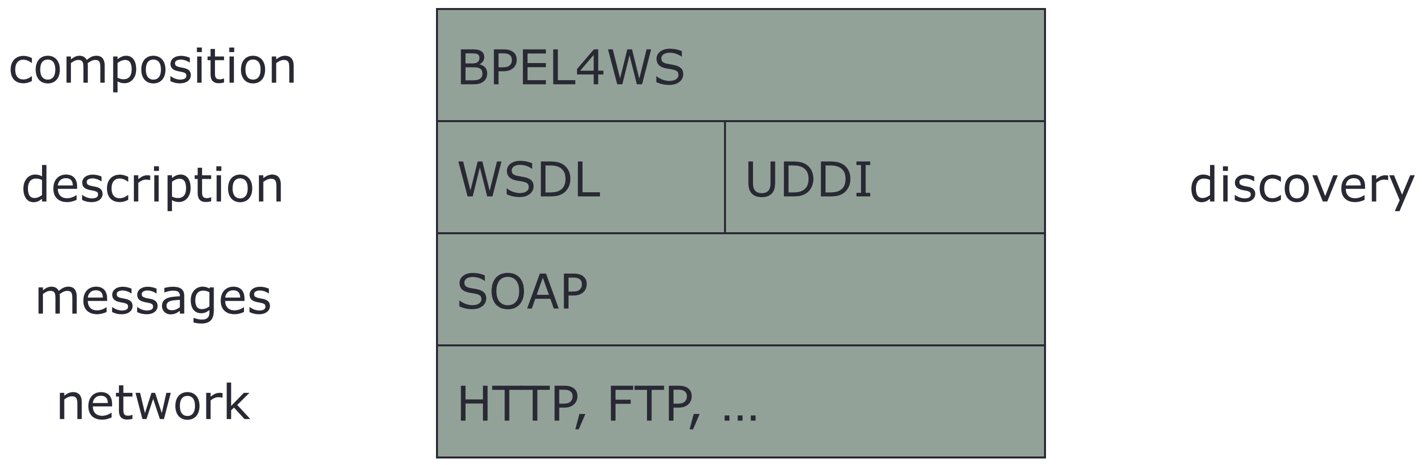
## Web services

- Implementation means to realize services
- Based on open standards:
  - XML
  - SOAP: Simple Object Access Protocol
  - WSDL: Web Services Description Language
  - UDDI: Universal Description, Discovery and Integration
  - BPEL4WS: Business Process Execution Language for Web Services
- Main standardization bodies: OASIS, W3C

## Coordination of Web services



# Web services stack



## XML

- Looks like HTML
- Language/vocabulary defined in schema: collection of trees
- Only syntax
- Semantic Web, Web 2.0: semantics as well: OWL and descendants

## SOAP

- Message inside an envelope
- Envelop has optional header (~address), and mandatory body: actual container of data
- SOAP message is unidirectional: it’s NOT a conversation

## WSDL

- Four parts:
  - Web service interfaces
  - Message definitions
  - Bindings: transport, format details
  - Services: endpoints for accessing service. Endpoint = (binding, network address)