



28th International Conference
on Software Engineering

Tutorial H11

Software Component Models

Kung-Kiu Lau, the University of Manchester, UK

Schedule

- 2:00–3:30 Introduction
- Category 1: JavaBeans
 - Category 2: EJB, COM, .NET, CCM, web services
- 3:30–4:00 Break
- 4:00–5:30 Category 3: Koala, SOFA, KobrA
- Category 4: ADLs, UML 2.0, PECOS, Pin, Fractal

Disclaimers:

We're not presenting user manuals!
Material continually being updated

Introduction

Software Component Models

A software component model defines:

- what components are
 - syntax of components
 - semantics of components
- how to compose components

Current Component Definitions

- Szyperski:

“A software component is a **unit of composition** with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

- Meyer:

“A component is a **software element (modular unit)** satisfying the following conditions:

1. It can be used by other software elements, its ‘clients’.
2. It possesses an official usage description, which is sufficient for a client author to use it.
3. It is not tied to any fixed set of clients.”

Current Component Definitions (Continued)

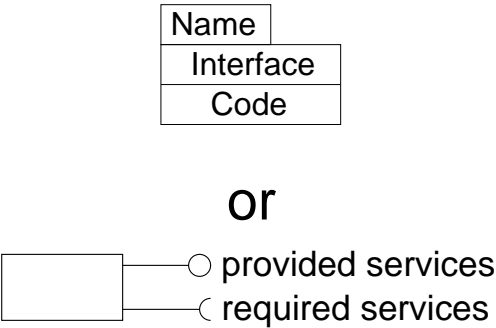
- **Heineman and Council**

“A [component is a] **software element** that conforms to a **component model** and can be independently deployed and composed without modification according to a composition standard.”

Comparison wrt **component models**:

Definition	Based on CM	Defines CM
Szyperski	No	No
Meyer	No	No
Heineman and Council	Yes	No

Current Software Component Models

	Components	Composition
Semantics		??
Typical examples	<ul style="list-style-type: none"> • objects • architectural units 	<ul style="list-style-type: none"> • method calls • ADL connectors

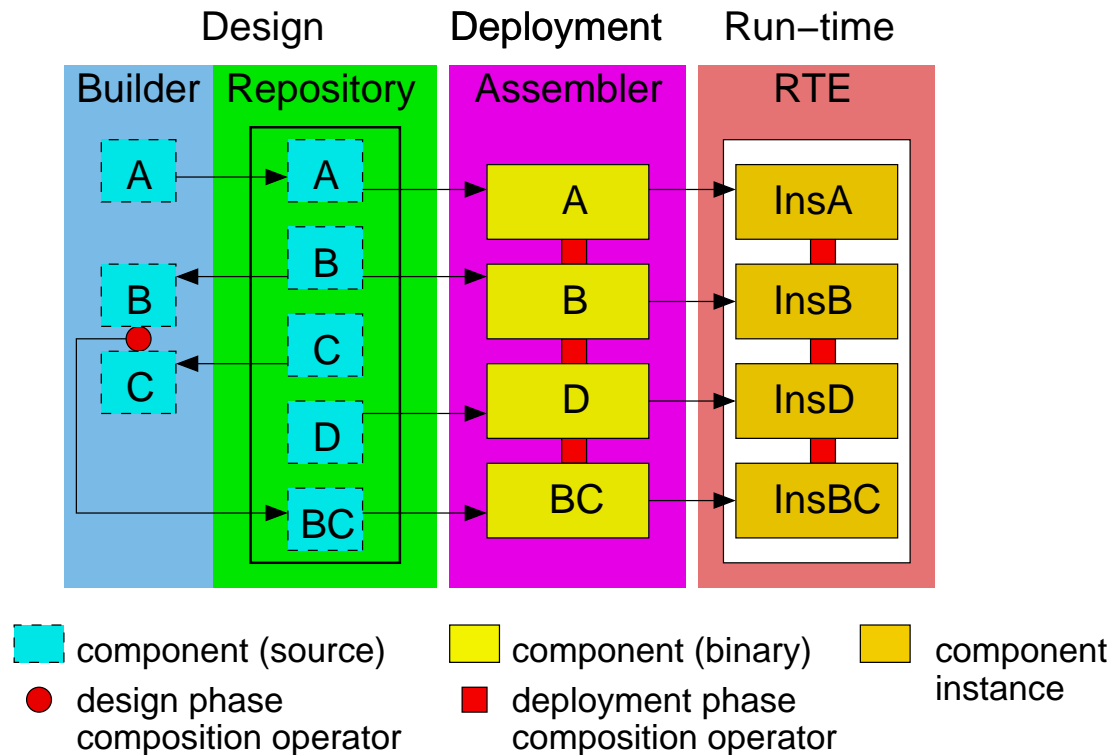
Categories based on Component Syntax

Component Syntax	Models
Object-oriented Programming Languages	JavaBeans, EJB
Programming Languages with IDL mappings	COM, .NET, CCM, web services, Fractal
Architecture Description Languages	ADLs, UML2.0, KobrA, Koala, SOFA, PECOS, Pin

Categories based on Component Semantics

Component Semantics	Models
Classes	JavaBeans, EJB
Objects	COM, .NET, CCM, web services, Fractal
Architectural Units	ADLs, UML2.0, KobrA, Koala, SOFA, PECOS, Pin

An Idealised Component Life Cycle

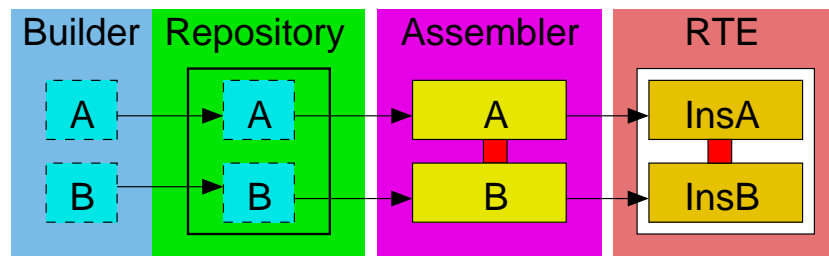


An Idealised Component Life Cycle (Continued)

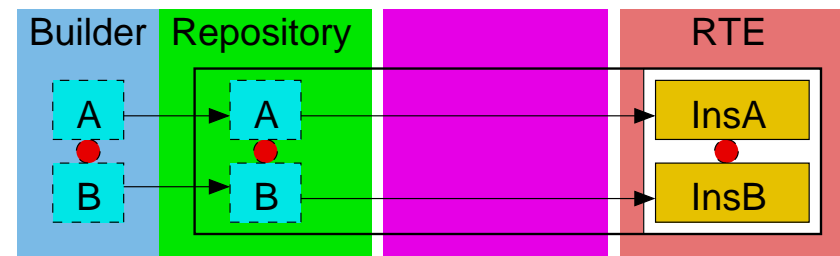
This life cycle reflects CBSE desiderata:

components pre-exist	repository
components produced & used independently	builder & assembler (+ repository)
components can be copied and instantiated	design & deployment + run-time phases
composites can be made and used for further composition	composites in repository

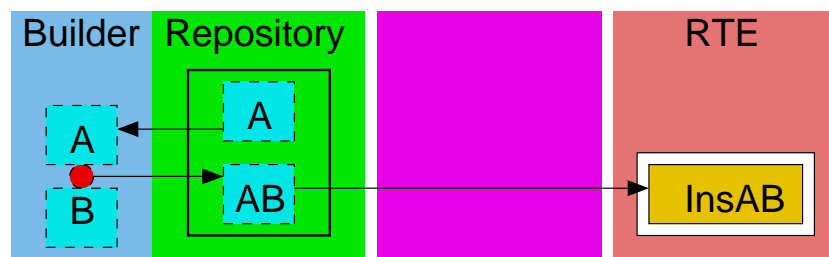
Categories based on Composition



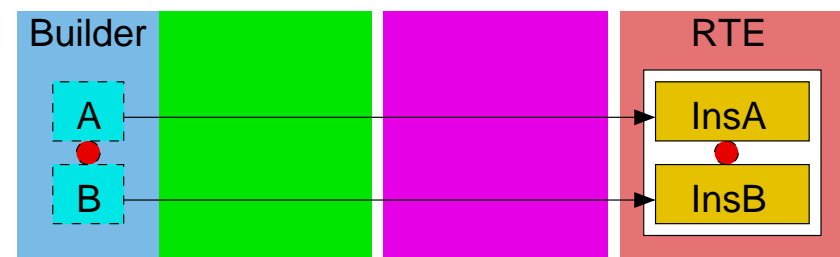
Category 1
(JavaBeans)



Category 2
(EJB, COM, .NET, CCM, web services)



Category 3
(Koala, SOFA, Kobra)



Category 4
(ADLs, UML2.0, PECOS, Pin, Fractal)

Category 1

JavaBeans

JavaBeans: Components

In **JavaBeans**, a component is a **bean**, which is just any **Java class** that has:

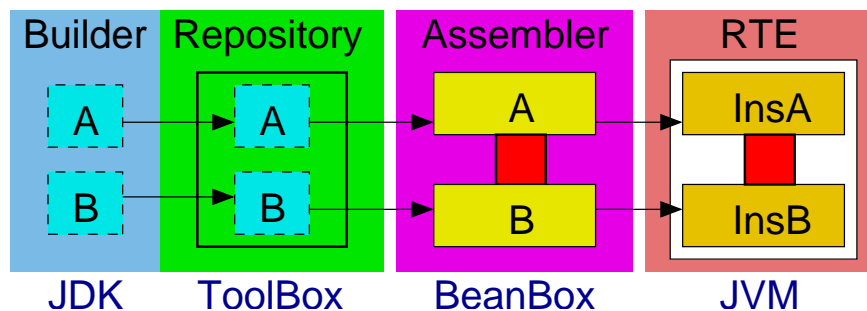
- **methods**
- **events**
- **properties**

A **bean** is intended to be constructed and manipulated in a **visual builder tool**.

JavaBeans: Builder, Repository, Assembler

Individual beans are constructed as Java classes in a JDK, and deposited in the ToolBox of the BDK.

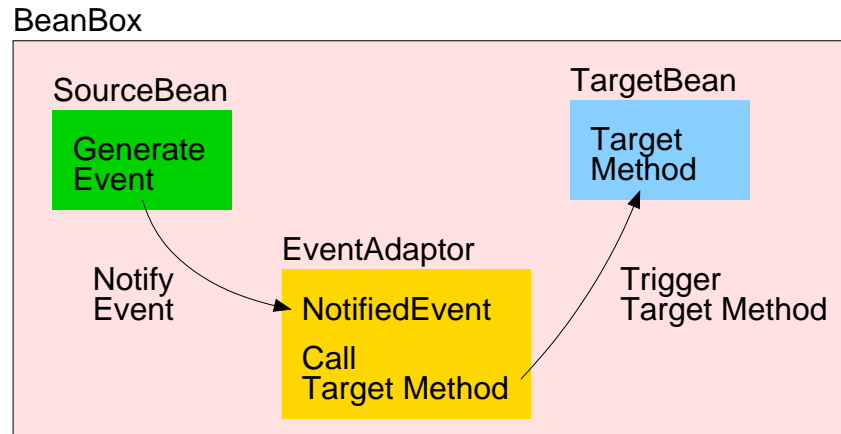
- **JDK** is the **builder** for Java beans
- **ToolBox** of the BDK is the **repository** (No composition)
- **Beanbox** is the **assembler** (Composition of bean instances)



A = BeanA (JAR file)
 B = BeanB (JAR file)
 InsA = BeanA instance
 InsB = BeanB instance
 ■ = Adaptor object

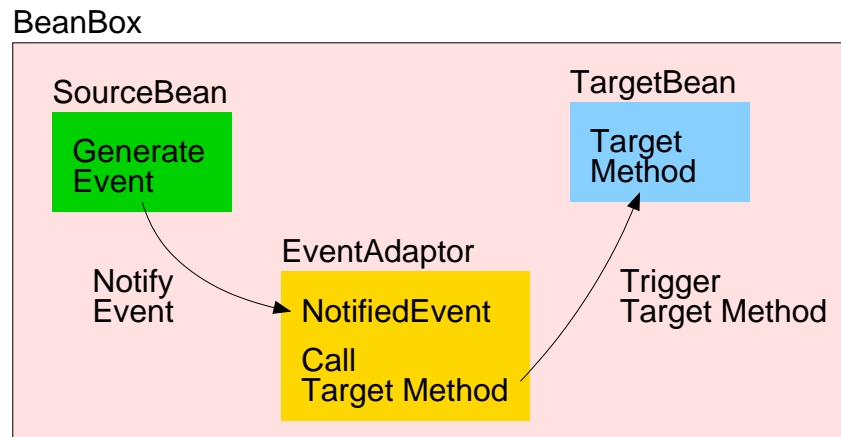
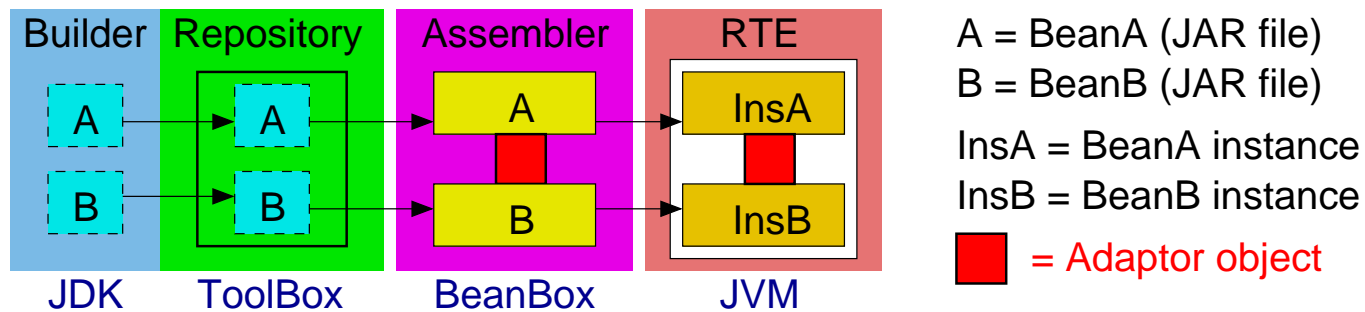
JavaBeans: Composition

In deployment phase, bean instances can be composed via event delegation

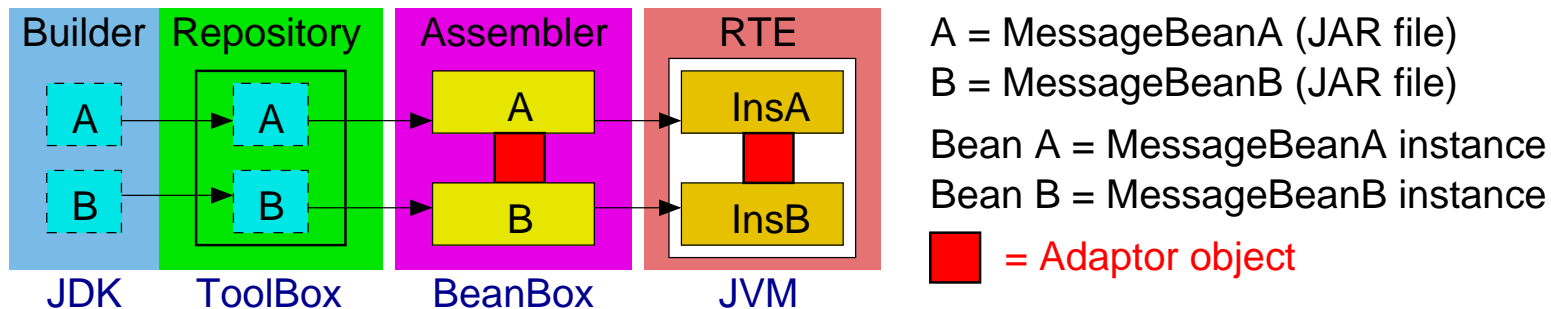


- a bean **'composes'** with another bean by sending a message through delegation of events
- **BeanBox** automatically generates, compiles, and loads event adaptor classes for logistics of events

JavaBeans: Summary



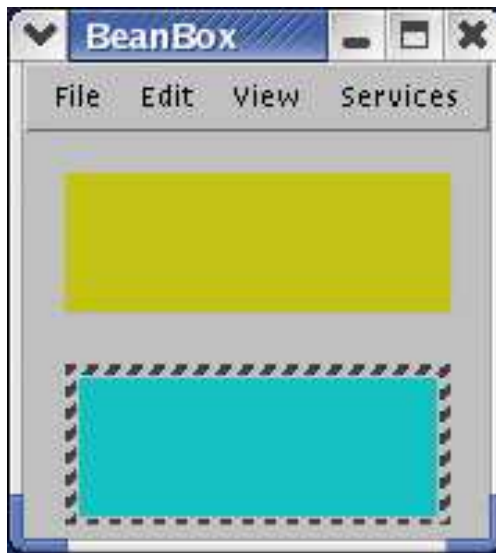
JavaBeans: Example



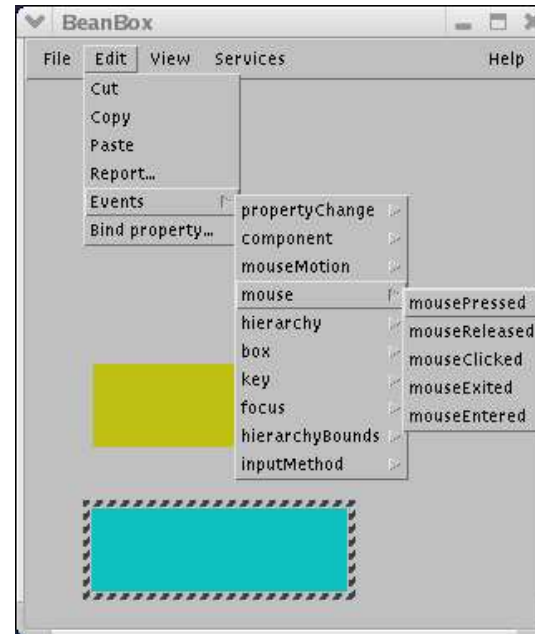
Consider a simple bean `MessageBean` that displays a message when it is notified of the event 'mousePressed' by another bean:

- It is a **Java class** that has a **method** for displaying a message
- It has **mouse events** such as 'mousePressed'
- It displays a message that is a **property** which can be set by the programmer

JavaBeans: Example (Continued)

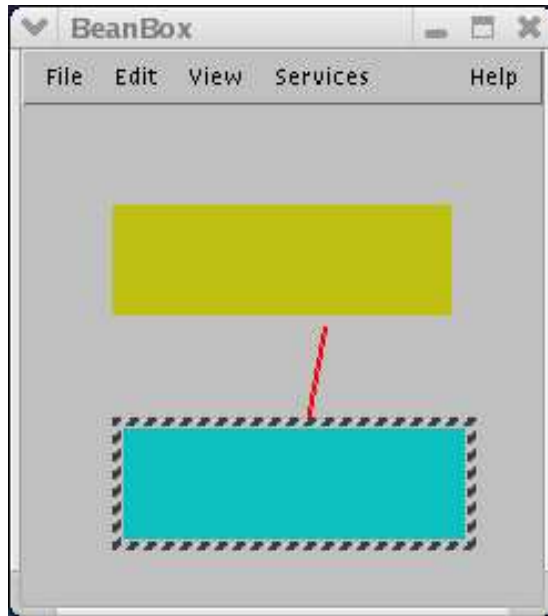


Beans A (yellow) & B (blue)
(Bean B is selected)

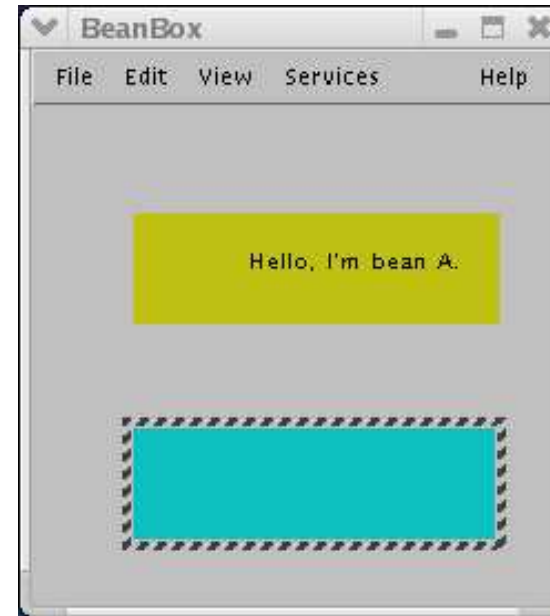


Choosing source event in B

JavaBeans: Example (Continued)



Bean B is linked to Bean A
by choosing target event in Bean A



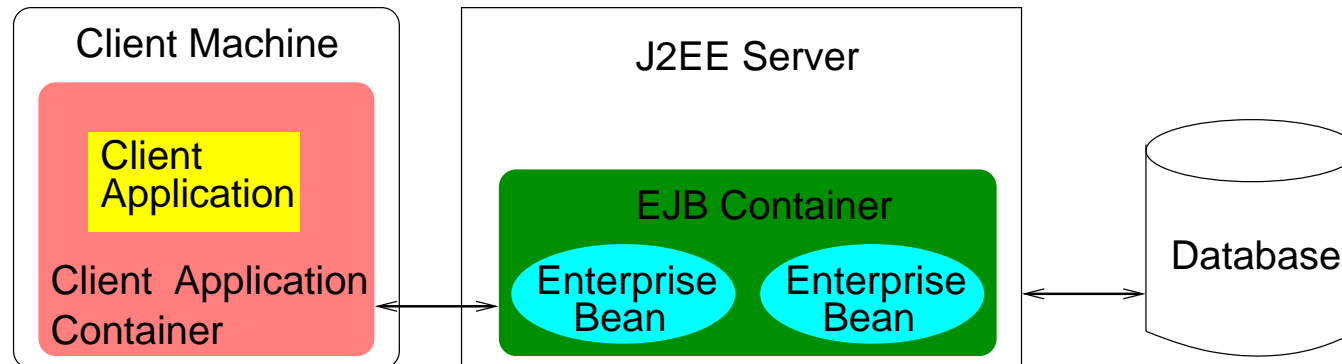
Bean A displays message
when mouse pressed in Bean B

Category 2

EJB, COM, .NET, CCM, web services

Enterprise JavaBeans (EJB): Components

In **EJB**, a component is a an **enterprise Java bean** with **two Java interfaces**:



- an enterprise Java bean is a **Java class** in an **EJB container** on a **J2EE server**
- an **EJB container** uses the **interfaces** to manage and execute the **Java class** and its instances.

Enterprise JavaBeans (EJB): Components (Continued)

For an EJB:

- its **Java class** defines the **methods** of the bean
- its **interfaces** expose the capabilities of the bean and provide all the methods needed for (remote) client applications to access the bean (over a network)
 - its **home interface** represents the **life-cycle methods** of the bean such as *create*, *destroy* and *locate* a bean instance
 - its **remote interface** represents the **tasks** performed by the bean

Enterprise JavaBeans (EJB): Components (Continued)

There are 3 kinds of EJBs:

- **Entity beans**

Entity beans model business **data**; they are Java objects that cache **database information**.

- **Session beans**

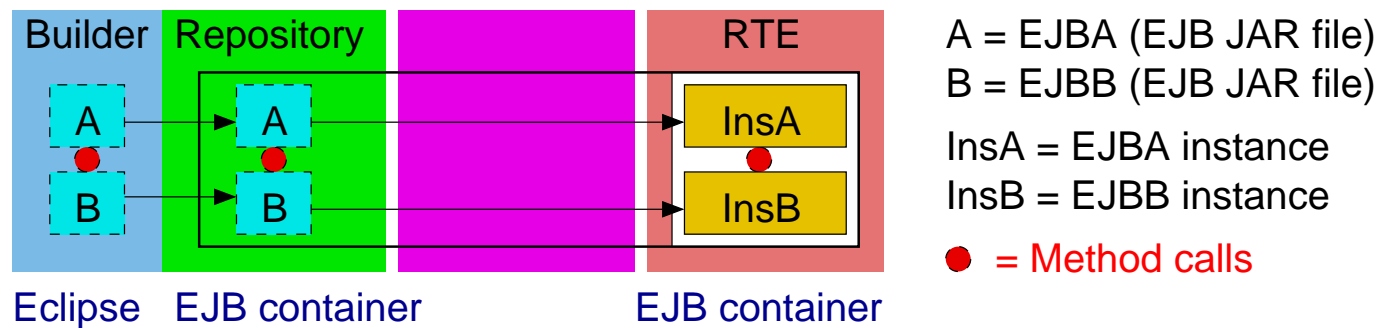
Session beans model business **processes**; they are Java objects that act as **agents performing tasks**.

- **Message-driven beans**

Message-driven beans model **message-related** business **processes**; they are Java objects that act as **message listeners**.

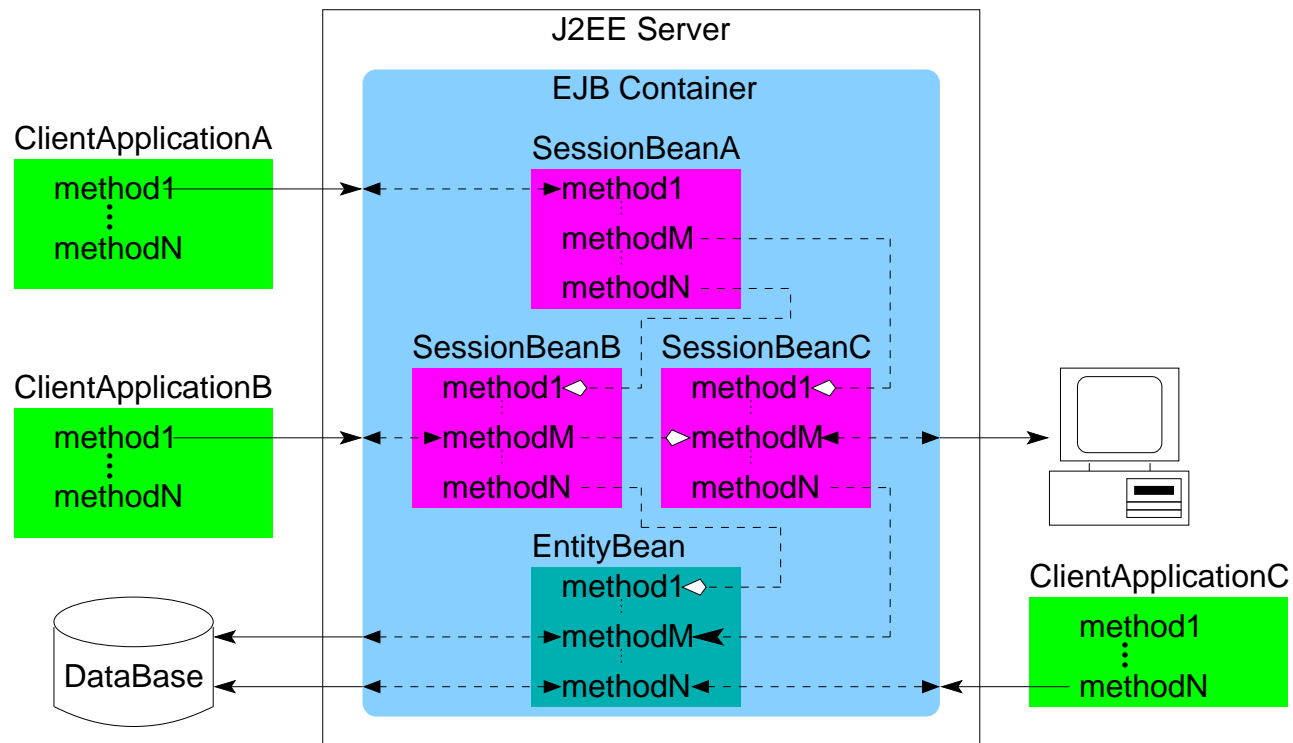
EJB: Builder, Repository

- JDK or Eclipse is the **builder** for EJB (Composition of beans)
- An EJB container is the **repository**
- There is **no assembler**

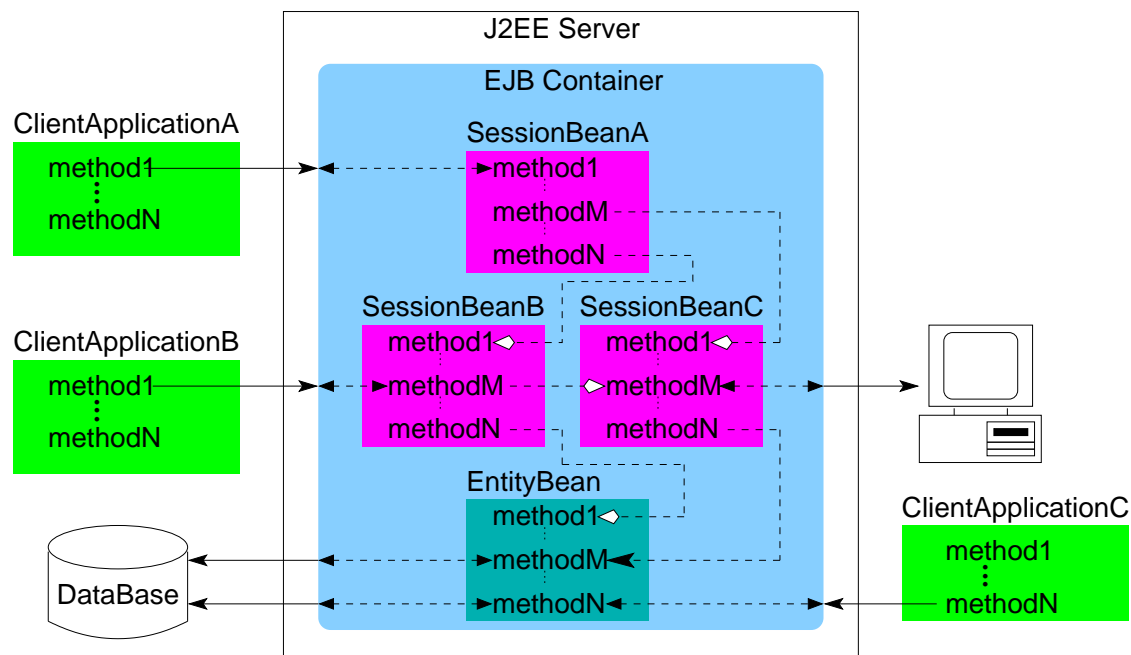
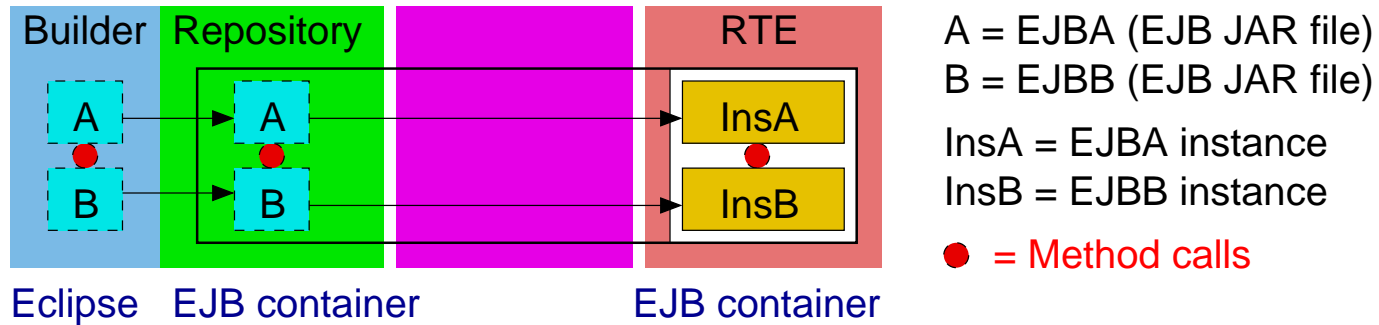


EJB: Composition

In **design phase**, enterprise beans are **composed** by **method** and **event delegation**



EJB: Summary



EJB: Example

Consider a book store which wishes to maintain a database of its book stock.

The **table of books** in the database can be represented as an **entity bean BookBean** that consists of **one class** and **two interfaces**:

- **BookBean** is the **Java class** that defines the **methods**
- **BookHome** is the **home interface**
- **Book** is the **remote interface**

Each **instance** of **BookBean** represents a **row of the table of books** in a database.

EJB: Example (Continued)

BookHome
+create(isbn : String) : Book +findByPrimaryKey(isbn : String) : Book

Book
+getBookName() : String +setBookName(bookname : String) : void +getAuthor() : String +setAuthor(author : String) : void +getPublisher() : String +setPublisher(publisher : String) : void +getPrice() : double +setPrice(price : double) : void

BookBean
+isbn : String +bookname : String +author : String +publisher : String +price : double -ctx : EntityContext
+ejbCreate(isbn : String) : String +ejbPostCreate(isbn : String) : void +getBookName() : String +setBookName(bookname : String) : void +getAuthor() : String +setAuthor(author : String) : void +getPublisher() : String +setPublisher(publisher : String) : void +getPrice() : double +setPrice(price : double) : void +setEntityContext(ctx : EntityContext) : void +unsetEntityContext() : void +ejbActivate() : void +ejbPassivate() : void +ejbLoad() : void +ejbStore() : void +ejbRemove() : void

EJB: Example (Continued)

To construct the book store application, we need a **session bean** **BookStoreBean** that consists of **one class**, **two interfaces** (and a helper class):

- **BookStoreBean** is the **Java class** that defines the **methods**
- **BookStoreHome** is the **home interface**
- **BookStore** is the **remote interface**
- (Books is the helper class)

BookStoreBean is used to add details of a set of books into the table of books in the database.

EJB: Example (Continued)

BookStoreHome
+create() : BookStore

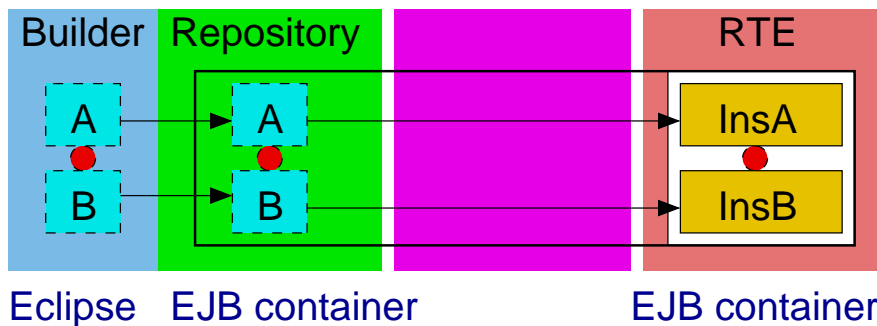
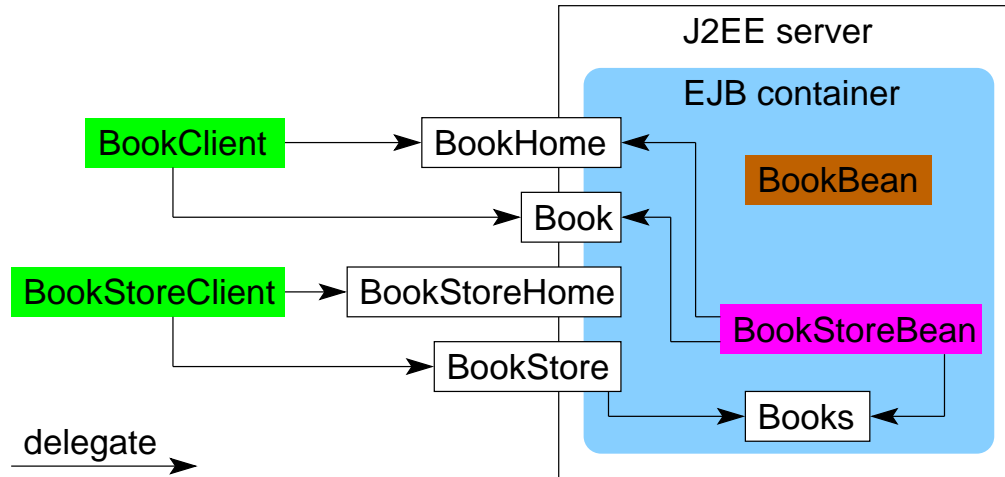
BookStore
+addBook(books : Books[]) : void

Books
+isbn : String
+bookname : String
+author : String
+publisher : String
+price : double
+toString() : String

BookStoreBean
-c : SessionContext
+addBook(boos : Books[]) : void
+setSessionContext(c:SessionContext):void
+ejbCreate() : void
+ejbActivate() : void
+ejbPassivate() : void
+ejbRemove() : void

EJB: Example (Continued)

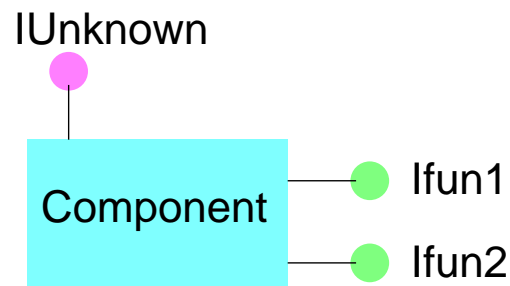
The book store application is assembled from the **BookBean** and the **BookStoreBean**.



- A = BookBean (EJB JAR file)
- B = BookStoreBean (EJB JAR file)
- InsA = BookBean instance
- InsB = BookStoreBean instance
- = Method calls

Component Object Model (COM): Components

In **Microsoft COM**, a component is a **unit of compiled code** on a **COM server**.

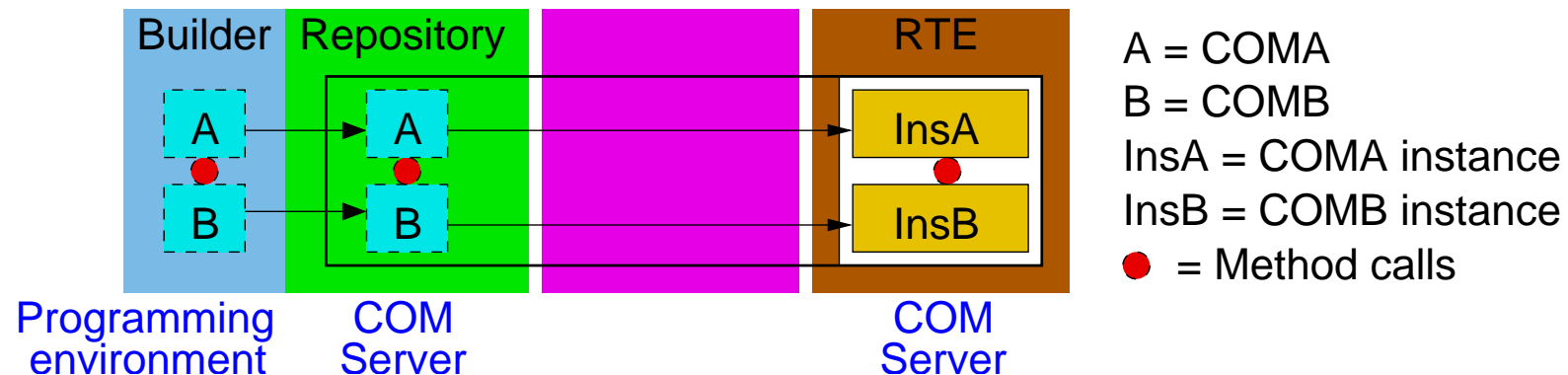


- **Services** in a component are **invoked** via **pointers** to the **functions** that implement them
- For each service provided there is an **interface** (a COM component can implement **multiple interfaces**)
- **COM interfaces** are specified in **Microsoft IDL**
- Every component must implement an **IUnknown** interface

COM: Builder, Repository

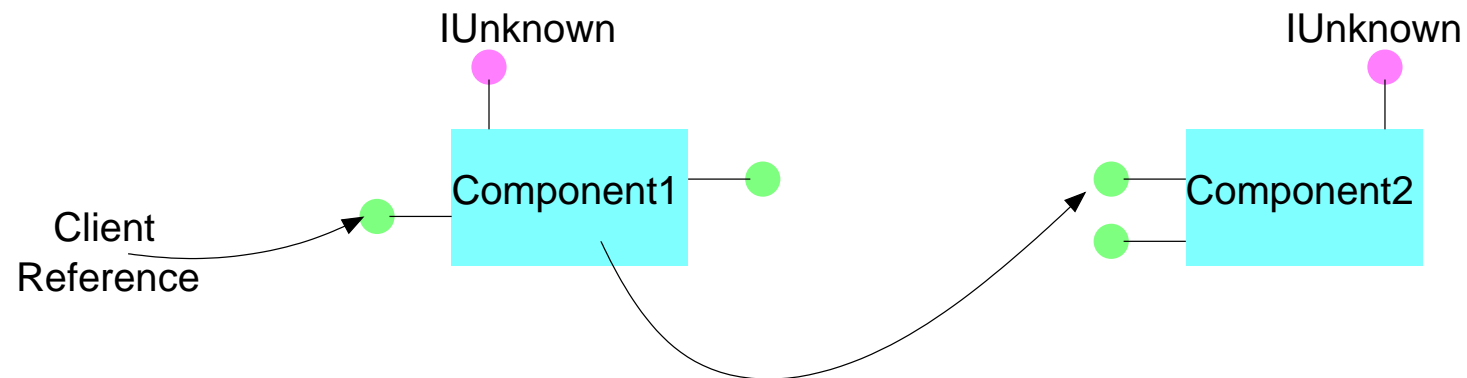
COM components are constructed in a programming environment such as Microsoft Visual Studio .NET

- The programming environment is the builder
- The COM server is the repository
- There is no assembler

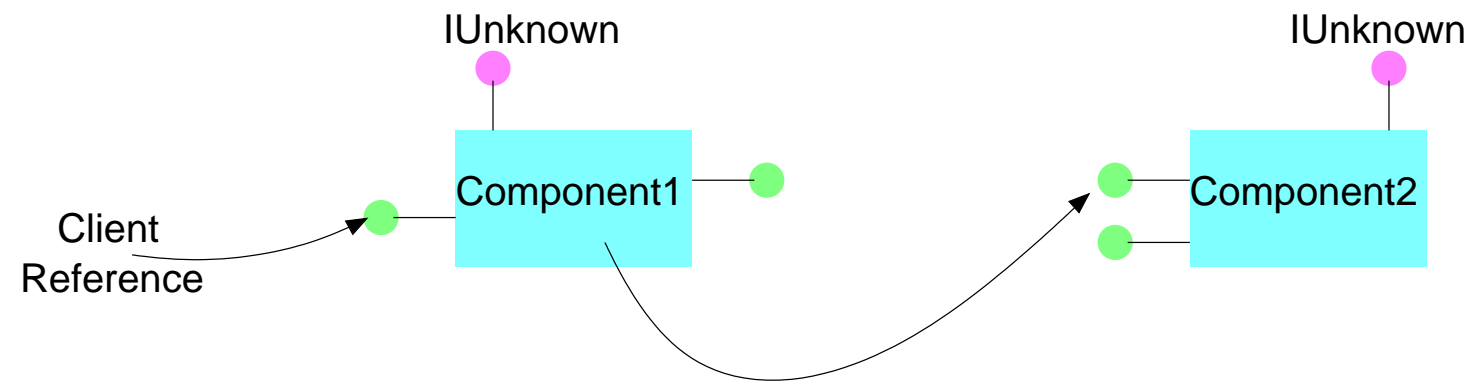
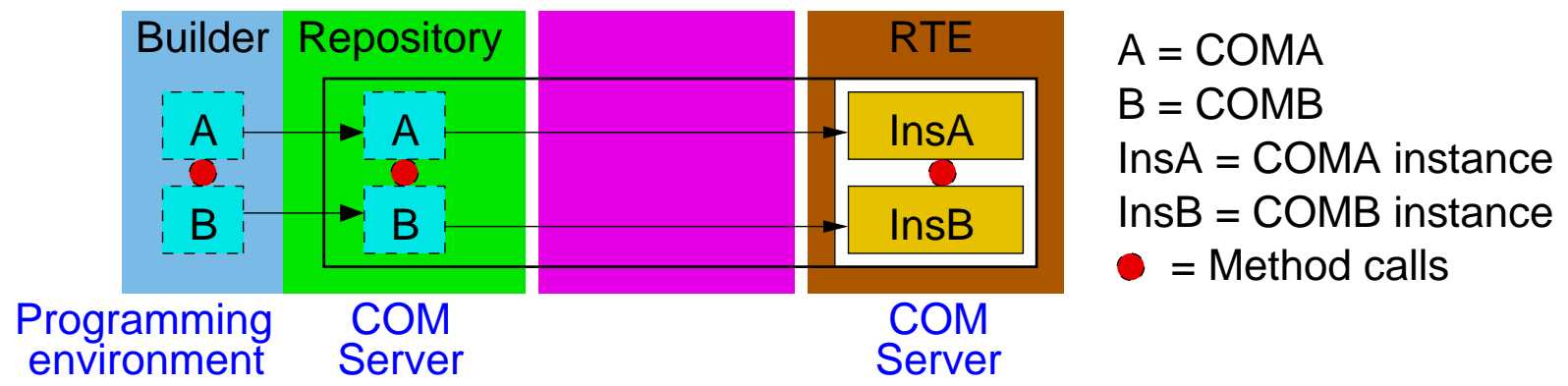


COM: Composition

In **design phase**, COM components are **composed** by **method calls** via **interface pointers**

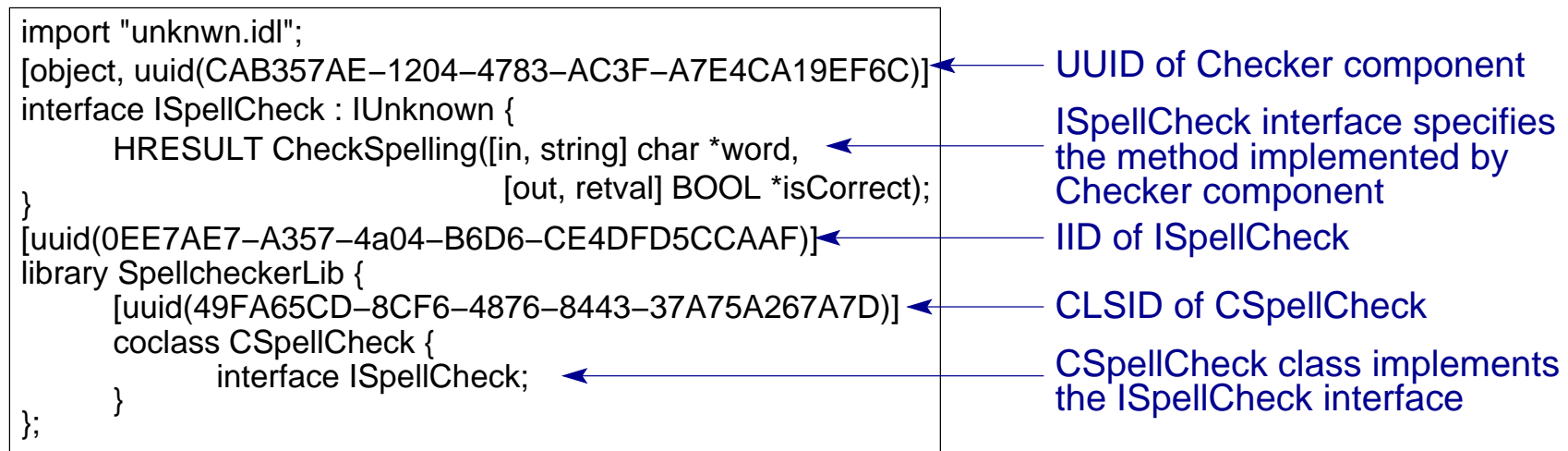


COM: Summary



COM: Example

Consider a **spell checker system** that comprises a **checker component** and a **dictionary component**.



Checker component interface -- ISpellCheck

A “library” is an interface glued with a coclass, e.g. the “library” of ISpellCheck and CSpellCheck makes the whole component

COM: Example (Continued)

```
import "unknwn.idl";
[object, uuid(D66AB784-75C8-4f52-8EB2-C5BE9796ABEF)]
interface IUseCustomDictionary : IUnknown {
    HRESULT UseCustomDictionary([out, retval] vector <string>* dict);
}
[uuid(1C381680-CF29-46b1-8060-1237C36EA6C7)]
library CustomdictionaryLib {
    [uuid(C51815AF-CB06-4028-956C-C5F3E5781780)]
    coclass CCustomDictionary {
        interface IUseCustomDictionary;
    }
};
```

← UUID of Dictionary component

← IUseCustomDictionary interface specifies the method implemented by Dictionary component

← CCustomDictionary class implements the IUseCustomDictionary interface

Dictionary component interface -- IUseCustomDictionary

COM: Example (Continued)

In **design phase**, the spell checker system is assembled through method calls via interface pointers.

```
#include <string.h>
-----
CSpellCheckImpl :: CSpellCheckImpl() { }
CSpellCheckImpl :: ~CSpellCheckImpl() { }
STDMETHODIMP_(ULONG) CSpellCheckImpl :: AddRef(void) {
}
-----
STDMETHODIMP_(ULONG) CSpellCheckImpl :: Release(void) {
}
-----
STDMETHODIMP CSpellCheckImpl :: QueryInterface(...) {
}
-----
STDMETHODIMP CSpellCheckImpl :: CheckSpelling(...) {
-----
CCustomDictionary* pc = 0;
pc = new CCustomDictionaryImpl();
IUseCustomDictionary* pi = 0;
HRESULT hr;
hr = pc -> QueryInterface(IID_IUseCustomDictionary, (void**) &pi);
if(FAILED(hr)) return ERROR;
pi -> UseCustomDictionary(&m_dictionary);
-----
}
```

Checker component implementation

```
#include <fstream>
-----
CCustomDictionaryImpl :: CCustomDictionaryImpl() { }
CCustomDictionaryImpl :: ~CCustomDictionaryImpl() { }
STDMETHODIMP_(ULONG) CCustomDictionaryImpl :: AddRef(void) {
}
-----
STDMETHODIMP_(ULONG) CCustomDictionaryImpl :: Release(void) {
}
-----
STDMETHODIMP CCustomDictionaryImpl :: QueryInterface(...) {
}
-----
STDMETHODIMP CCustomDictionaryImpl :: UseCustomDictionary(...) {
    *p = dictionary;
    return NOERROR;
}
```

Dictionary component implementation

.NET Component Model: Components

In Microsoft .NET, a component is an assembly that is a binary unit supported by Common Language Runtime (CLR)

Metadata

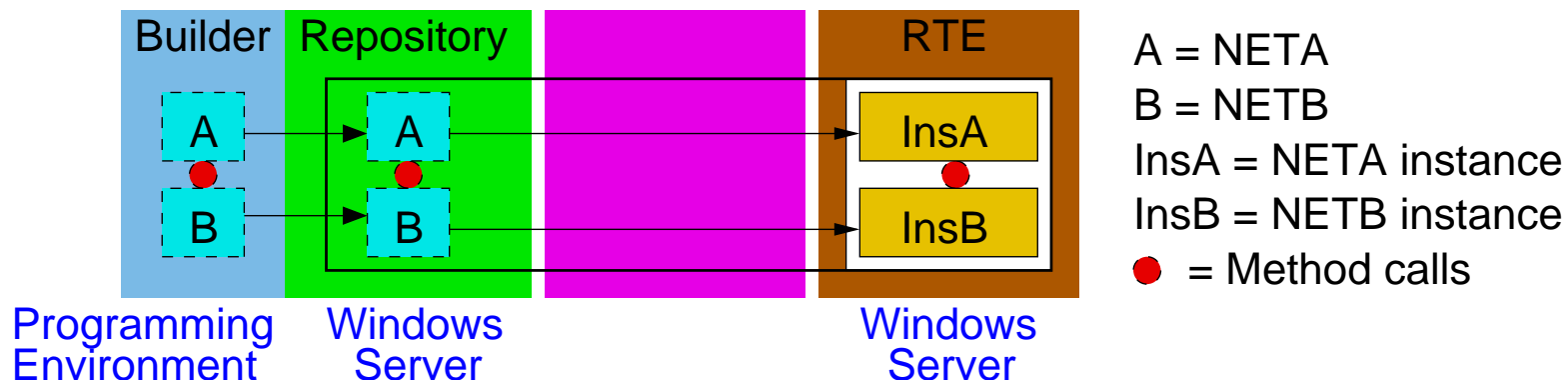
IL code

- A .NET component is made up of metadata and code in Intermediate Language (IL)
- The metadata contains the description of assembly, types and attributes
- The IL code can be executed in CLR

.NET: Builder, Repository

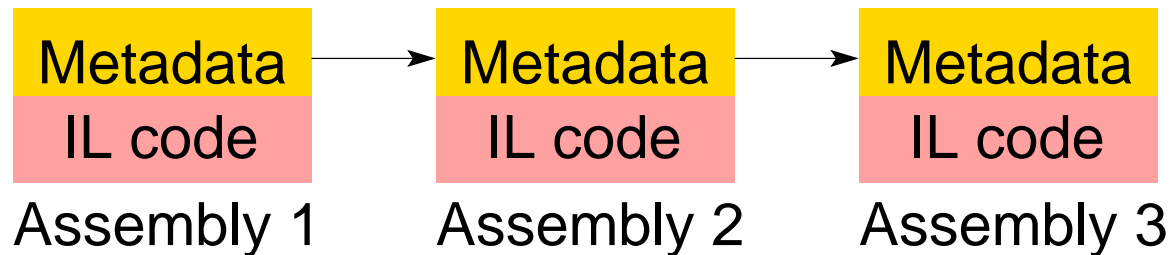
.NET components are constructed in a programming environment such as Microsoft Visual Studio .NET

- The programming environment is the builder
- The Windows server is the repository
- There is no assembler



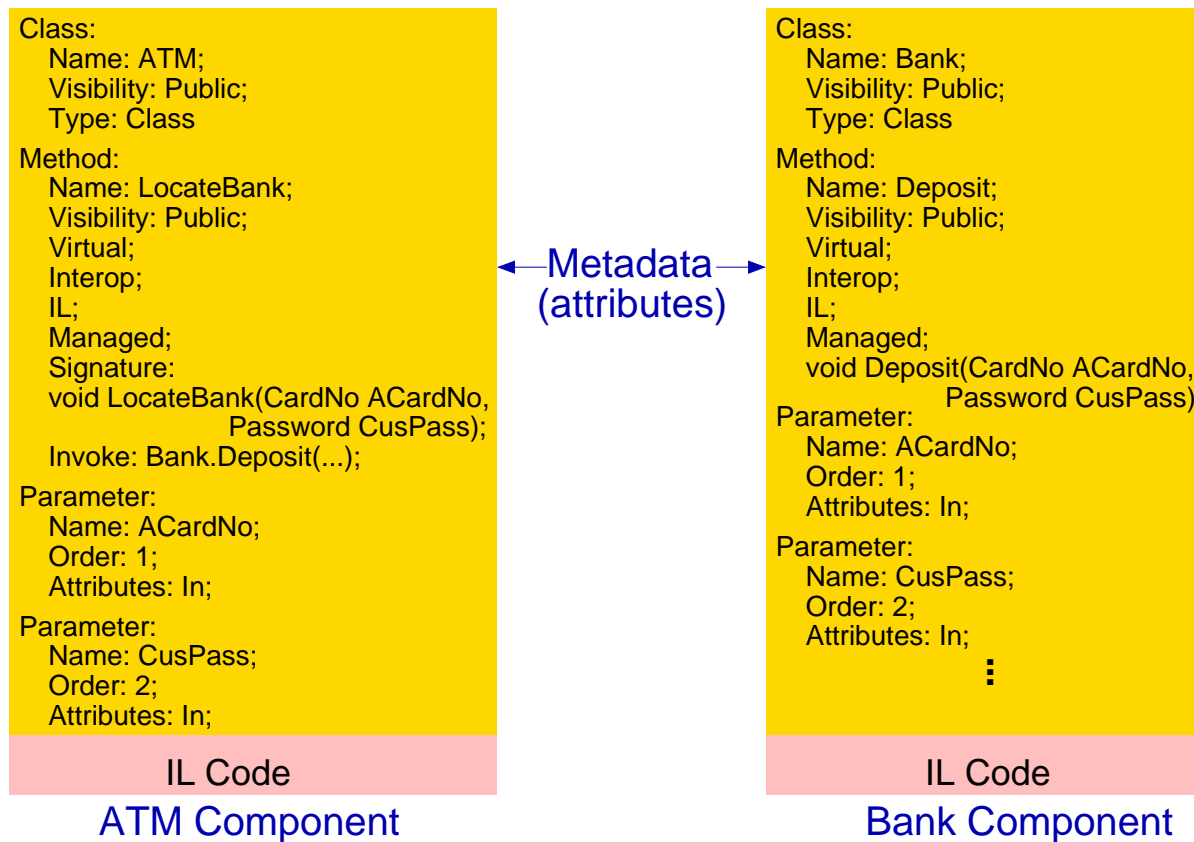
.NET: Composition

In the **design phase**, .NET components are **composed** by **method calls** through **references via metadata**



.NET: Example

Consider a banking system with an **ATM** component, which serves two instances **Bank1** and **Bank2** of a **Bank** component.



.NET: Example (Continued)

The banking system is assembled from the **ATM** component and two instances of **Bank** component.

```

Class:
  Name: ATM;
  Visibility: Public;
  Type: Class

Method:
  Name: LocateBank;
  Visibility: Public;
  Virtual;
  Interop;
  IL;
  Managed;
  Signature:
  void LocateBank(CardNo ACardNo,
                  Password CusPass);
  Invoke: Bank.Deposit(...);

Parameter:
  Name: ACardNo;
  Order: 1;
  Attributes: In;

Parameter:
  Name: CusPass;
  Order: 2;
  Attributes: In;
  
```

IL Code
ATM Component

```

Class:
  Name: Bank;
  Visibility: Public;
  Type: Class

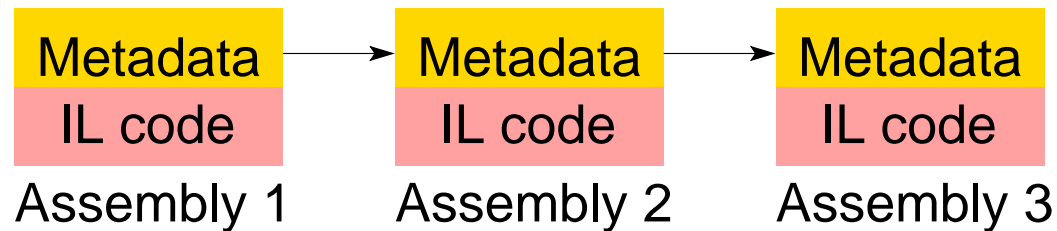
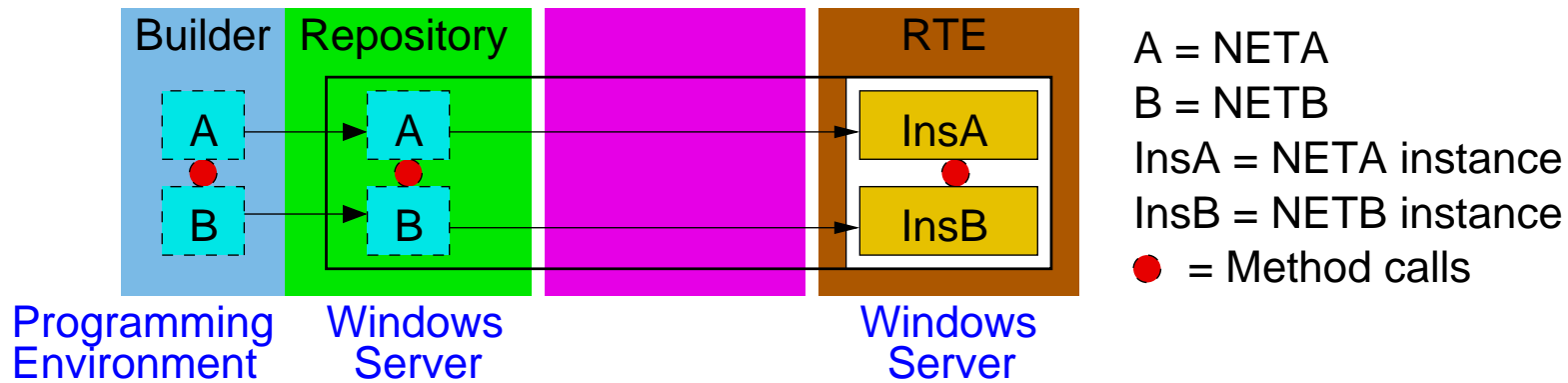
Method:
  Name: Deposit;
  Visibility: Public;
  Virtual;
  Interop;
  IL;
  Managed;
  void Deposit(CardNo ACardNo,
              Password CusPass);

Parameter:
  Name: ACardNo;
  Order: 1;
  Attributes: In;

Parameter:
  Name: CusPass;
  Order: 2;
  Attributes: In;
  :
  
```

IL Code
Bank Component

.NET: Summary



CORBA Component Model (CCM): Components

In CCM, a component is a CORBA meta-type hosted by a CCM container on a CCM platform such as OpenCCM.



- A CORBA meta-type is an extension and specialisation of a CORBA Object
 - Component interfaces are made up of ports
- CCM supports 4 kinds of ports: Facets (provided services), Receptacles (required services), Event Sources and Sinks.

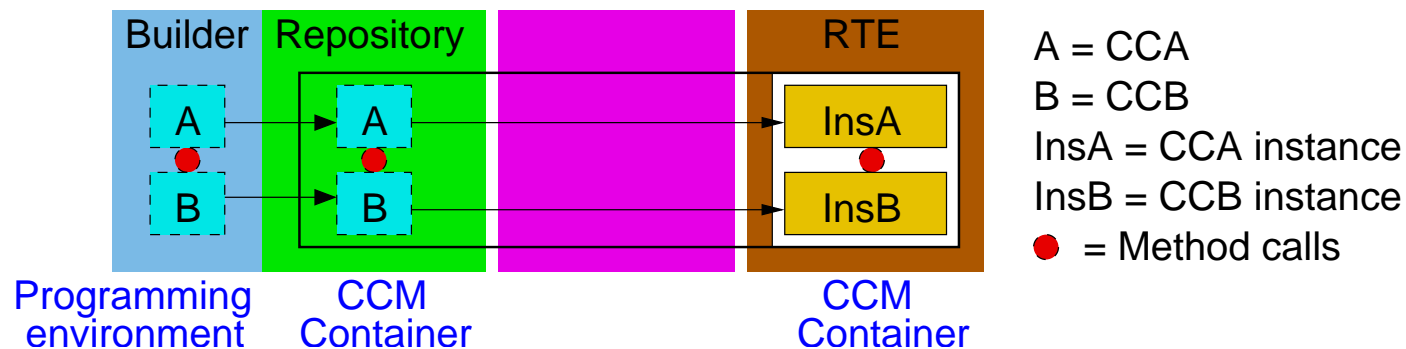
CORBA Component Model (CCM): Components (Continued)

- **Component types** are specific, named collections of features that can be described in **OMG IDL 3**
- CORBA components have **homes** that are component factories to manage a component instance life cycle

CCM: Builder, Repository

CORBA components are constructed in a programming environment such as Open Production Tool Chain and deposited into a CCM container hosted and managed by a CCM platform such as OpenCCM.

- The programming environment is the builder
- The CCM container is the repository
- There is no assembler

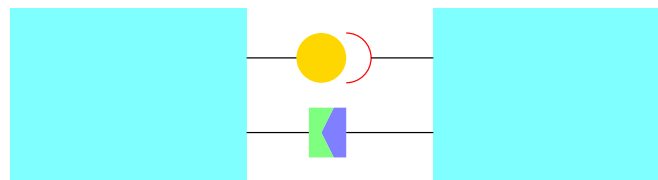


CCM: Composition

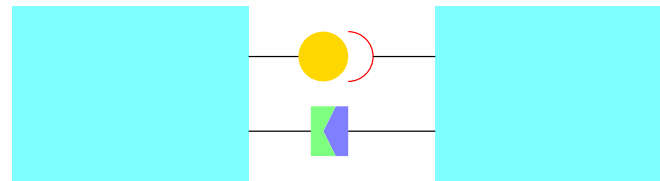
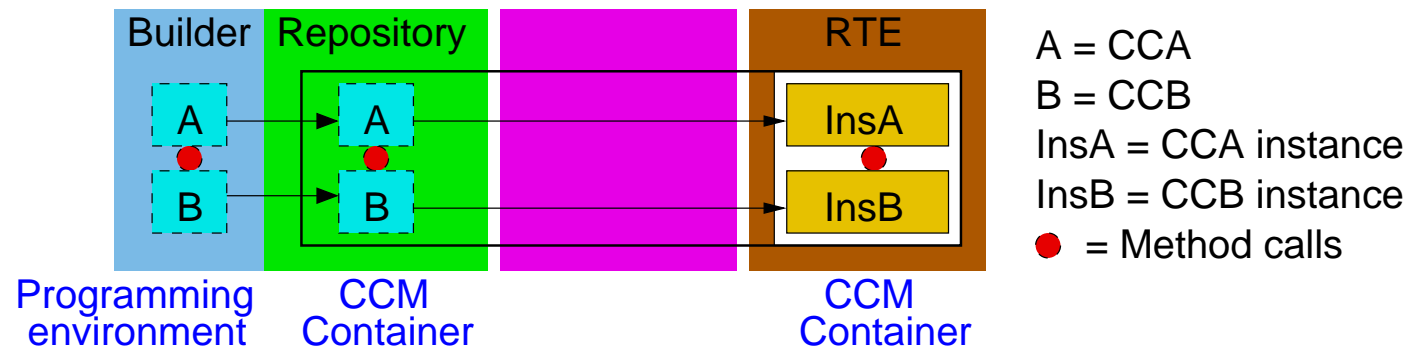
In **design phase**, CORBA components are **assembled** by **method** and **event delegations**

in such a way that

- **facets** match **receptacles**
- **event sources** match **event sinks**



CCM: Summary



CCM: Example

Consider a simple bank system implemented by ATM, BankConsortium, Bank1 and Bank2 components (in OMG IDL 3):

```
interface Bank {
  string getBankID(string cardno);
  void deposit(string cardno);
  void withdraw(string cardno);
  void checkBalance(string cardno);
}
```

```
enum BankState {
  IsCustomer, NotCustomer
};

eventtype AccountInfo {
  public string cardno;
  public BankState customerinfo;
};
```

```
component ATM {
  attribute string atmid;
  uses Bank getBankID;
  consumes AccountInfo customer;
};

home ATMhome manages ATM {
  factory new(in string atmid);
};
```

receptacle —> uses Bank getBankID;

event sink —> consumes AccountInfo customer;

manages instances —> home ATMhome manages ATM {

CCM: Example (Continued)

```

component Bank {
  attribute string bankid;
  provides Bank deposit;
  provides Bank withdraw;
  provides Bank checkBalance;
};
home Bankhome manages Bank {
  factory new(in string bankid);
};
  
```

facet →

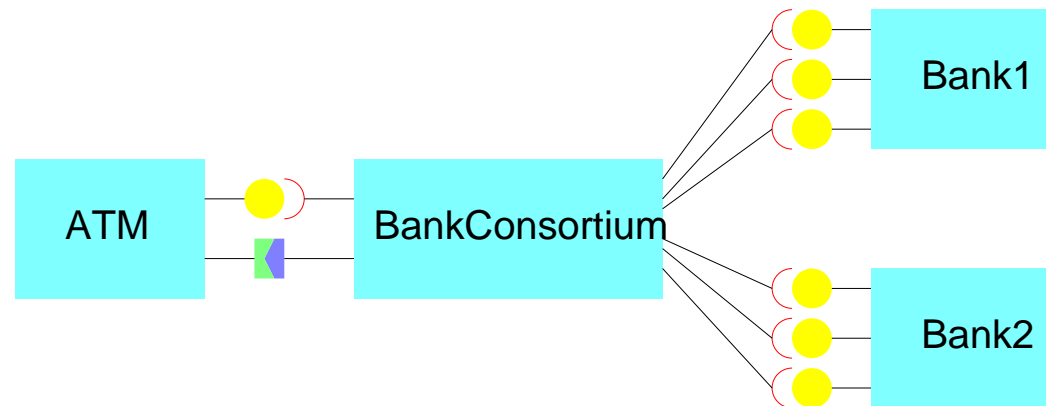
```

component BankConsortium {
  attribute string bankconsortiumid;
  provides Bank getBankID;
  uses Bank deposit;
  uses Bank withdraw;
  provides Bank checkBalance;
  publishes AccountInfo customer;
};
home BankConhome manages BankConsortium {
  factory new(in string bankconsortiumid);
};
  
```

event source →

CCM: Example (Continued)

The bank system is assembled from the ATM, BankConsortium, Bank1 and Bank2 components.



The composition of CORBA components is specified in a Component Assembly Descriptor (an XML file)

CCM: Example (Continued)

```

<?xml version = "1.0"?>
<!DOCTYPE component assembly BANKSYSTEM "componentassembly.dtd">
<component assembly id = "banksys">
  <description> bank assembly descriptor</description>
  <componentfiles>
    <componnetfile id = "ATM component">
      <filearchive name = "ATM.csd">
        </componentfile>
      <componnetfile id = "BankConsortium component">
        <filearchive name = "BankConsortium.csd">
          </componentfile>
        <componnetfile id = "Bank component">
          <filearchive name = "Bank.csd">
            </componentfile>
          </componentfile>
        </componentfiles>
      <partitioning>
        <homereplacement id = "ATMHome">
          <componentfileref idref = "ATM Component"/>
          <componentinstantiation id = "atm">
            <registerwithnaming name = "ATMHome"/>
          </homereplacement>
          <homereplacement id = "BankConsortiumHome">
            <componentfileref idref = "BankConsortium Component"/>
            <componentinstantiation id = "bankconsortium">
              <registerwithnaming name = "BankConsortiumHome"/>
            </homereplacement>
            <homereplacement id = "BankHome">
              <componentfileref idref = "Bank Component"/>
              <componentinstantiation id = "bank1">
                <componentinstantiation id = "bank2">
                  <registerwithnaming name = "BankHome"/>
                </componentinstantiation>
              </homereplacement>
            </partitioning>
          <connections>
            ⋮
          </connections>
        </component assembly>

```

CCM: Example (Continued)

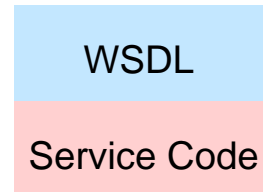
```

<connections>
  <connectinterface>
    <usesport>
      <usesidentifier>getBankID</usesidentifier>
      <componentinstantiationref idref = "atm"/>
      <usesidentifier>deposit</usesidentifier>
      <usesidentifier>withdraw</usesidentifier>
      <usesidentifier>checkBalance</usesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </usesport>
    <providesport>
      <providesidentifier>getBankID</providesidentifier>
      <componentinstantiationref idref = "bankcon"/>
      <providesidentifier>deposit</providesidentifier>
      <providesidentifier>withdraw</providesidentifier>
      <providesidentifier>checkBalance</providesidentifier>
      <componentinstantiationref idref = "bank"/>
    </providesport>
  </connectinterface>
  <connectevent>
    <publishesport>
      <publishesidentifier>customer</publishesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </publishesport>
    <consumesport>
      <consumesidentifier>customer</consumesidentifier>
      <componentinstantiationref idref = "atm"/>
    </consumesport>
  </connectevent>
</connections>

```

Web Services: Components

In **Web Services**, a component is a **service** that is a **resource** that represents a capability of performing some tasks

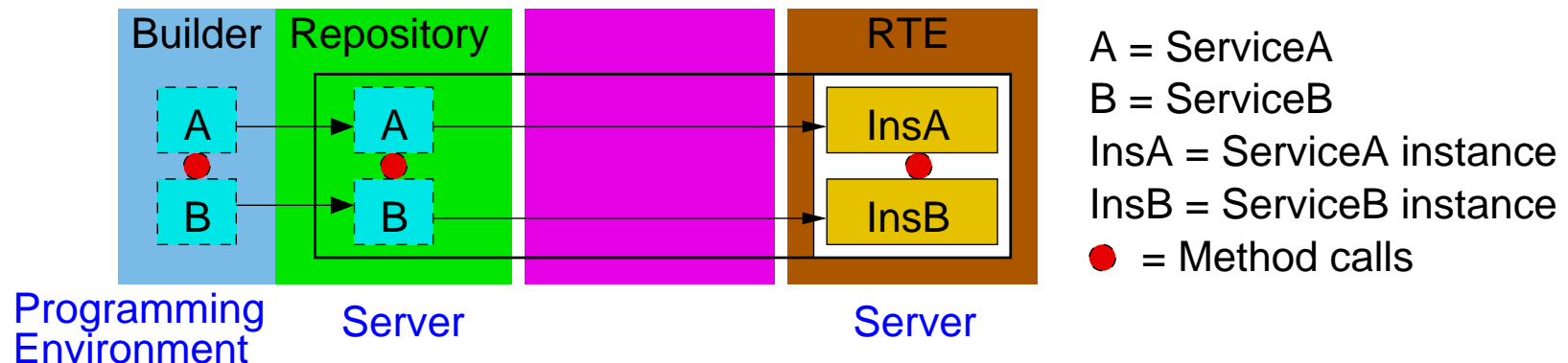


- A Web Service contains an **interface** in **WSDL** (**Web Service Description Language**) and a **binary implementation**
- The **WSDL interface** describes the **functionalities** that the web service can provide and are published in **UDDI** (**Universal Description Discovery and Integration**)
- The **service code** is the implementation that performs the task

Web Services: Builder, Repository

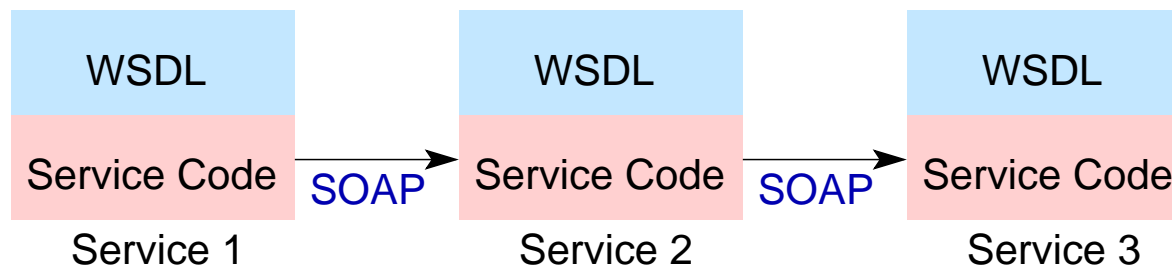
Web services are constructed in a **programming environment**, e.g. Eclipse for Java

- The **programming environment** is the **builder**
- The **server** is the **repository**
- There is **no assembler**



Web Services: Composition

In the **design phase**, Web services are **composed** by **method calls** through **SOAP messages**



Web Services: Example

Consider a banking system with an **ATM** service and two **Bank** services

```
public class ATM .....{
  CardNo ACardNo = new CardNo();
  Password CusPass = new Password();
  .....
  public String LocateBank(CardNo ACardNo,
                           Password CusPass) {
    if(B1) then
      .....
      getServiceLocation(....,
        http://localhost:8080/axis/services/Bank1, ...);
    elseif(B2) then
      .....
      getServiceLocation(....,
        http://localhost:8080/axis/services/Bank2, ...);
    .....
  }
}
```

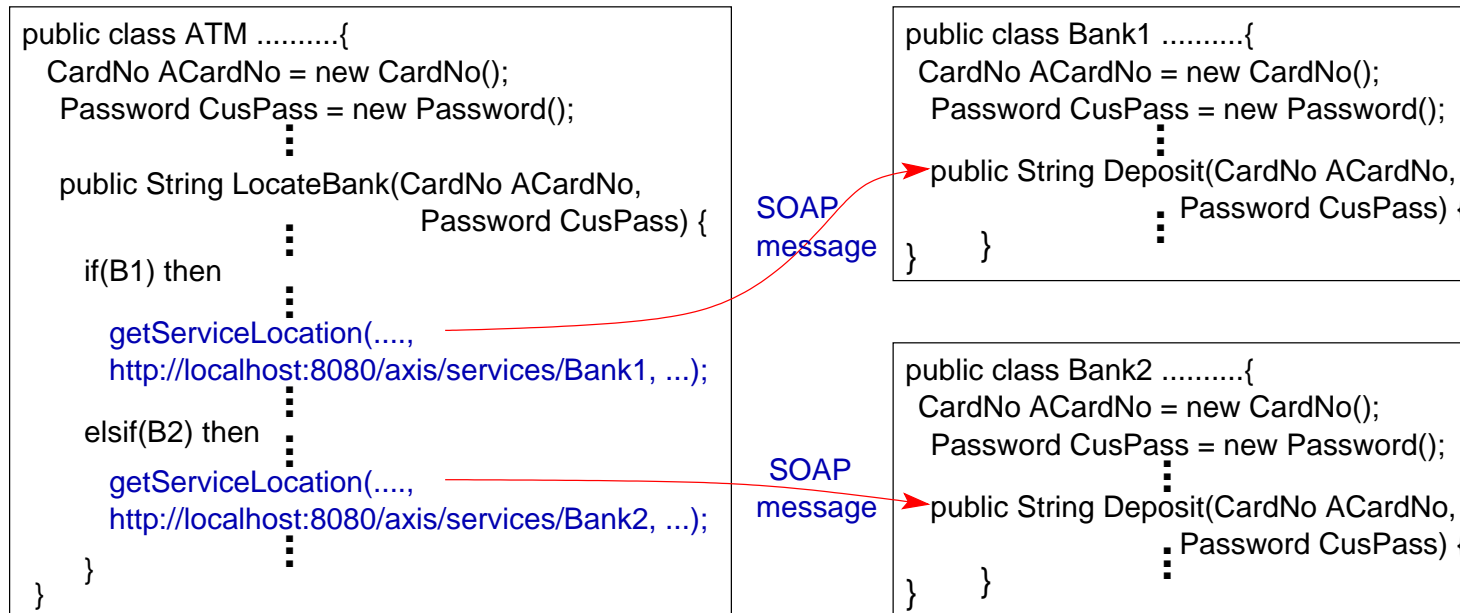
```
public class Bank1 .....{
  CardNo ACardNo = new CardNo();
  Password CusPass = new Password();
  .....
  public String Deposit(CardNo ACardNo,
                        Password CusPass) {
    .....
  }
}
```

```
public class Bank2 .....{
  CardNo ACardNo = new CardNo();
  Password CusPass = new Password();
  .....
  public String Deposit(CardNo ACardNo,
                        Password CusPass) {
    .....
  }
}
```

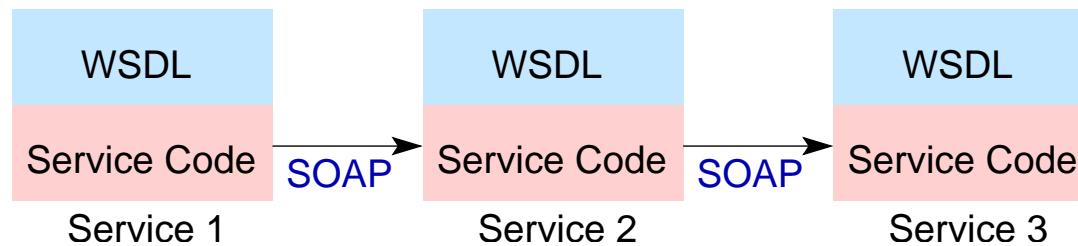
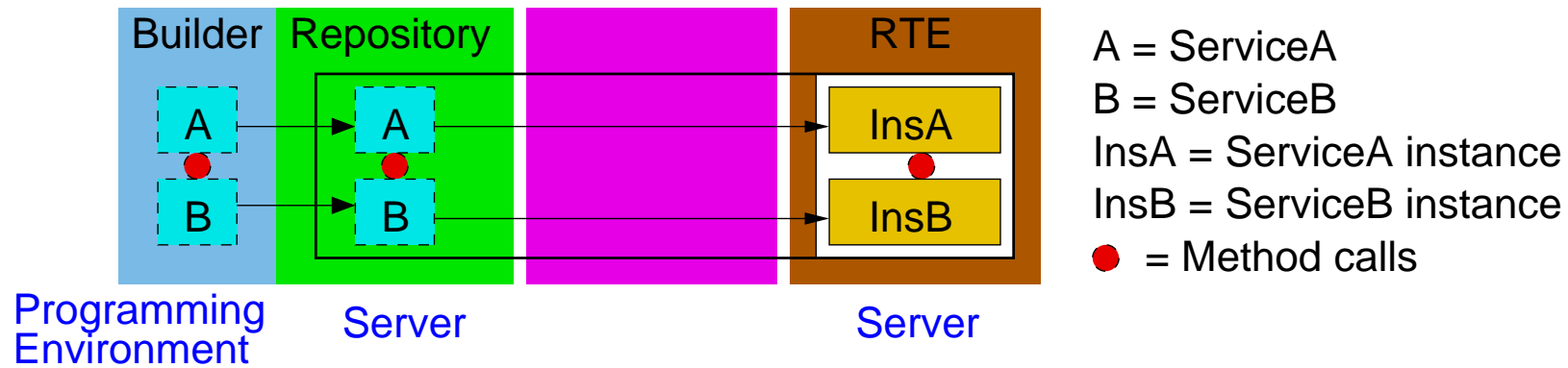
The ATM service has **SOAP messages** to the Bank services hard-coded into its service code.

Web Services: Example (Continued)

In the **design phase**, the banking system is assembled from the ATM service and two Bank services



Web Services: Summary

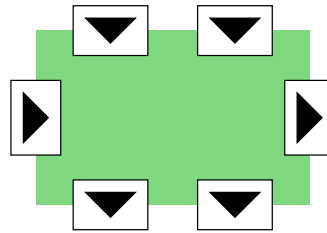


Category 3

Koala, SOFA, KobrA

Koala: Components

In **Koala*** a component is a **unit of design** which has a specification and an implementation.



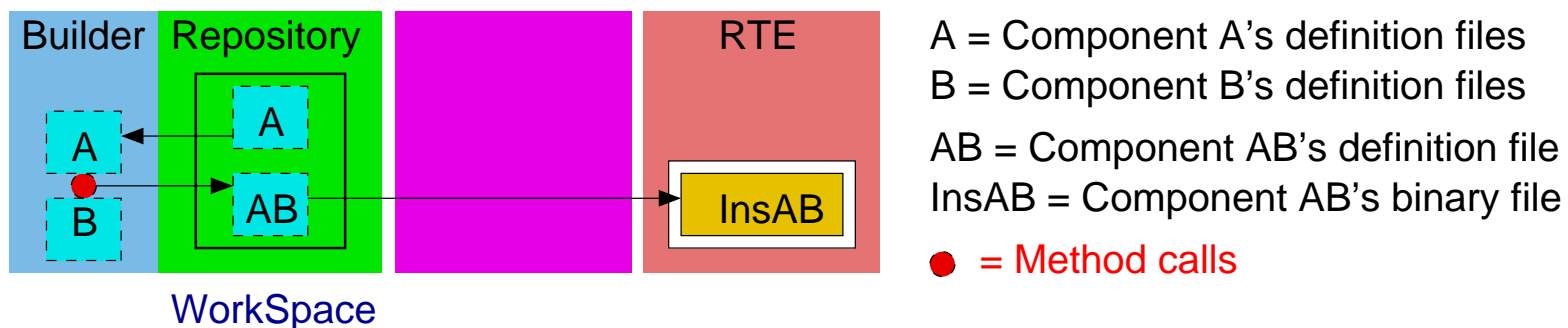
- **Semantically**, components are **units of computation and control** (and data) connected together in an **architecture**.
- **Syntactically**, components are defined in an **ADL-like language** (Koala).

Components are **definition files only** (no implementation).

*C[K]omponent Organizer And Linking Assistant

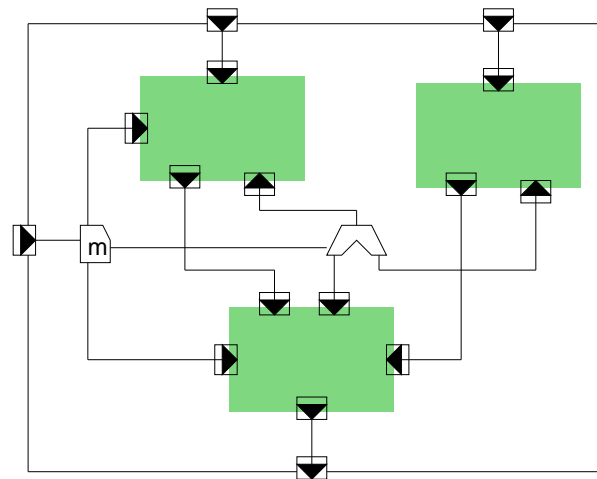
Koala: Builder, Repository

- The **builder** is a Koala programming environment
- **KoalaModel Workspace** (a file system) provides the **repository** (Composition of definition files)
- There is **no assembler**



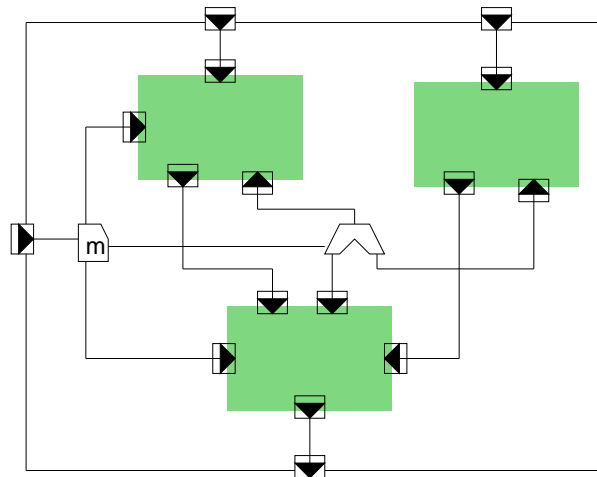
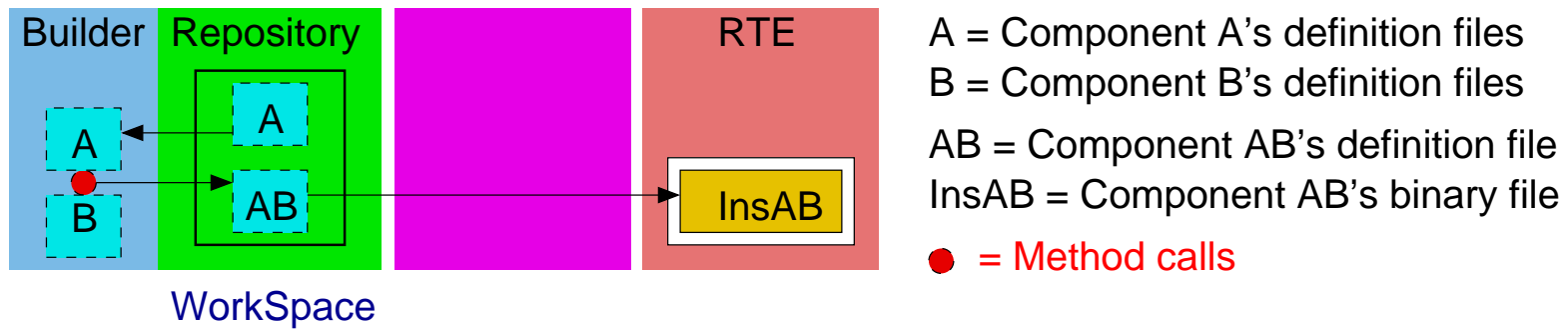
Koala: Composition

In **design phase**, Koala components are **composed** by **method calls** through **connectors**.



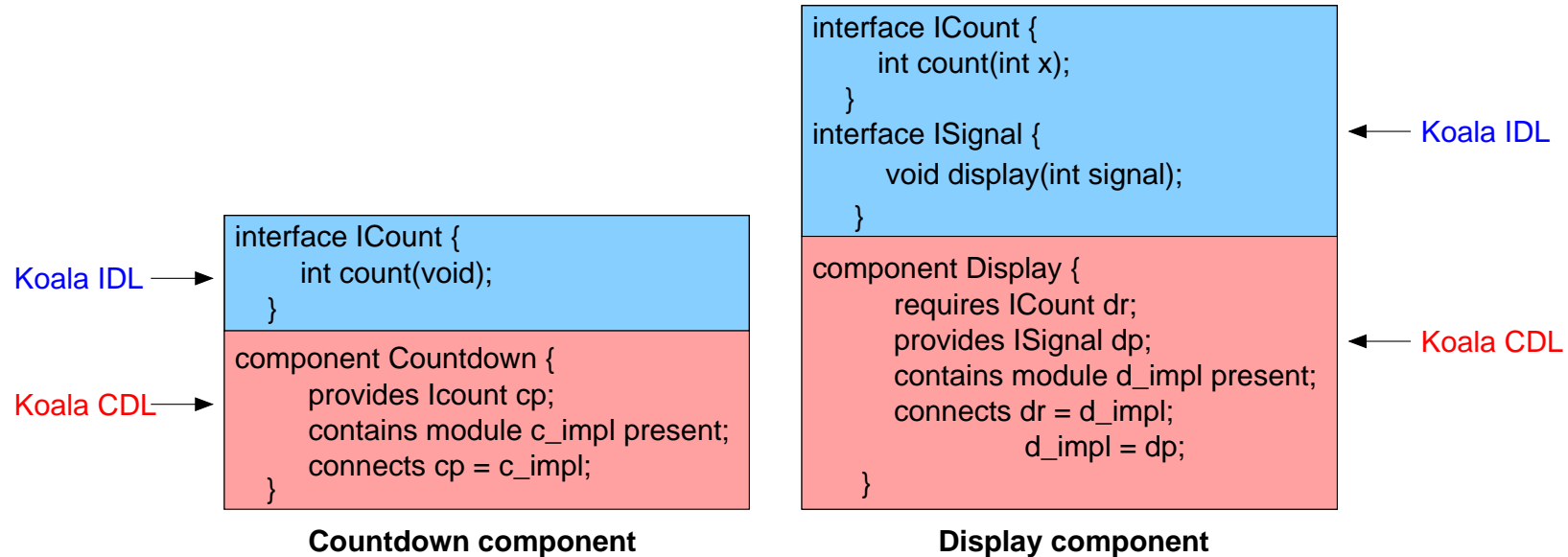
At **run-time**, Koala components are compiled into a programming language and executed in the run-time environment of that language.

Koala: Summary



Koala: Example

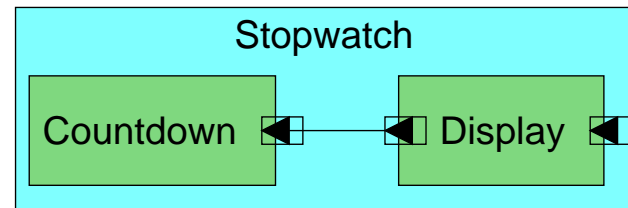
Consider a **Stopwatch** device that comprises a **Countdown** component and a **Display** component.



- The **interfaces** are specified in **Koala IDL**
- The **component definitions** are in **Koala CDL**

Koala: Example (Continued)

In **design phase**, the **Stopwatch** device is constructed by **composing** a **Countdown** component (new) with a **Display** component (from the repository)



The **definition file** for **Stopwatch** is assembled from **Countdown** and **Display**

```
component Stopwatch {
  contains component Countdown c;
  contains component Display d;
  connects d.dr = c.cp;
}
```

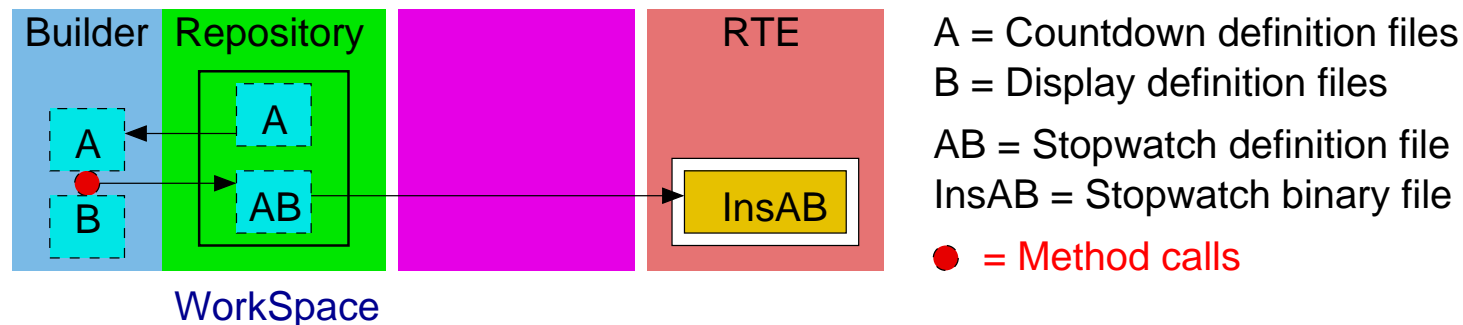
Stopwatch configuration

Koala: Example (Continued)

In **deployment phase**, the **definition files** of **Stopwatch**, **Countdown** and **Display** are compiled by the Koala compiler to **C header files**.

Then **the programmer** has to

- **write C files** (to implement the components)
- **compile** these with the header files to **binary C code** for **Stopwatch**.



SOFA: Components

In **SOFA*** a component is a **unit of design** which has a specification and an implementation, and is specified by its **frame** and **architecture**.



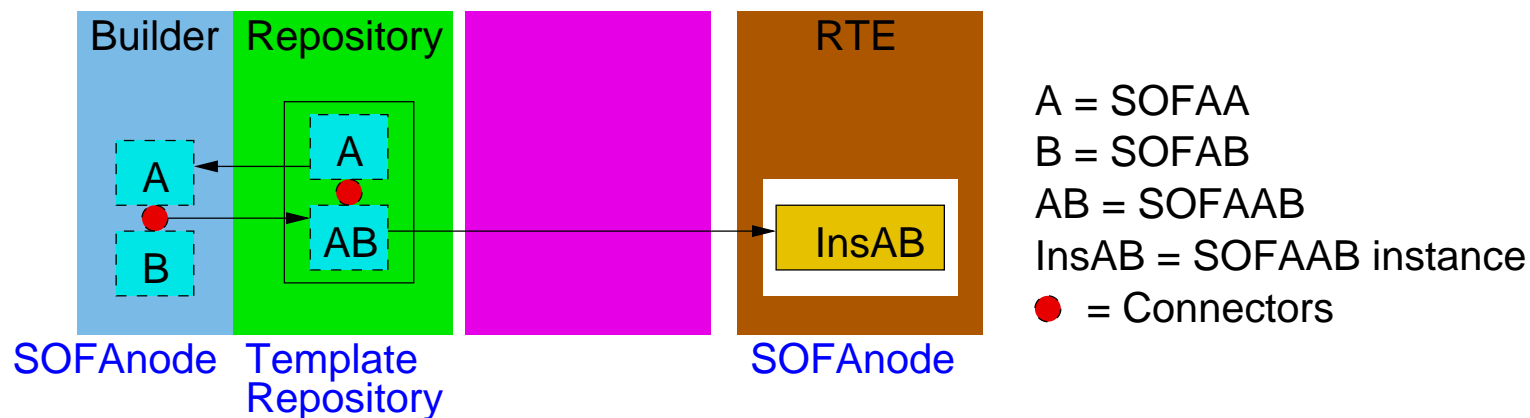
- The **frame** defines provides and requires **interfaces**, and properties of the component
- The **architecture** describes the **structure** of the component
- SOFA components are defined in an ADL-like language — **SOFA Component Definition Language (CDL)**.

*SOFTware Appliances

SOFA: Builder, Repository

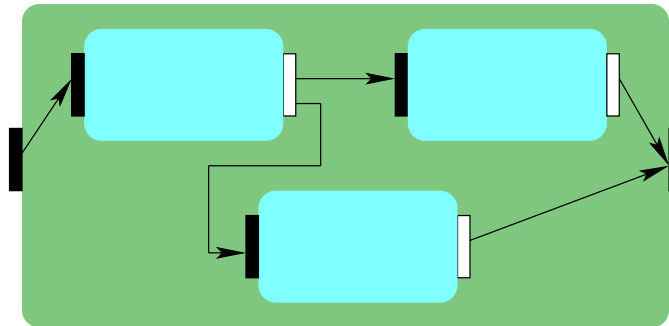
SOFA components are constructed in **SOFAnode** and deposited into the **Template Repository**.

- **SOFAnode** is the **builder**
- The **Template Repository** is the **repository**
- There is **no assembler**

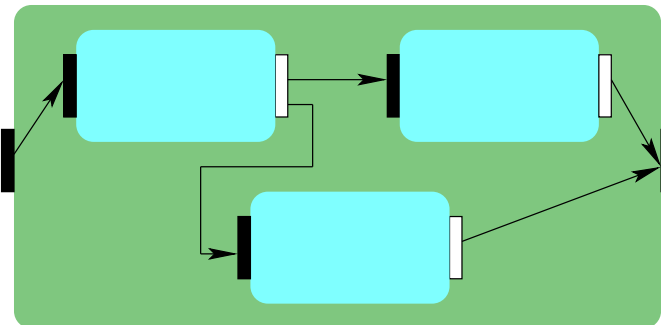
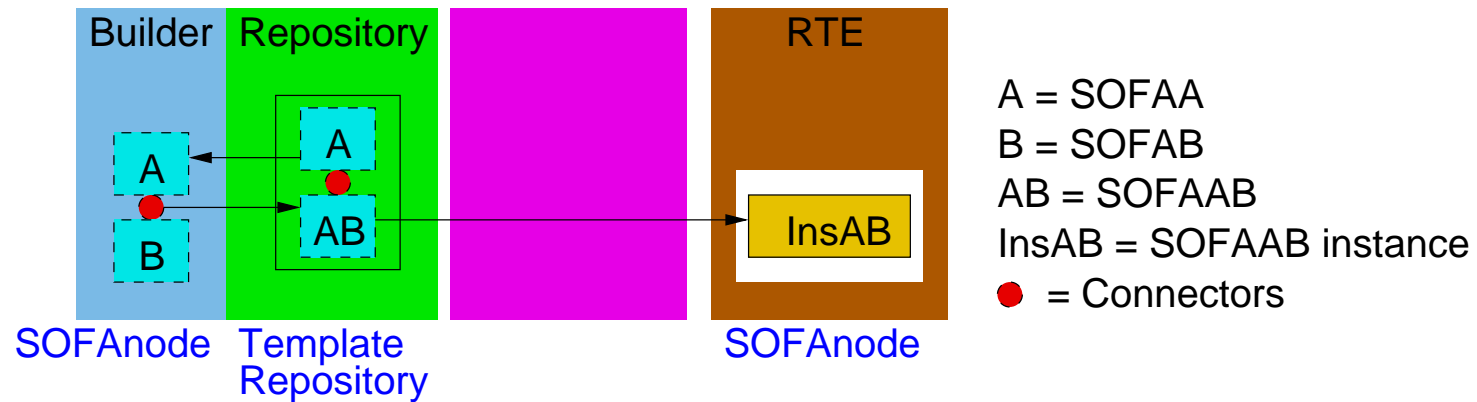


SOFA: Composition

In **design phase**, SOFA components are **composed** by **method calls** through **connectors**.



SOFA: Summary



SOFA: Example

Consider a **Stopwatch device** that comprises a **Countdown** component and a **Display** component.

```
interface CountInterface {
    int count(void);
};
frame Countdown {
    provides:
        CountInterface Count;
};
architecture CUNI Countdown
    version "1.0" primitive;
```

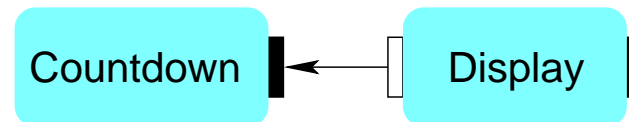
Countdown component

```
interface CountInterface {
    int count(int x);
};
interface SignalInterface {
    void display(int signal);
};
frame Display {
    requires:
        CountInterface Count;
    provides:
        SignalInterface Signal;
};
architecture CUNI Display
    version "1.0" primitive;
```

Display component

The components are specified in **SOFA CDL**.

In **design phase**, the **Stopwatch** device is implemented by constructing a new **Countdown** component and composing it with a **Display** component from the repository.



Stopwatch Architecture

The **definition file** for Stopwatch device is assembled from the Countdown and Display components.

```
system CUNI Stopwatch version "1.0" {
  inst Countdown aCountdown;
  inst Display aDisplay;
  bind aDisplay.Count to aCountdown.count using CSProcCall;
};
```

Stopwatch device

KobrA: Components

In **KobrA*** a component is a **UML component**. Every KobrA component has a **specification** and an **implementation**

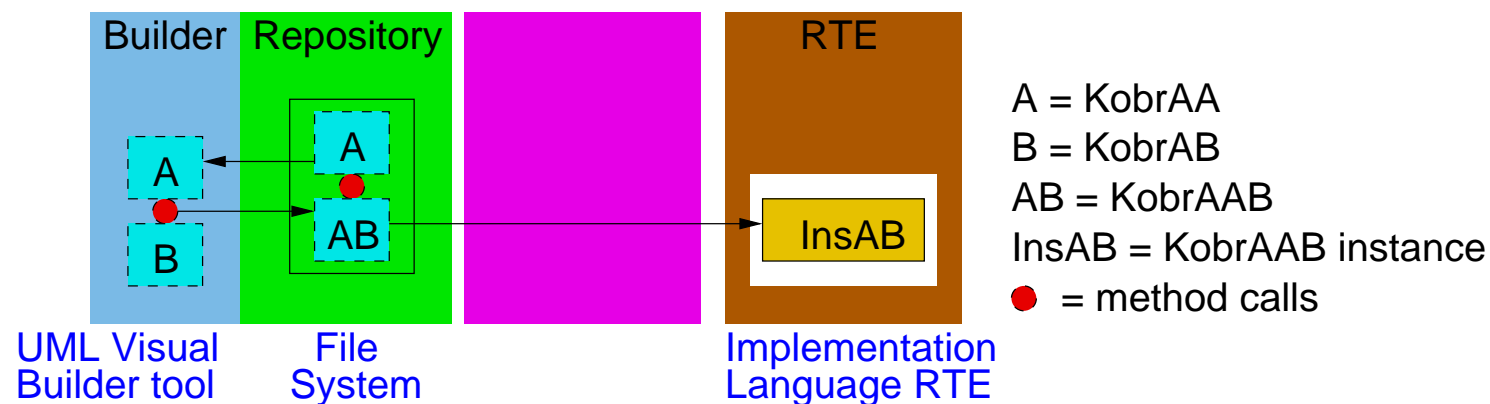
- The **specification** describes what a component does and thus it is the **interface** of the component
- The **implementation** describes how it does it

*Komponenten-basierte Anwendungsentwicklung (component-based application development)

KobrA: Builder, Repository

KobrA components can be constructed in a **visual builder tool** such as Visual UML and deposited into a **file system**.

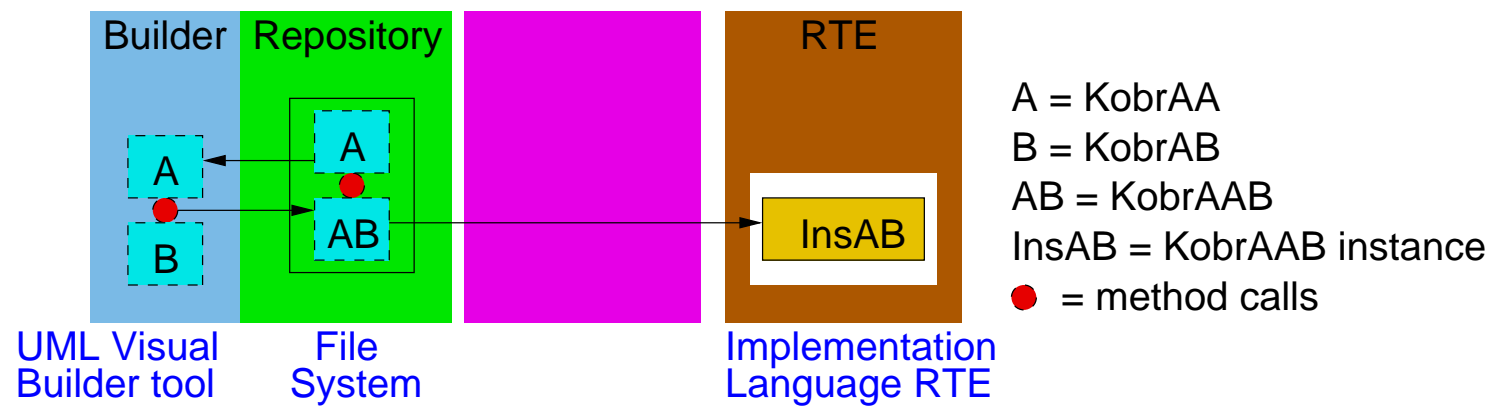
- The **visual builder tool** is the **builder**
- The **file system** is the **repository**
- There is **no assembler**



KobrA: Composition

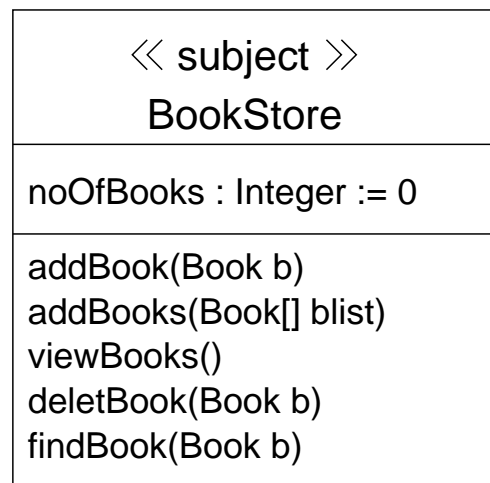
In the design phase, KobrA components are **composed** by **direct method calls**.

KobrA: Summary



KobrA: Example

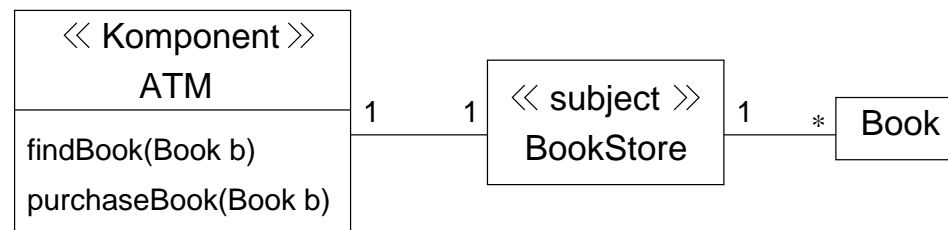
Consider a **book store** that maintains a database of its book stock and sells its books by an **Automatic Teller Machine (ATM)**.



The specification of the BookStore component is a **UML class diagram** that specifies what the BookStore component does.

KobrA: Example (Continued)

In **design phase**, the **book store system** is implemented by constructing a new **ATM** component and **composing** it with **BookStore** and **Book** components from the **repository**.



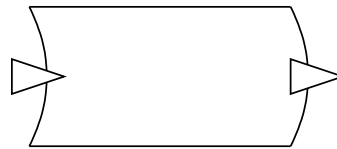
The book store system is **assembled** from the ATM, BookStore and Book components by **direct method calls**.

Category 4

ADLs, UML 2.0, PECOS, Pin, Fractal

Architecture Description Languages (ADLs): Components

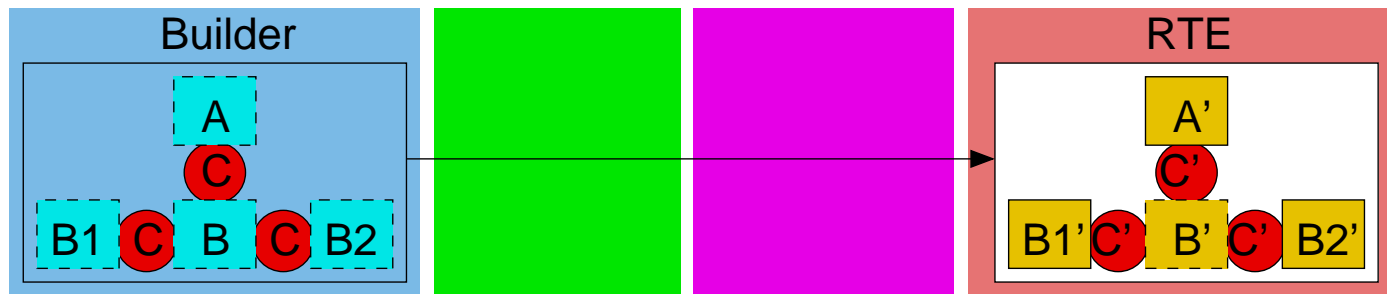
In **ADLs**, a component is an **architectural unit** that represents a primary **computational element** and **data store** of a system.



- **Interfaces** are defined by a set of **ports**
- Each **port** identifies a **point of interaction** between the **component** and its **environment**
- A component may have **multiple interfaces** by using different types of ports
- **Connectors** connect components via their **ports**

ADLs: Builder

- The **builder** is the **ADL tool** if any (Composition of architectural units by connectors)
- There is **no repository**
- There is **no assembler**



A = Component A
 B = Component B
 B1 = Component B1
 B2 = Component B2
 C = Connector

A' = Implementation of A
 B' = Implementation of B
 B1' = Implementation of B1
 B2' = Implementation of B2
 C' = Connector implementation

ADLs: Composition

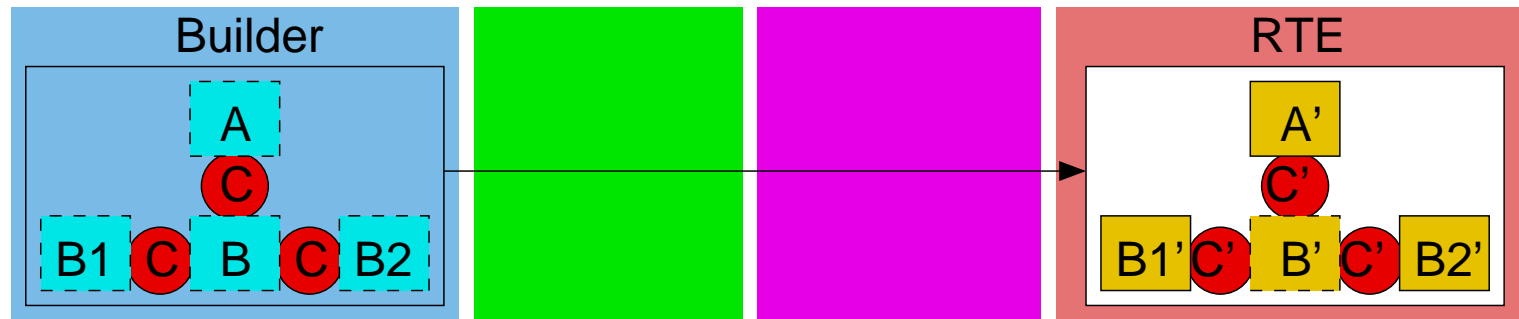
In **design phase**, components are

- **identified** and **defined**
- **assembled** by **connectors** into a **system design**

The **design** has to be implemented (somehow) in a chosen programming language.

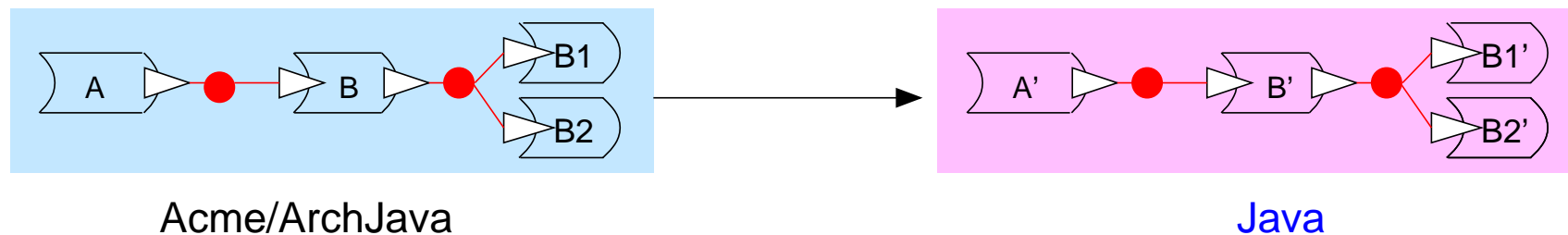
At **run-time**, the implemented system is executed in the run-time environment of that programming language.

ADLs: Summary



A = Component A
 B = Component B
 B1 = Component B1
 B2 = Component B2
 C = Connector

A' = Implementation of A
 B' = Implementation of B
 B1' = Implementation of B1
 B2' = Implementation of B2
 C' = Connector implementation



Acme/ArchJava

Java

ADLs: Example

Consider a simple bank system consisting of an **ATM** component, a **BankConsortium** component, and 2 **Bank** components **Bank1** and **Bank2**.

```
Component ATM = {
    Port send;
}
```

ATM component

```
Component BankConsortium = {
    Port receive;
    Port send;
}
```

BankConsortium component

```
Component Bank1 = {
    Port receive;
    Property bankid : String =
        "Bank 1";
}
```

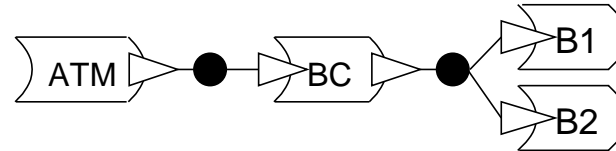
Bank1 component

```
Component Bank2 = {
    Port receive;
    Property bankid : String =
        "Bank 2";
}
```

Bank2 component

ADLs: Example (Continued)

In **design phase**, the **architecture** for the whole system is **designed**



using the above **components** and the following **connectors**:

```

Connector ATMtoBankCon = {
    Role request;
    Role produce;
}
  
```

```

Connector BankContoB1 = {
    Role request;
    Role produce;
}
  
```

```

Connector BankContoB2 = {
    Role request;
    Role produce;
}
  
```


ADLs: Example (Continued)

```

System BankSys = {
  Component ATM = {
    Port send;
  };

  Component Bank1 = {
    Port receive;
    Property bankid : String = "Bank1";
  };

  Connector ATMtoBankCon = {
    Role request;
    Role produce;
  };

  Connector BankContoB1 = {
    Role request;
    Role produce;
  };

  Attachments {
    ATM.send to ATMtoBankCon.request;
    ATMtoBankCon.produce to BankConsortium.receive;
    BankConsortium.send to BankContoB1.request;
    BankContoB1.produce to Bank1.receive;
    BankConsortium.send to BankContoB2.request;
    BankContoB2.produce to Bank2.receive;
  }
}

Component BankConsortium = {
  Port receive;
  Port send;
};

Component Bank2 = {
  Port receive;
  Property bankid : String = "Bank2";
};

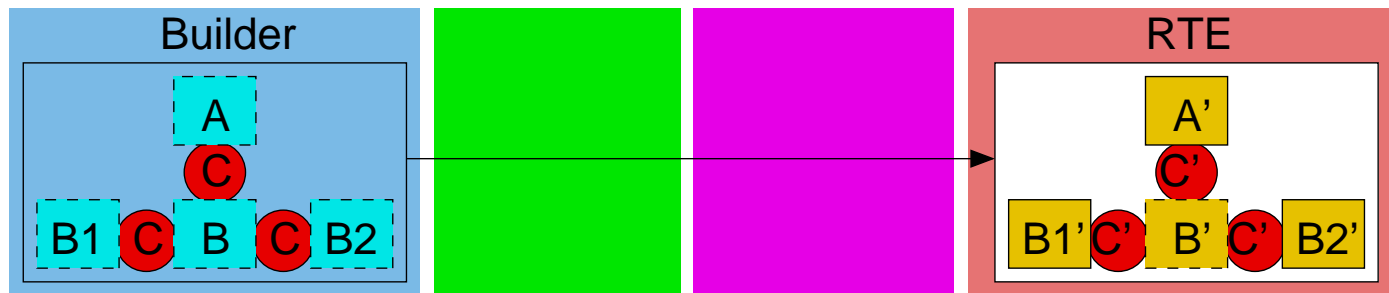
Connector BankContoB2 = {
  Role request;
  Role produce;
};

```

ADLs: Example (Continued)

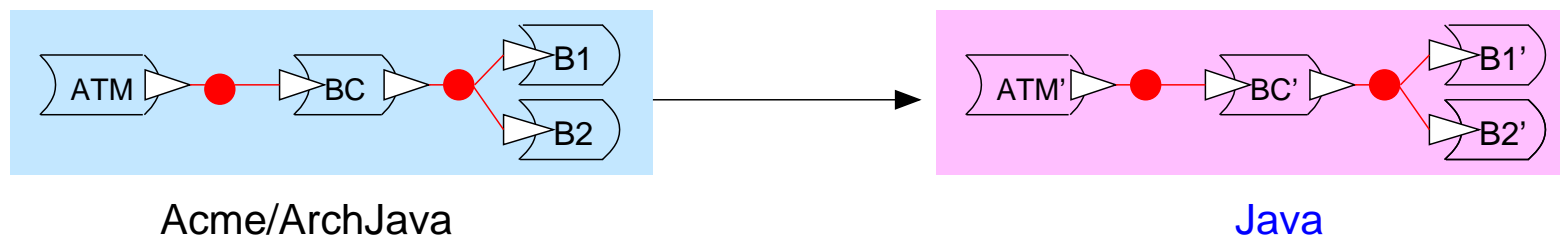
The **design** has to be implemented in some programming language.

At **run-time**, the implementation is executed in the run-time environment of that language.



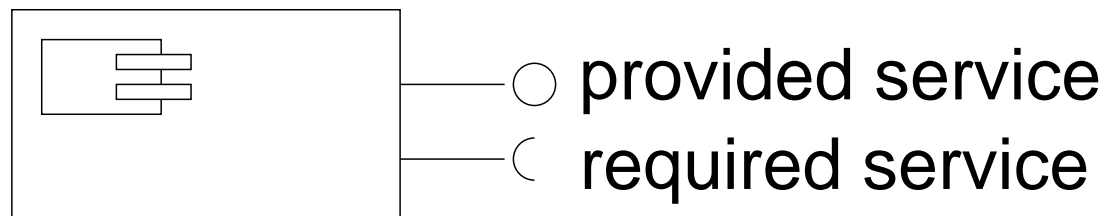
A = ATM component
 B = BankConsortium component
 B1 = Bank component 1
 B2 = Bank component 2
 C = Connector

A' = Implementation of A
 B' = Implementation of B
 B1' = Implementation of B1
 B2' = Implementation of B2
 C' = Connector implementation



UML 2.0 Component Model: Components

In **UML 2.0**, a component is a **modular unit** of a system with well-defined interfaces that is replaceable within its environment.



- A **component** defines its behaviour by **required** and **provided interfaces** (ports);
- **Services** of components are encapsulated through their required and provided interfaces.

Components are represented in UML 2.0.

UML 2.0: Connectors

UML components are **composed** by UML **connectors**:

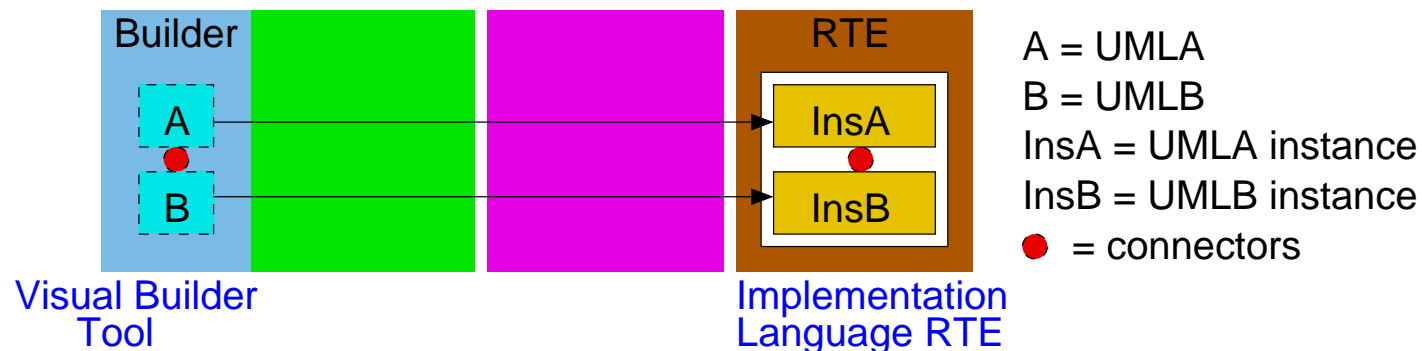
- **delegation** connectors
- **assembly** connectors

Composites are assembled by **assembly connectors**

Systems are assembled by **delegation** and **assembly connectors**

UML 2.0: Builder

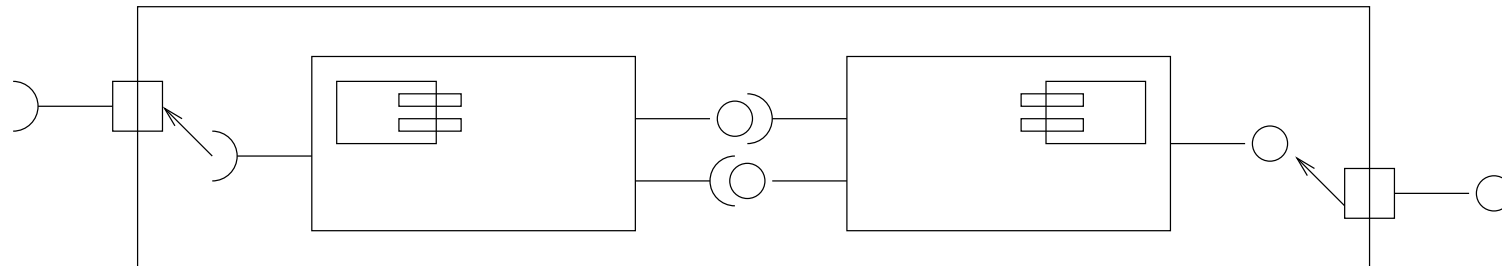
UML components can be **constructed** in a visual builder tool such as Visual UML.



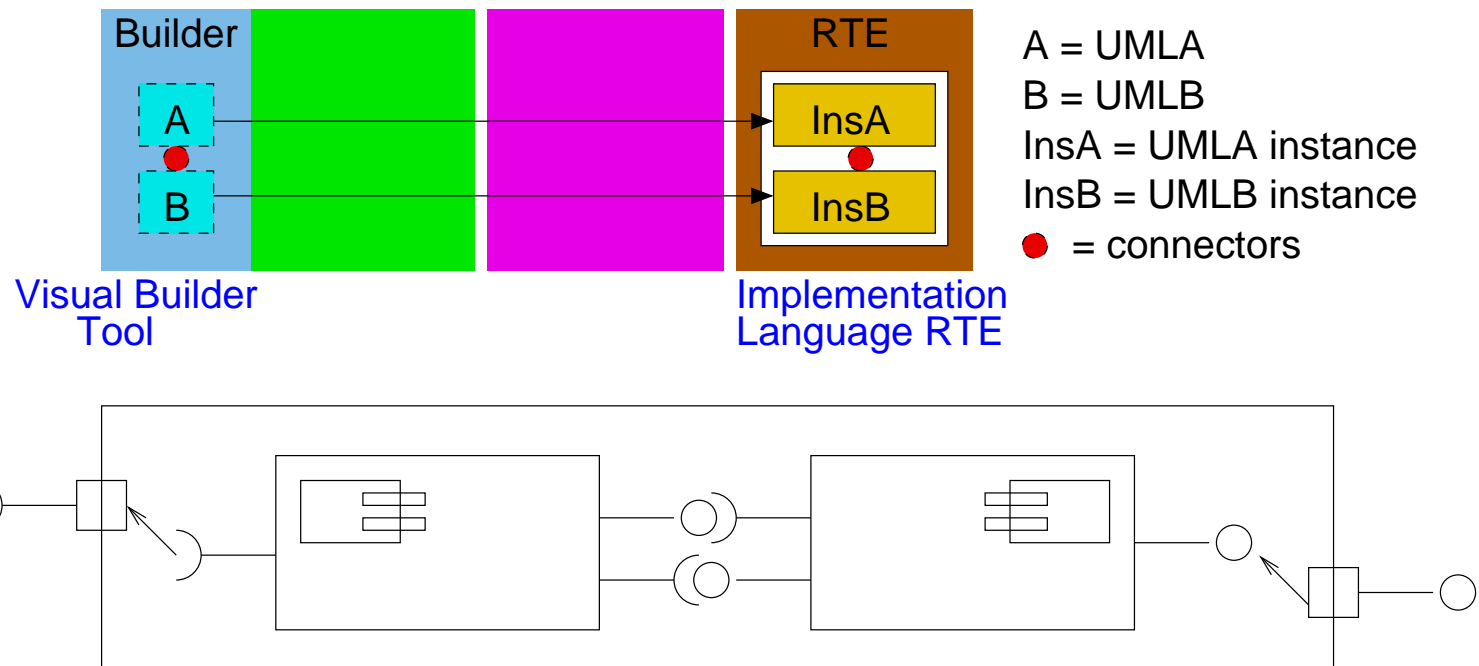
- The **visual builder tool** is the builder
- There is **no repository**
- There is **no assembler**

UML 2.0: Composition

In **design phase**, the **architecture** for the whole system is designed.

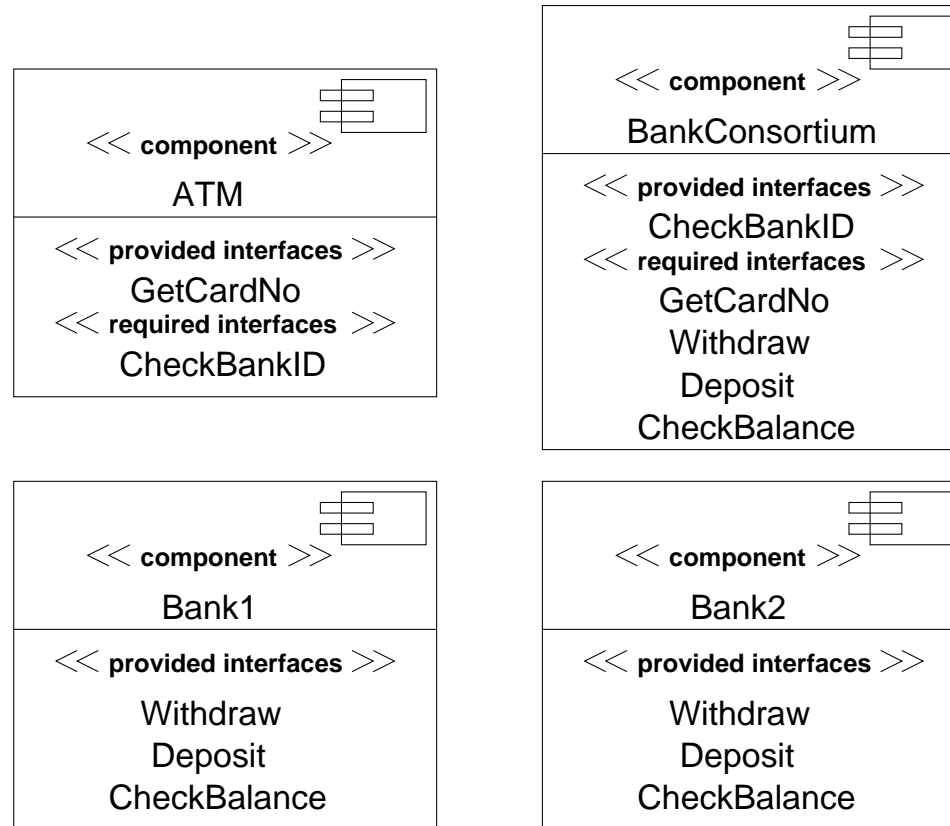


UML 2.0: Summary



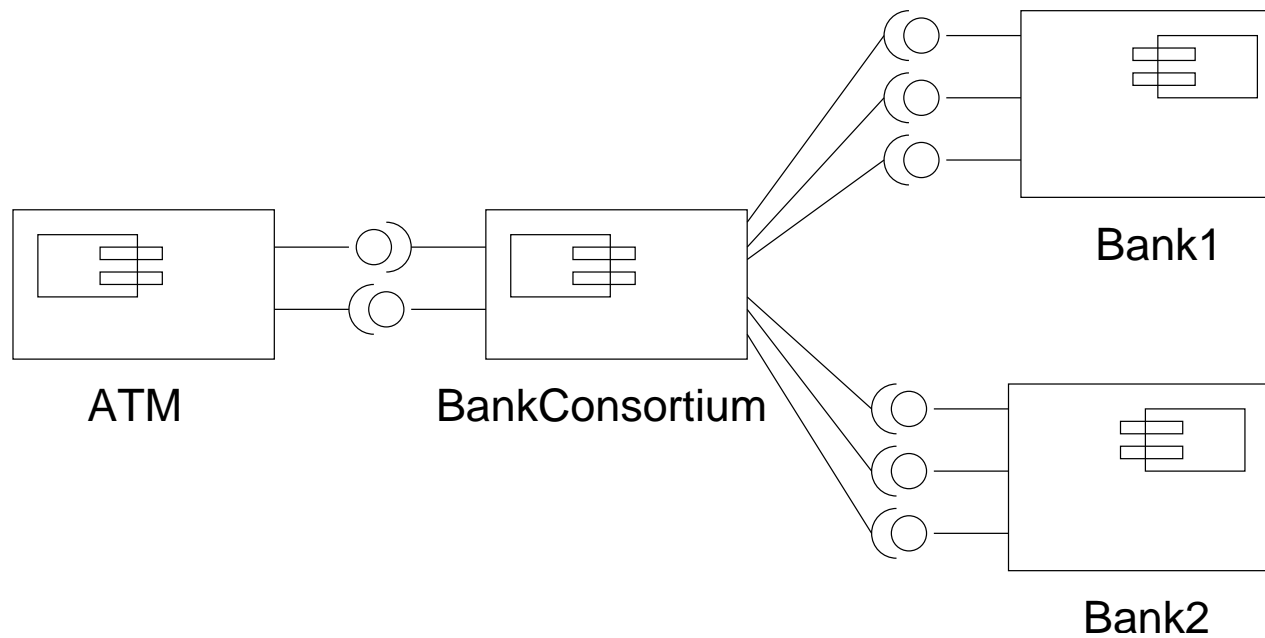
UML 2.0: Example

Consider a simple bank system that is implemented by ATM, BankConsortium, Bank1 and Bank2 components.



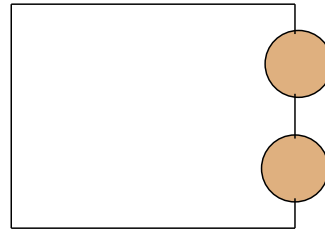
UML 2.0: Example (Continued)

In **design phase**, the architecture for the whole system is designed.



PECOS: Components

In **PECOS*** a component is a unit of **design** which has a specification and an implementation.



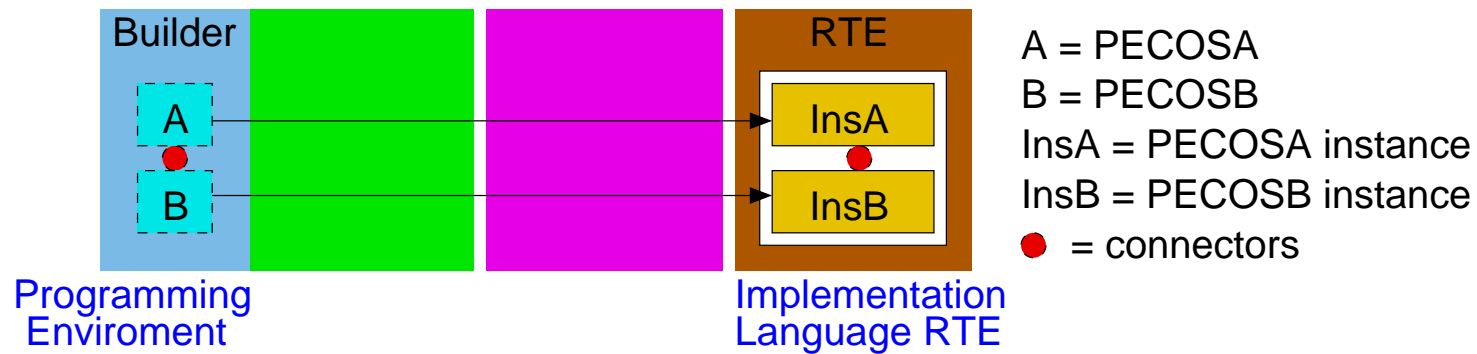
- Every component has a **name**, a number of **property bundles**, a set of **ports**, and **behaviour**
- Ports are **interfaces** of components

PECOS components are specified in the **CoCo** (Component Composition) language.

* PErvasive COmponent Systems

PECOS: Builder

Components in PECOS are **constructed** in a programming environment such as **Eclipse**.



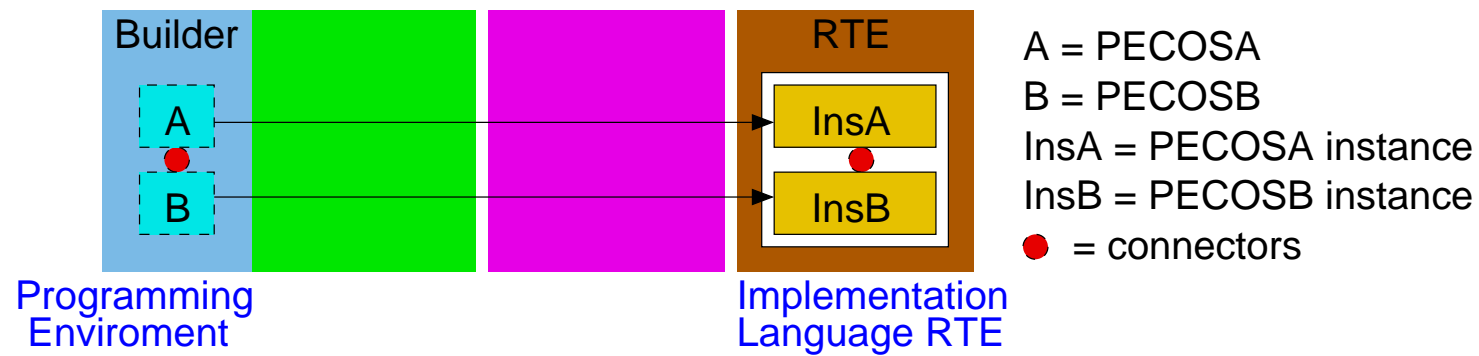
- The **programming environment** is the builder
- There is **no repository**
- There is **no assembler**

PECOS: Composition

In the **design phase**, components are **composed** by linking their **ports** with **connectors**.

Implementation of PECOS components is usually done in **Java** or **C++**, and so the run-time environment in the deployment phase is that for Java or C++.

PECOS: Summary



PECOS: Example

Consider a device that is assembled from **Clock**, **Display**, **EventLoop** and **DigitalDisplay** components.

```
component Clock {
    output long msecs;
}
```

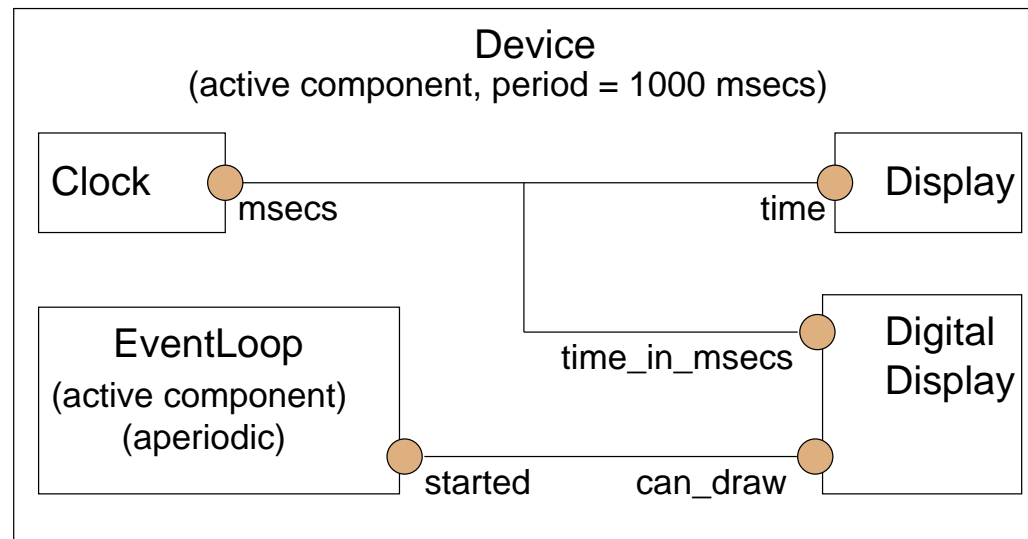
```
component Display {
    input long time;
}
```

```
active component EventLoop {
    output bool started;
}
```

```
component DigitalDisplay {
    input long time_in_msecs;
    input bool can_draw;
}
```

PECOS: Example (Continued)

In the **design phase**, the architecture for the device is designed:



```
active component Device {
    Clock clock;    Display display;    DigitalDisplay digitalDisplay;
    EventLoop eventLoop;
    connector time(clock.msecs, display.time,
                  digitalDisplay.time_in_msecs);
    connector eventLoop_started (eventLoop.started,
                                 digitalDisplay.can_draw);
}
```

Pin: Components

In **Pin**, a component is an **architectural unit** that specifies a **stimulus-response** behaviour by a set of **ports (pins)**.

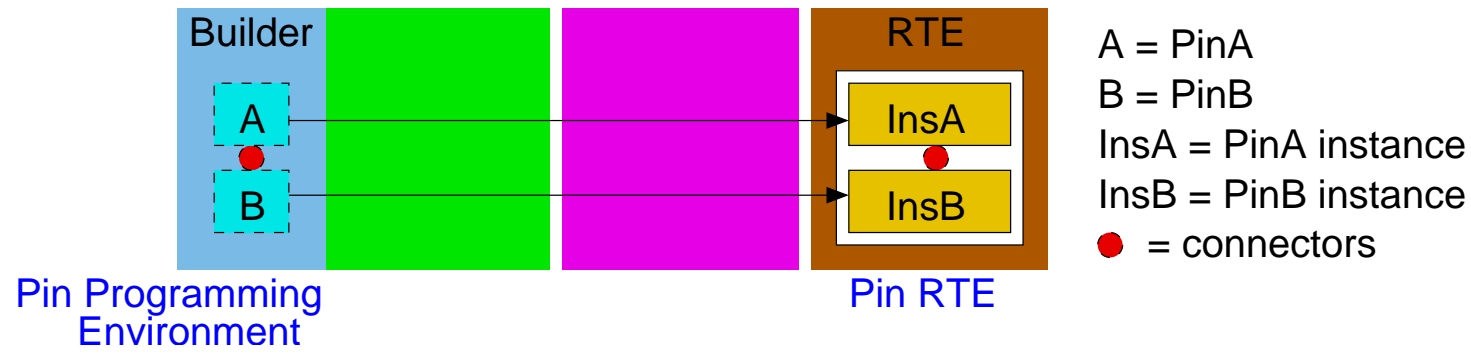


- A component is represented by a set of **sink pins** and **source pins** together with the component's **behaviour**.

Components are defined in **CCL** (Construction and Composition Language).

Pin: Builder

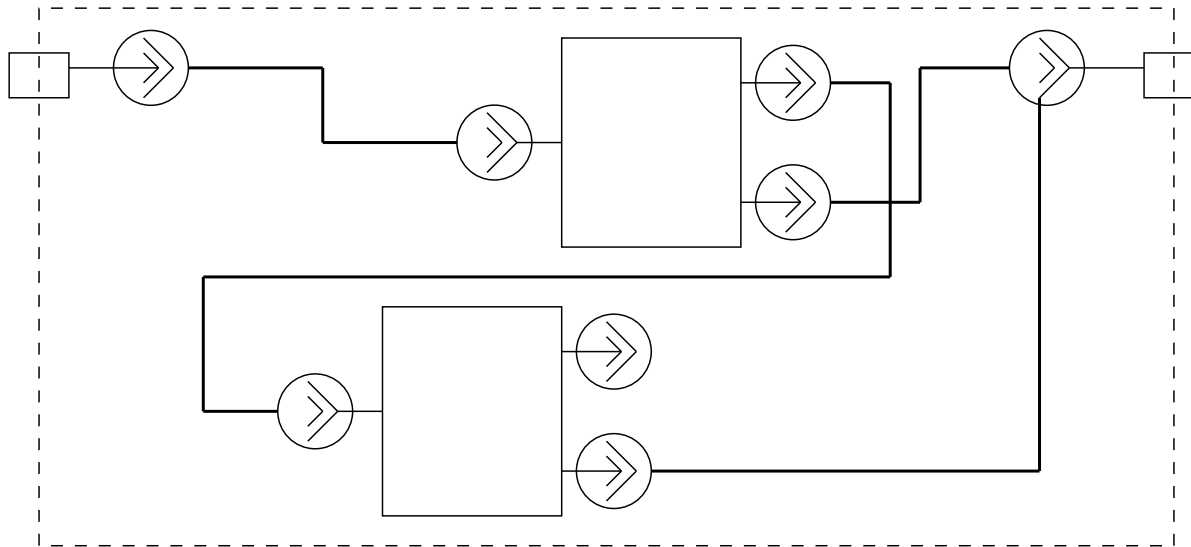
Pin components can be constructed in the CCL programming environment.



- The CCL programming environment is the builder
- There is no repository
- There is no assembler

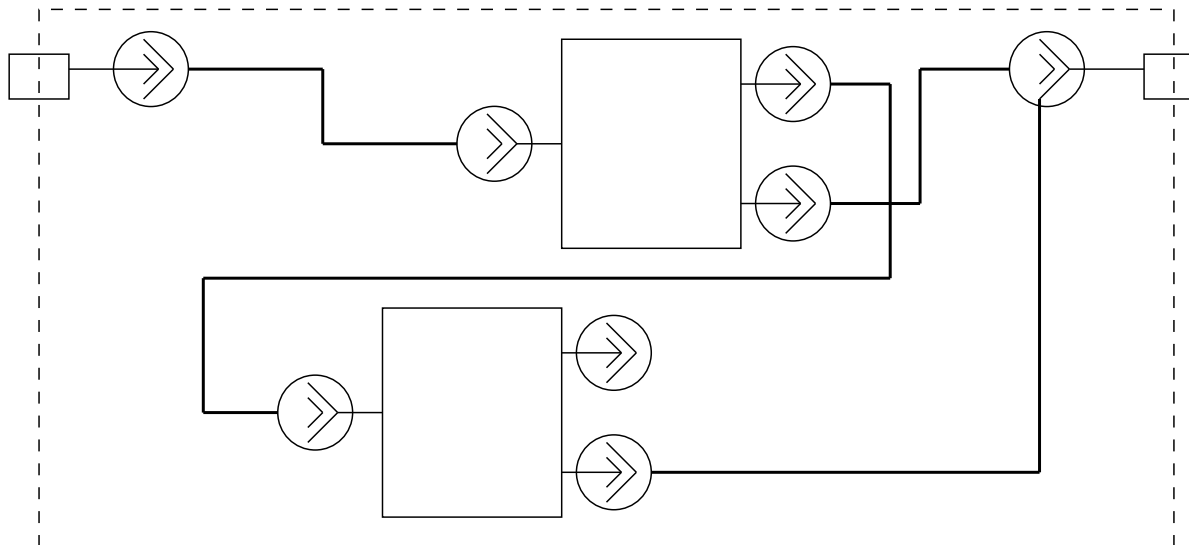
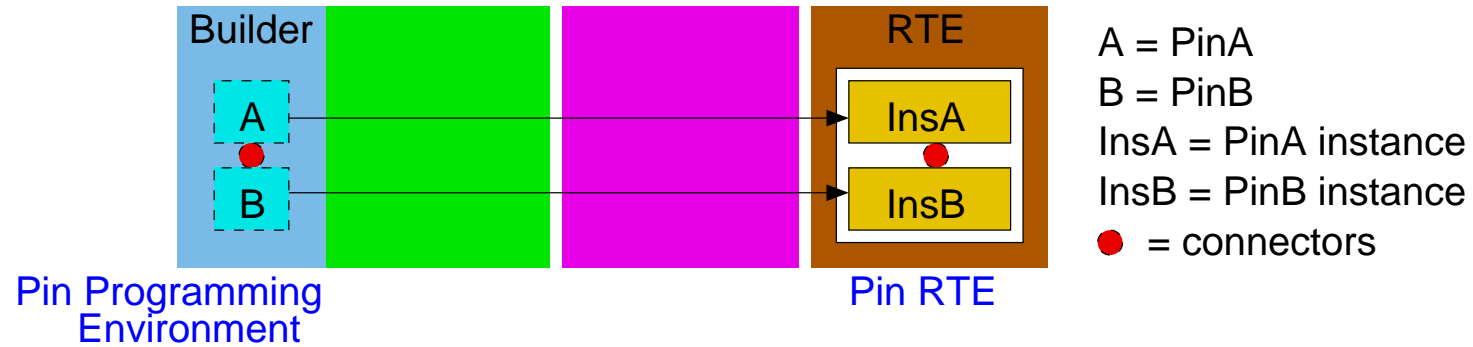
Pin: Composition

In **design phase**, components are **composed** by **connectors** that link the source pins of one component to the sink pins of another.



In **deployment phase**, **implementations** are usually generated by the **CCL processor** and components are executed in the **Pin run-time environment**.

Pin: Summary



Pin: Example

Consider a simple component **AComp** that is specified with both structural and behavioural aspects in CCL.

```

component AComp() {
  sink asynch receive();
  source unicast send();
  source publish();
  threaded react mission (receive, send, publish) {
    start -> ready { }
    ready -> work {
      trigger ^receive();
      action ^send();
    }
    work -> log {
      trigger ^send();
      action ^publish();
    }
    log -> ready {
      trigger ^publish();
      action ^receive();
    }
  }
}

```

Pin: Example (Continued)

In **design phase**, components **comp1** and **comp2** of type AComp are composed by connectors that link comp1's source pin (send) to comp2's sink pin (receive) to an **assembly** – **AComposite**.

```
environment E {
  service Receive() {
    source unicast receive();
    threaded react received(receive) {
      start -> ready {};
      ready -> work {
        action ^receive;
      }
      work -> ready {};
    }
  }

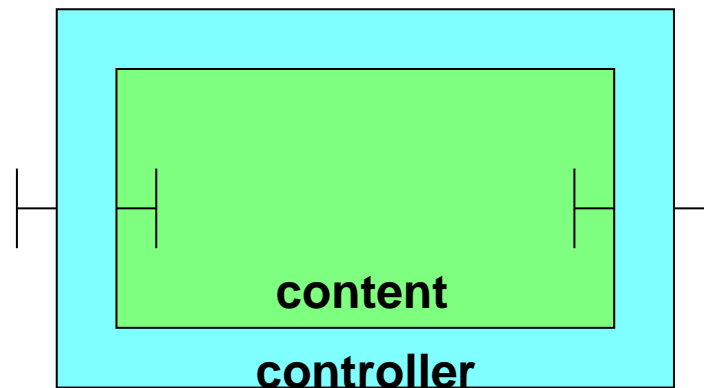
  service Send() {
    sink asynch send();
    threaded react sending(send) {
      start -> ready {};
      ready -> work {
        action ^send;
      }
      work -> ready {};
    }
  }
}
```

```
assembly AComposite() (E) {
  assume {
    E: Send compositesend();
    E: Receive compositereceive();
  }
  AComp comp1(), comp2();
  compositesend:receive ~> comp1:receive;
  comp1:send ~> comp2:receive;
  comp1:publish ~> compositesend;
  comp2:publish ~> compositesend;
}
```

Fractal: Components

In **Fractal**, a component is a **run-time entity** that behaves like an **object**.

A Fractal component comprises a **content** and a **controller**.



- The **content** contains its **interfaces** and **implementation**
- The **controller** defines the **control behaviour** associated with the component

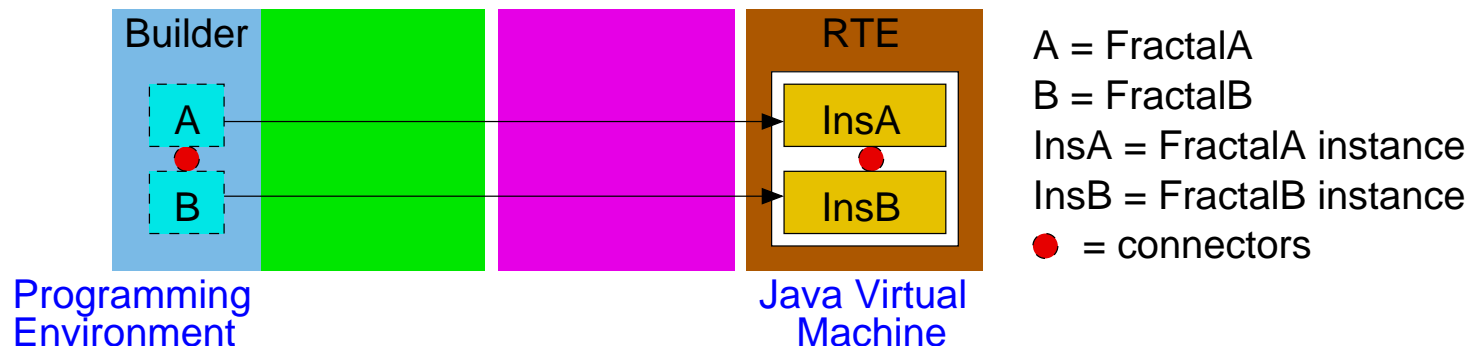
Fractal: Components (Continued)

Interface Definition Languages (e.g. OMG IDL) are used to define generic interfaces that can be implemented by components in specific programming languages.

Current **Fractal API** is extended and modified from **Java API** with JavaBeans-like **introspection** facilities.

Fractal: Builder

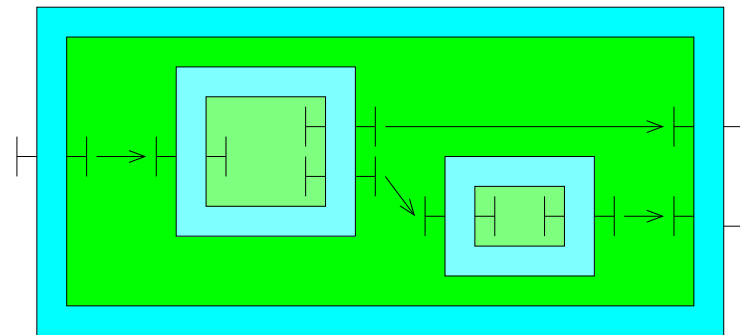
Fractal components are **constructed** in a **programming environment** with Fractal APIs.



- The **programming environment** is the **builder**
- There is **no repository**
- There is **no assembler**

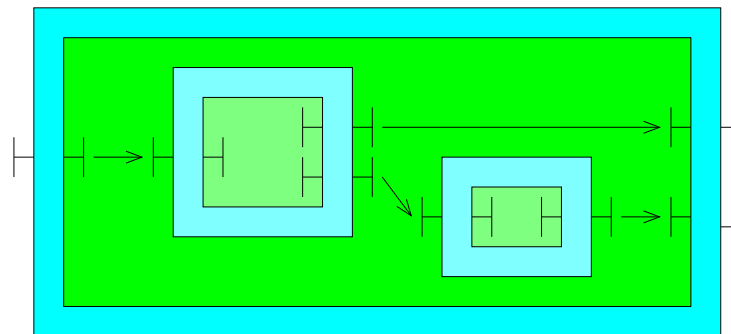
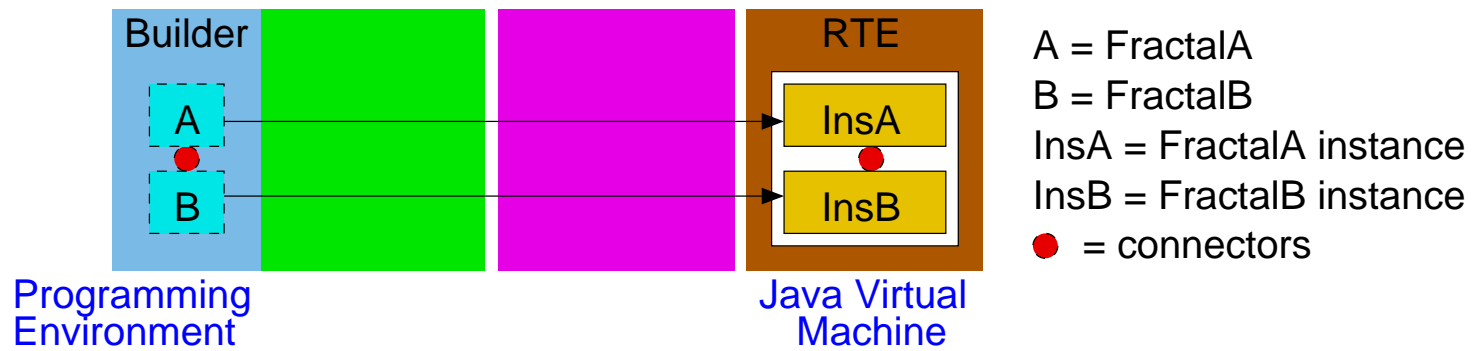
Fractal: Composition

In **design phase**, Fractal components are **composed** by **method calls** through **connectors**.



The **Java Virtual Machine** serves as the **run-time environment** for Fractal components.

Fractal: Summary



Fractal: Example

Consider a **Stopwatch** device that comprises a **Countdown** component and a **Display** component.

```
public interface Count{
} void count ();
public interface ControlTotal extends AttributeController{
    int getTotal ();
    void setTotal (int total);
}
public class Countdown implements Count, ControlTotal{
    private int total = 0;
    public void count () {
        for (int i = total; i > 0; i--) {
            System.out.print(i);
        }
    }
    public int getTotal (){
        return total;
    }
    public void setTotal (final int total){
        this.total = total;
    }
}
```

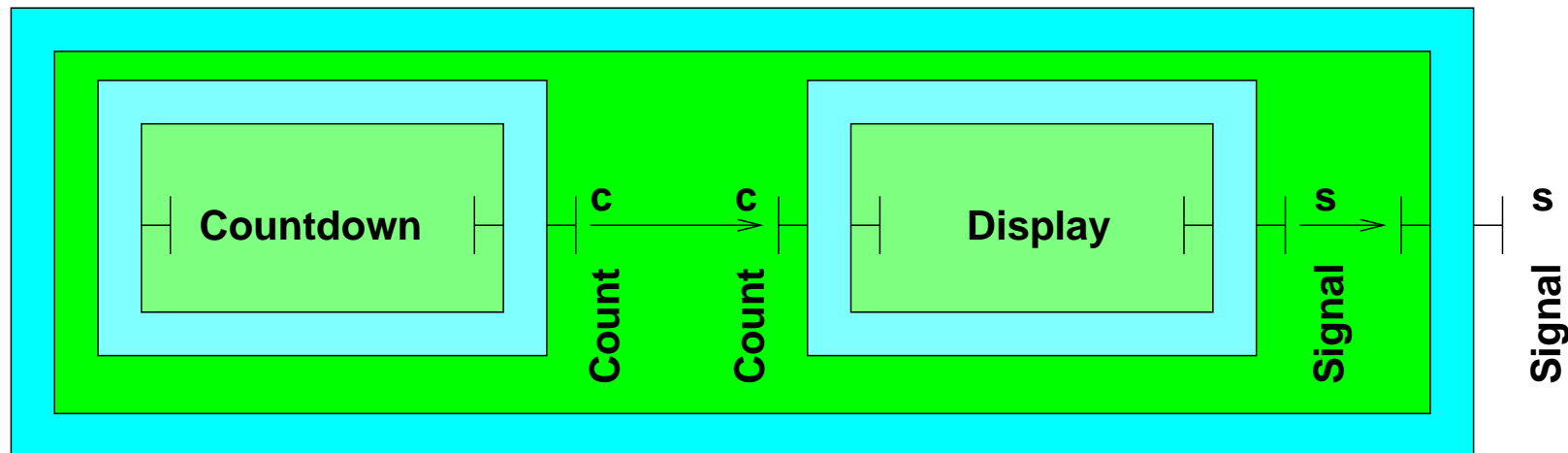
Countdown component

```
public interface Signal{
    void display ();
}
public class Display implements Signal, BindingController{
    private Count count;
    public void display () {
        count.count();
    }
    public String[] listFc (){
        return new String[] {"c"};
    }
    public Object lookupFc (final string disstr){
        if (disstr.equals("c")) {
            return count;
        }
        return null;
    }
    public void bindFc (final string disstr, final Object countobj){
        if (disstr.equals("c")) {
            count = (Count)countobj;
        }
    }
    public void unbindFc (final string disstr){
        if (disstr.equals("c")) {
            count = null;
        }
    }
}
```

Display component

Fractal: Example (Continued)

The **Stopwatch** device is implemented by constructing and composing **Countdown** and **Display**.



The instances of **Countdown** and **Display** are composed by **method calls**.

Fractal: Example (Continued)

```

Component boot = Fractal.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
ComponentType deviceType = tf.createFcType(new InterfaceType[] {
    tf.createFcltType("s", "Signal", false, false, false);
});
ComponentType displayType = tf.createFcType(new InterfaceType[] {
    tf.createFcltType("s", "Signal", false, false, false),
    tf.createFcltType("c", "Count", true, false, false)});
ComponentType countdownType = tf.createFcType(new InterfaceType[] {
    tf.createFcltType("c", "Count", false, false, false),
    tf.createFcltType("total-controller", "ControlTotal", false, false, false)});
GenericFactory cf = (GenericFactory)boot.getFcInterface("generic-factory");
Component deviceTmpl = cf.newFcInstance(deviceType, "deviceTemplate",
    new Object[] {"composite", "Device"});
Component displayTmpl = cf.newFcInstance(displayType, "displayTemplate",
    new Object[] {"primitive", "Display"});
Component countdownTmpl = cf.newFcInstance(countdownType, "countdownTemplate",
    new Object[] {"parametricPrimitive", "Countdown"});
ControlTotal ct = (ControlTotal)countdownTmpl.getFcInterface("total-controller");
ct.setTotal(100);
ContentController cc = (ContentController)deviceTmpl.getFcInterface("content-controller");
cc.addFcSubComponent(displayTmpl);
cc.addFcSubComponent(countdownTmpl);
((BindingController)deviceTmpl.getFcInterface("binding-controller")).bindFc("s", displayTmpl.getFcInterface("s"));
((BindingController)displayTmpl.getFcInterface("binding-controller")).bindFc("c", countdownTmpl.getFcInterface("c"));
Component stopwatchdevice = ((Factory)deviceTmpl.getFcInterface("factory")).newFcInstance();
(LifeCycleController)stopwatchdevice.getFcInterface("lifecycle-controller").startFc();
((Signal)stopwatchdevice.getFcInterface("s")).display();
    
```

A Taxonomy based on Composition

Category	Models	Characteristics				
		DR	RR	CS	DC	CP
1	JavaBeans	✓	×	×	×	✓
2	EJB, COM, .NET, CCM, web services	✓	×	✓	×	×
3	Koala, SOFA, KobrA	✓	✓	✓	✓	×
4	ADLs, UML2.0, PECOS, Pin, Fractal	×	×	✓	×	×

DR In design phase new components can be deposited in a repository

RR In design phase components can be retrieved from the repository

CS Composition is possible in design phase

DC In design phase composite components can be deposited in the repository

CP Composition is possible in deployment phase

Conclusion

- **Best fit:** Category 3 (Koala, KobrA, SOFA)
(product lines, repositories, composites, units of designs)
- **Worse fit:** Category 4 (ADLs)
(design only)
- **Middle of the road:** Categories 1 & 2
(repositories, binaries)
- **No component model with composition in both design and deployment phases**