SWA – Architectural Design Methods

Lecture 03

BIL428 Software Architectures

Asst.Prof.Dr. Mustafa Sert
msert@baskent.edu.tr

Department of Computer Engineering, Başkent University
Ankara 06810 TURKEY

Contents

- Context diagrams
- SWA description languages

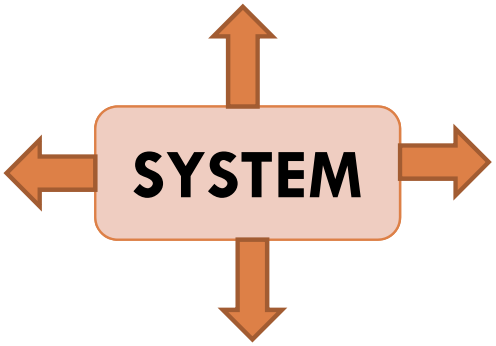
Context Diagrams (CDs)

- A context diagram shows
  - What is in
  - What is out

  Of the system under construction and the external entities with which it interacts
- Shows how the system (under construction) interacts with the outside world
  - External entities may be
    - Humans, other computer systems, or physical systems such as remote devices (e.g., sensors, controlled devices, etc.)

Context Diagrams

- Every system has an architecture
  - Every system is composed of elements and there are relationships among them
- A pure context diagram does not give any architectural detail about an entity
  - Black box approach



  - In practice, most context diagrams show some internal structure of the entity being put in context

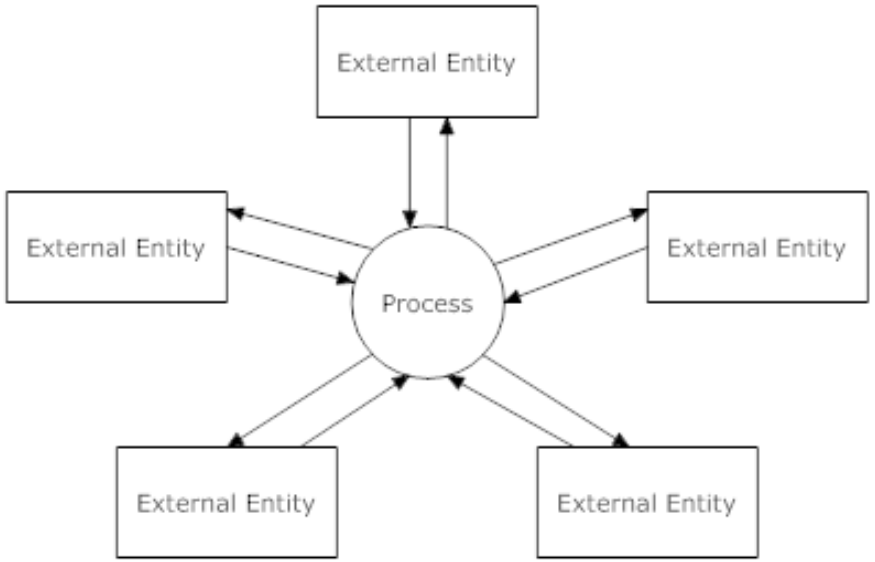
Context Diagrams

- Do NOT show
  - Any temporal information, such as order of interactions or data flow..
  - The conditions, under which data is transferred, triggered, messages transmitted, etc.

- There are two way to show CDs
  - Visual
    - UML based, others..
  - Textual
    - Architecture Description Languages (ADLs)

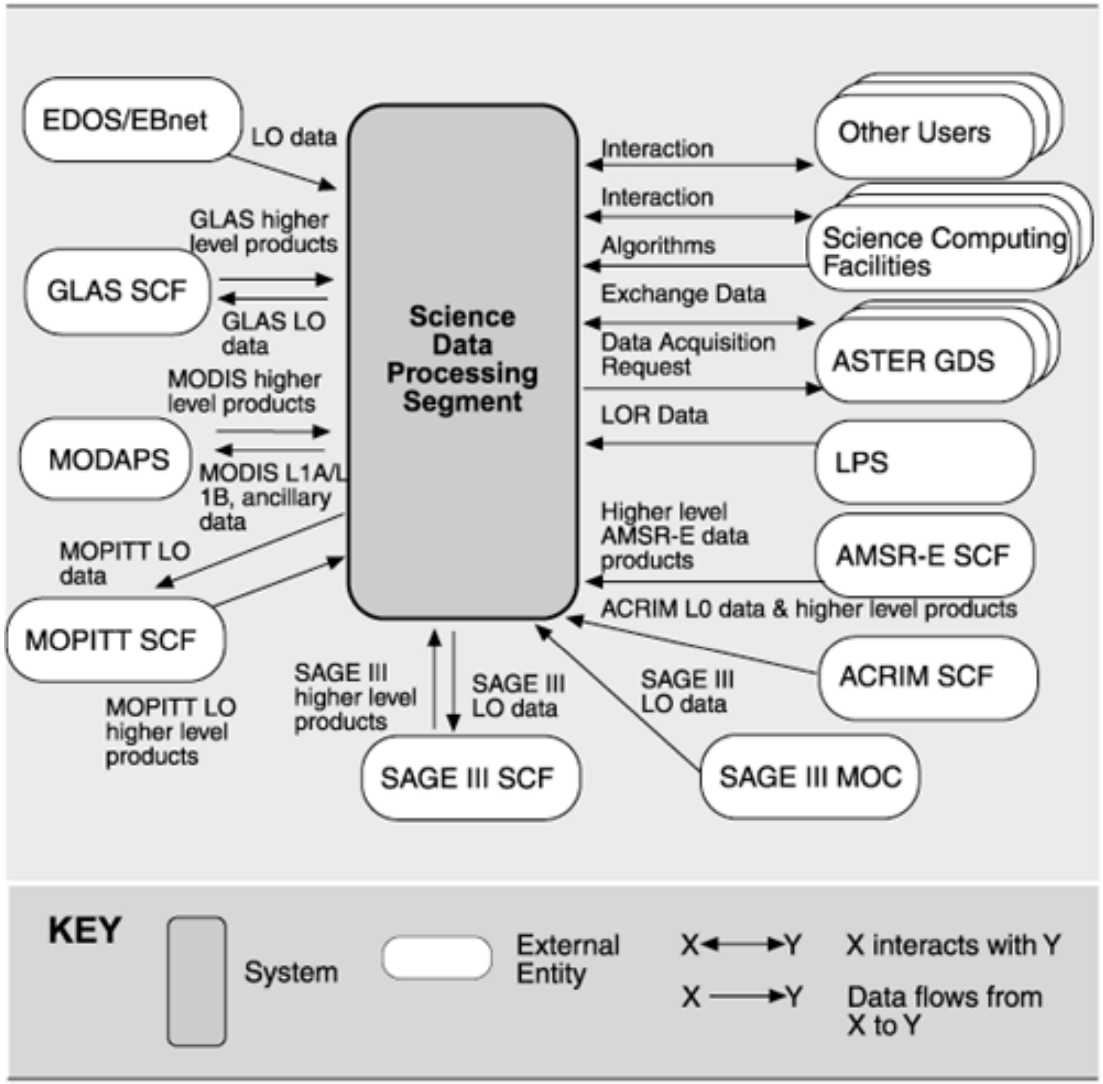
Notations for Context Diagrams

- Informal Notations
  - Circle-and-line drawing, with the entity being defined, depicted in the center as a circle..
  - The entity interactions that are external to it depicted as various shapes, and lines btw them to indicate connections



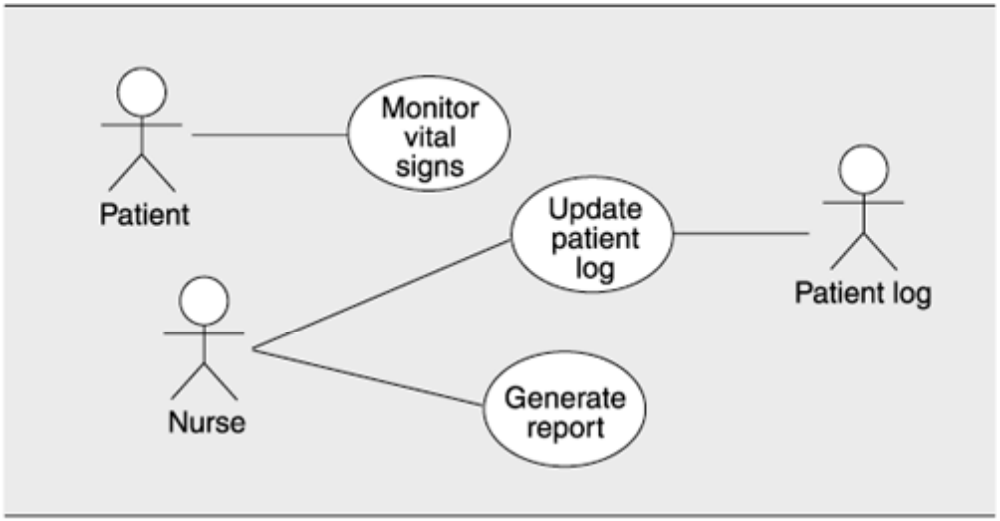
Notations for Context Diagrams

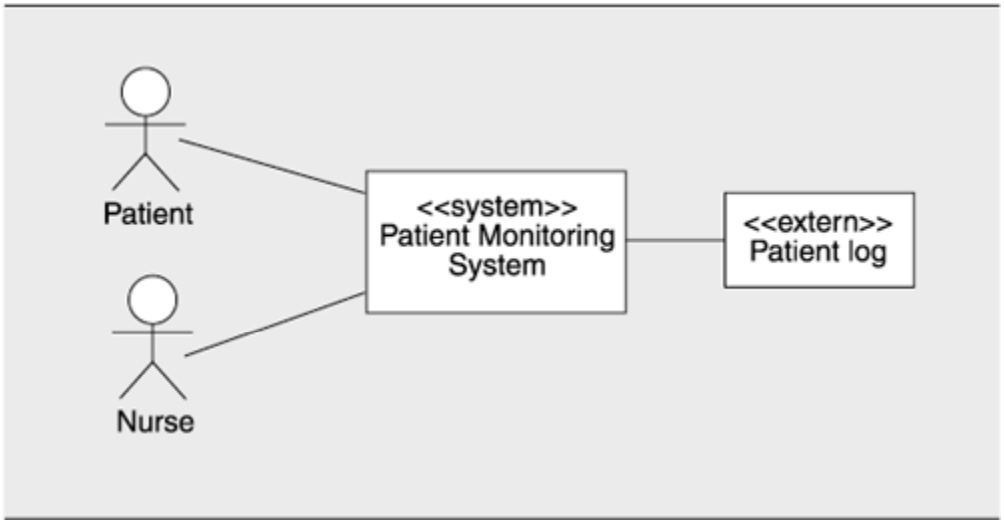
- Example from the text book (Documenting SWA, Clements)



Notations for CDs- UML

- UML does not have an explicit mechanism for CDs
- UML's way to system context is using **use-case diagrams**





Architecture Description Languages

Architecture Description Languages (ADLs)

- **Why we need ADLs?**
  - Many programs are ambiguous
  - Programming languages (PLs) are too low-level
- **ADLs provide**
  - A precise but abstract description!

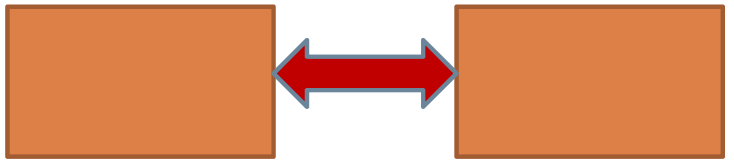
Several ADLs have been developed..

- Acme (developed by CMU – Carnegie Mellon Univ.)
- AADL (standardized by SAE – Society of Automotive Engineers)
- C2 (developed by UCI – University of California, Irvine)
- Darwin (developed by Imperial College London)
- Wright (developed by CMU)
- …

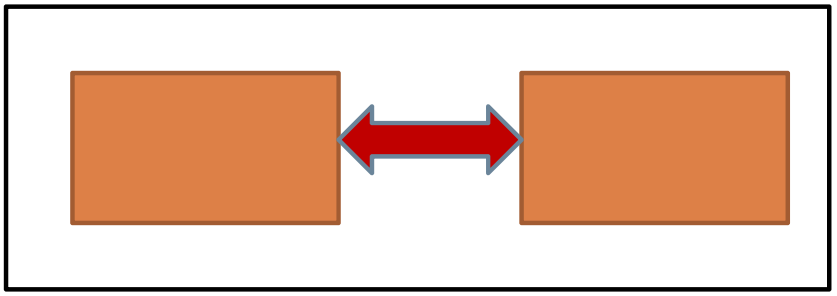
Elements of ADLs

- **Components**
  - Primitive building blocks

- **Connectors**
  - Mechanisms for combining components



- **Configuration**
  - Rules for referring to a combination of components and combinations



Component

- 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 third-party composition

Components vs. Objects

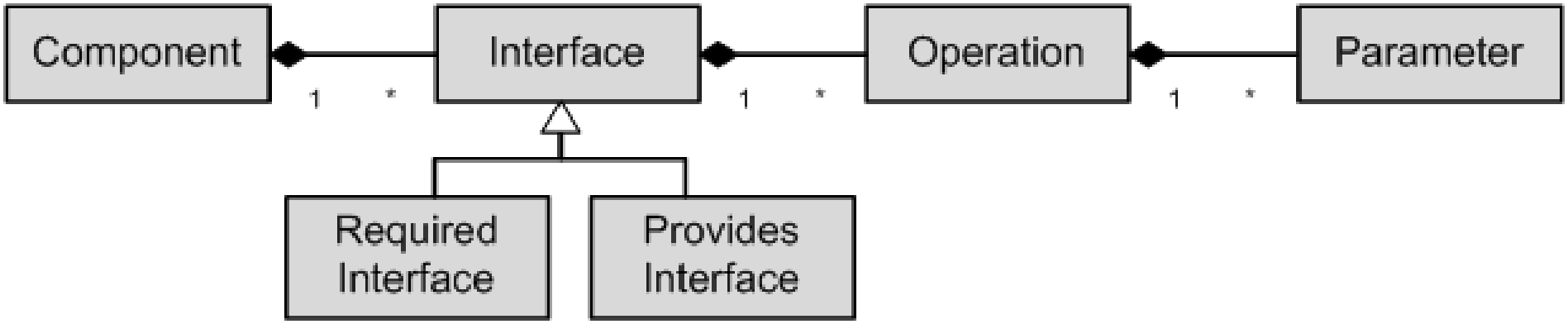
- Components are considered to be a **higher level of abstraction** than objects and thus they do NOT share state
- Components have a **more extensive set of intercommunication mechanisms** whereas objects usually use the messaging mechanism
- Components are **often larger units of granularity** than objects and have complex actions in their interfaces
- A component **can be viewed as a collection of objects** in which the objects co-operate with each other and intertwined tightly.
- Components **often use persistent storage** whereas objects have local state
- Components are **deployable entities**
- Components are usually **language-independent**

Component Interfaces

A component has

- **Provides Interface**
  - Defines the services that are provided by the component to other components..
- **Requires Interface**
  - Defines the services that specifies what services must be made available for the component to execute as specified..

Component - Metamodel



Semantics of Components

- We need documentation for components to decide if interfaces are really compatible
- A component's behavior can be achieved through **contracts**
- A contract is comprised of
  - The **invariant** – the global constraint which the component will maintain
  - The **pre-condition** – the constraints which need to be met by the client
  - The **post-condition** – the constraints which the component promised to establish in return

Interface Incompatibility

- **Parameter incompatibility**
  - Where operations have the same name, but are of distinct types
- **Operation incompatibility**
  - Where the names of operations in the composed interfaces are different
- **Operation incompleteness**
  - Where the provides interface of one component is a subset of the requires interface of another

Component Composition

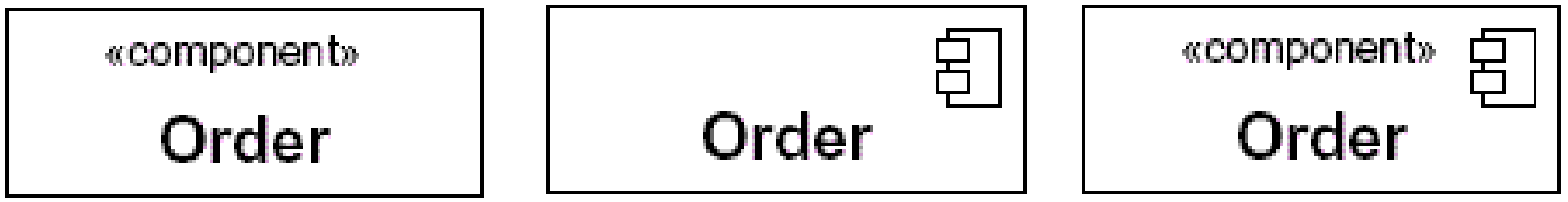
- The process of **assembling components** to create a system
- Composition involves **integrating components** with each other and with the component infrastructure
- When composing
  - You must consider both the **functional** and **non-functional** requirements

Component Models

- Is a definition of standards for component implementation, documentation, and deployment
- Some examples are
  - EJB model – Enterprise Java Beans
  - COM+ (Component Object Model +) model (by Microsoft) – .NET model
  - CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture) component model
- The model specifies
  - How interfaces should be defined
  - How the elements that should be included in an interface definition

Modeling Components in UML

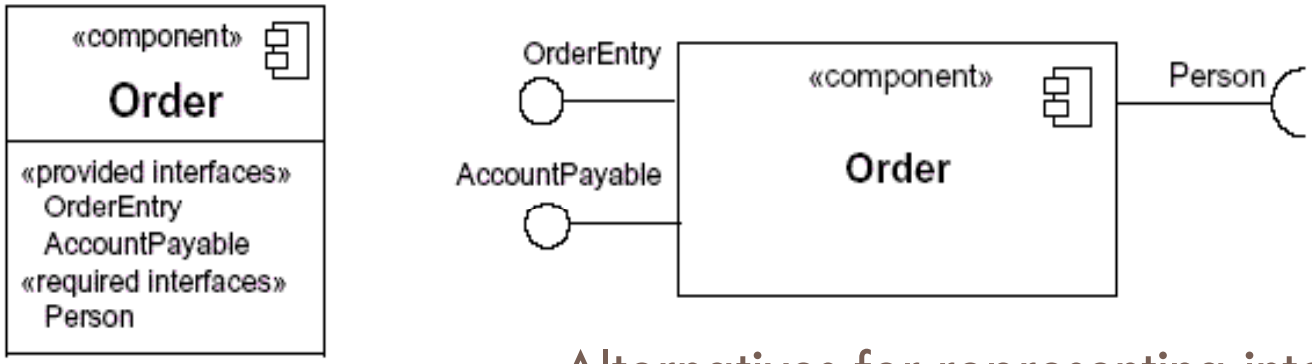
- Captures the physical structure of the implementation
- When drawing a component on a diagram, it is important that you always include the component stereotype text (the word "component" inside double angle brackets, as shown in the figure) and/or icon.



Alternatives for representing components

Modeling Components in UML

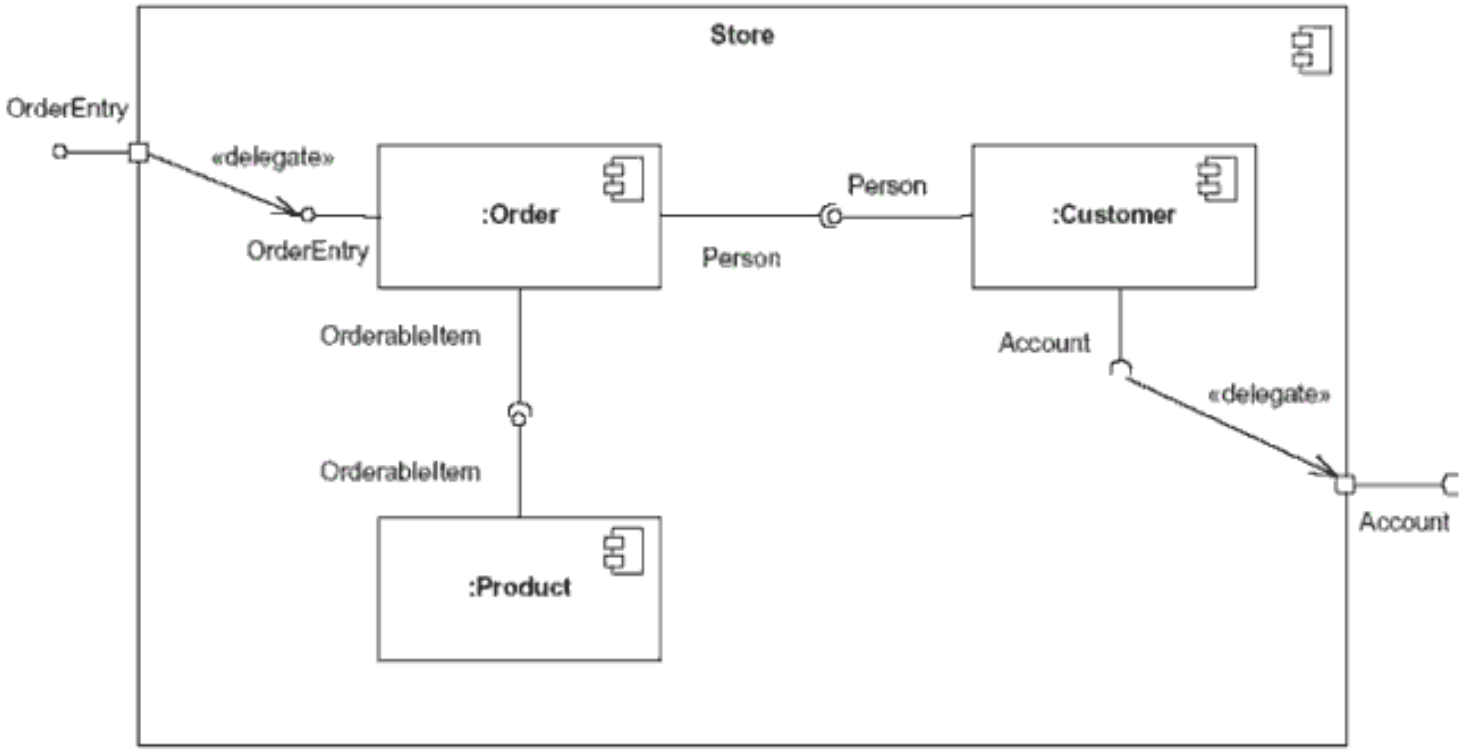
- Components may have both **provide** and **require interfaces**
- An **interface** is the definition of a collection of one or more methods, and zero or more attributes, ideally one that defines a cohesive set of behaviors
- A **provide interface** is modeled using the lollipop notation and a **required interface** is modeled using the socket notation



Alternatives for representing interfaces

Modeling Components in UML

- A **port** is a named interface on a component, it defines a set of operations and events that are provided by a component or that are required from it's environment.
  - Ports are illustrated by small squares on the sides of classifiers



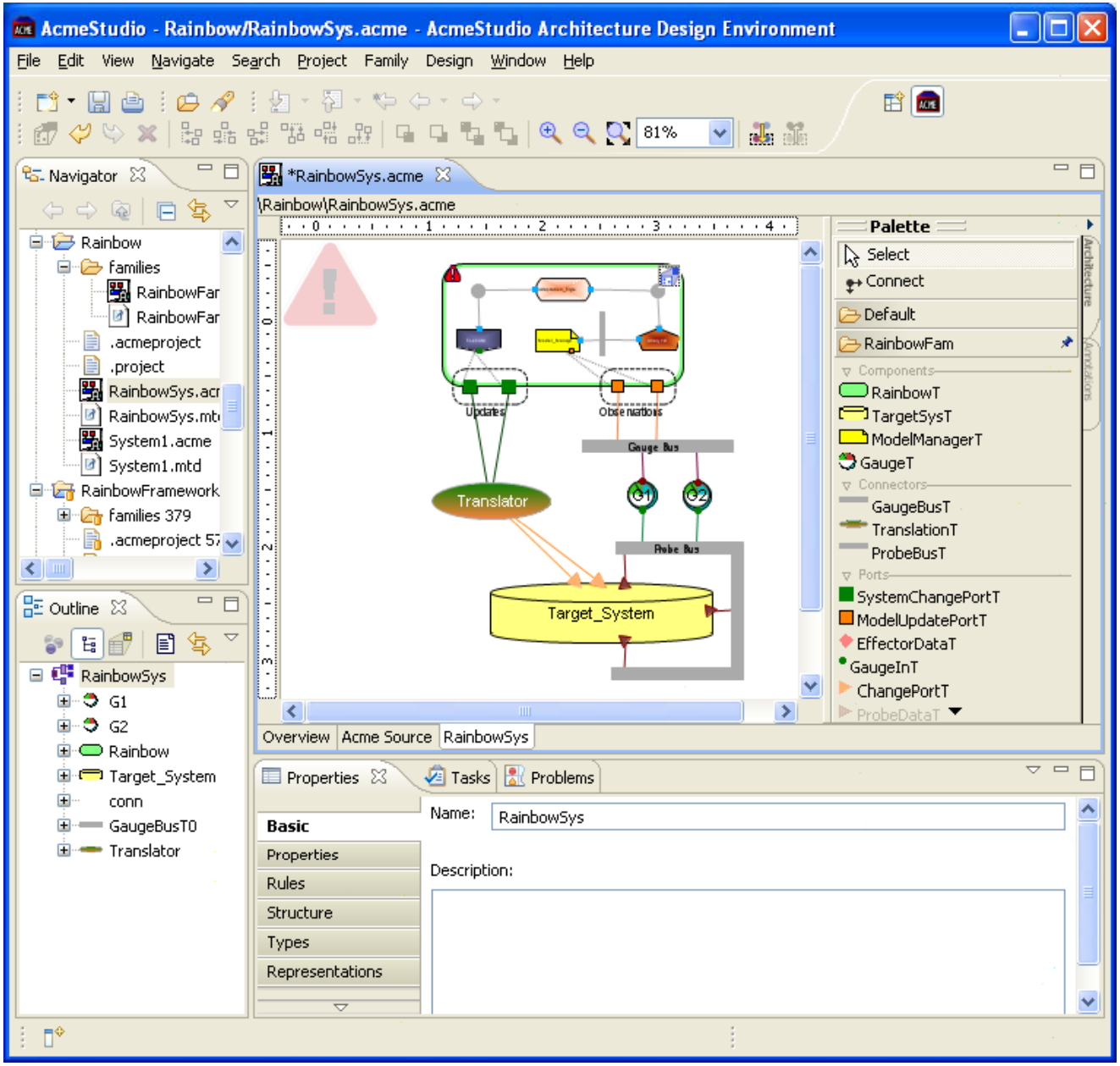
Example component diagram

ADL Example – Wright

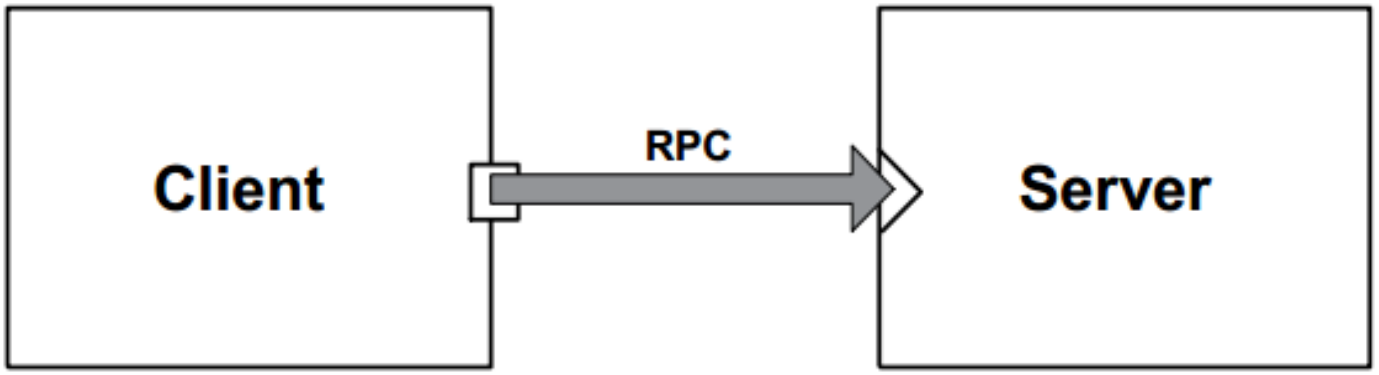

```
System simple_cs
  Component client =
    port send-request = [behavioral spec]
    spec = [behavioral spec]
  Component server =
    port receive-request= [behavioral spec]
    spec = [behavioral spec]
  Connector rpc =
    role caller = (request!x -> result?x ->caller) ^ STOP
    role callee = (invoke?x -> return!x -> callee) [] STOP
    glue = (caller.request?x -> callee.invoke!x
          -> callee.return?x -> callee.result!x -> glue) [] STOP
  Instances
    s : server; c : client; r : rpc
  Attachments :
    client.send-request as rpc.caller
    server.receive-request as rpc.callee
  end simple_cs.
```

ADL Example – ACME

- You may use ACME Studio or ACME plugin for Eclipse for architectural designs, visual and/or textual..



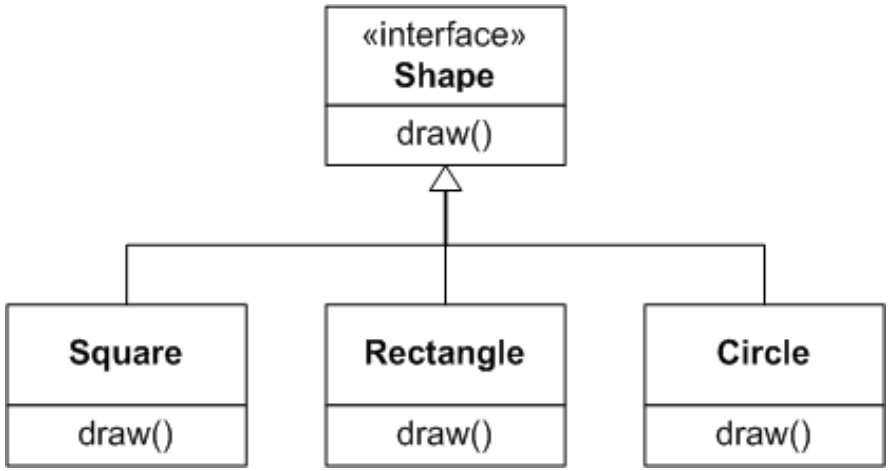
ADL Example – ACME

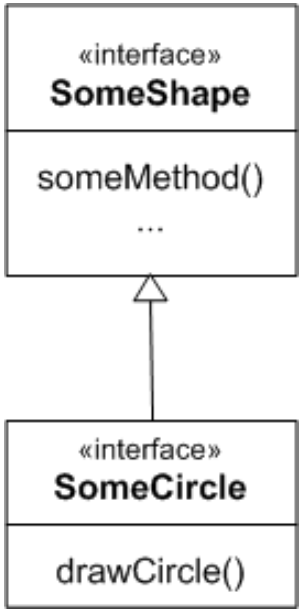



```
System simple_cs = {
   Component client = {Port send-req}
   Component server = {Port receive-req}
   Connector rpc = {Roles {caller, callee}}
   Attachments :
     {client.send-req to rpc.caller;
      server.receive-req to rpc.callee}
}
```

Example – Interfacing Problem btw Objects

- We are supposed to draw some shapes by implementing the Shape interface. Here are the constraints:
  - There is an existing class, namely SomeCircle, that fully supports what we want. And we want to use it, but:
    - It is implemented with a different interface
    - The name of drawing method differs than ours
    - There is no source-code of the existing class, only the binary..
  - We want to use it "as is"





Solution – Interfacing Problem btw Objects

- Use the structural ADAPTER design pattern
  - There are two types of adapter pattern
    - Object Adapter Pattern vs. Class Adapter Pattern
  - We will make use of Object Adapter Pattern:

```
public class Circle implements Shape {
    SomeCircle sc;
    …

    public Circle() {
            sc = new SomeCircle();
    }

    public void draw() { sc.drawCircle(); }
    …
}
```

Lecture Example

- Try to solve the previous example by using “Class Adapter Pattern”