Software Reuse and Component-Based SE

ITSE422

Lecture #3: CBSE

Introduction and Basic Concepts & Modeling Components with UML

Main References

- Ian Sommerville, *Software Engineering*, 8th edition, chapter 19.1 (*Components and component models*)
- Ivica Crnkovic, Magnus Larsson. *Building reliable component based software systems*, Artech House, 2002.
- Roger S. Pressman, Software Engineering: A Practitioner's Approach, Eighth Edition, McGraw-Hill Higher Education, 2015

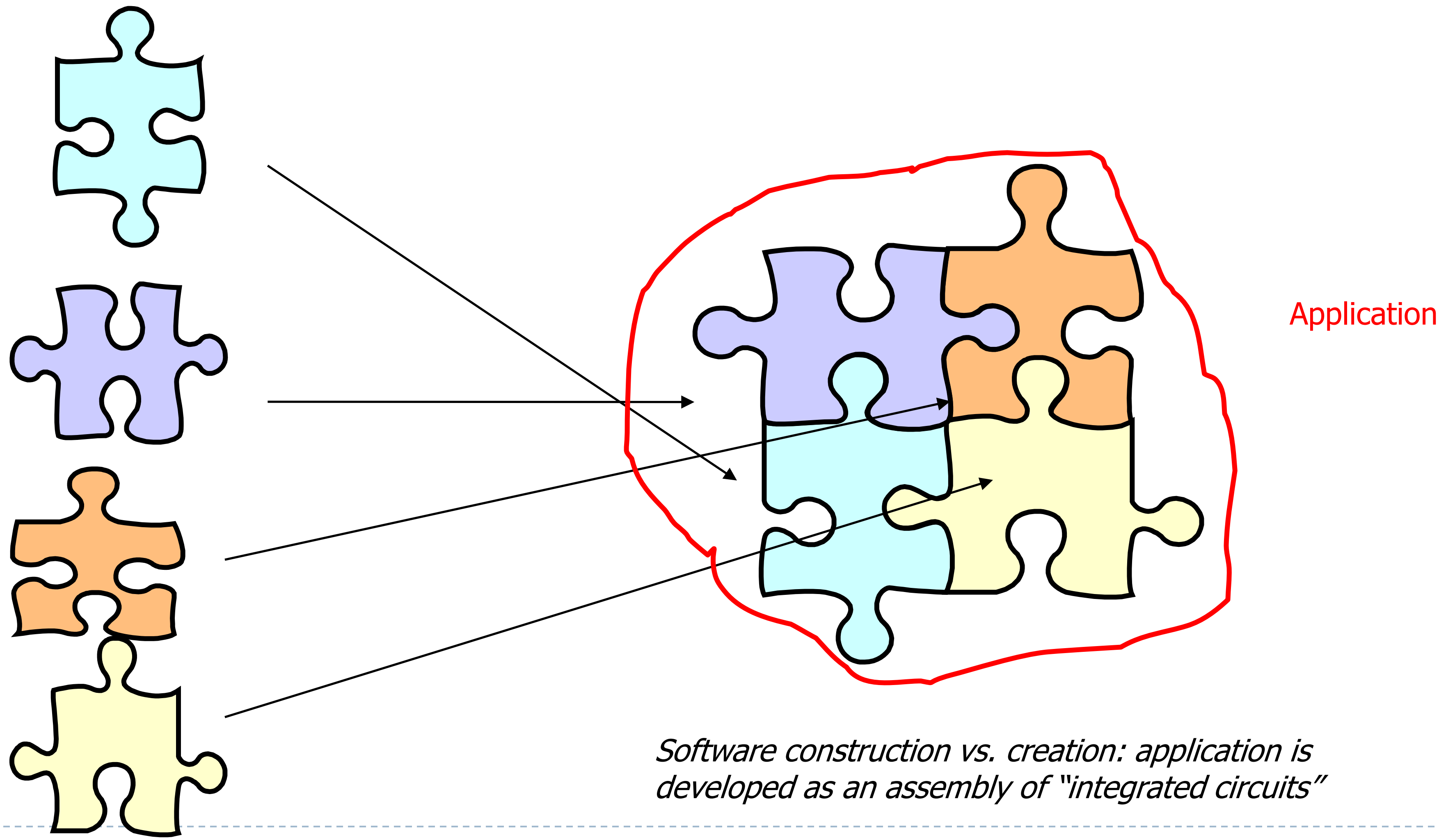
Component based development

"Systems should be assembled from existing components"

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

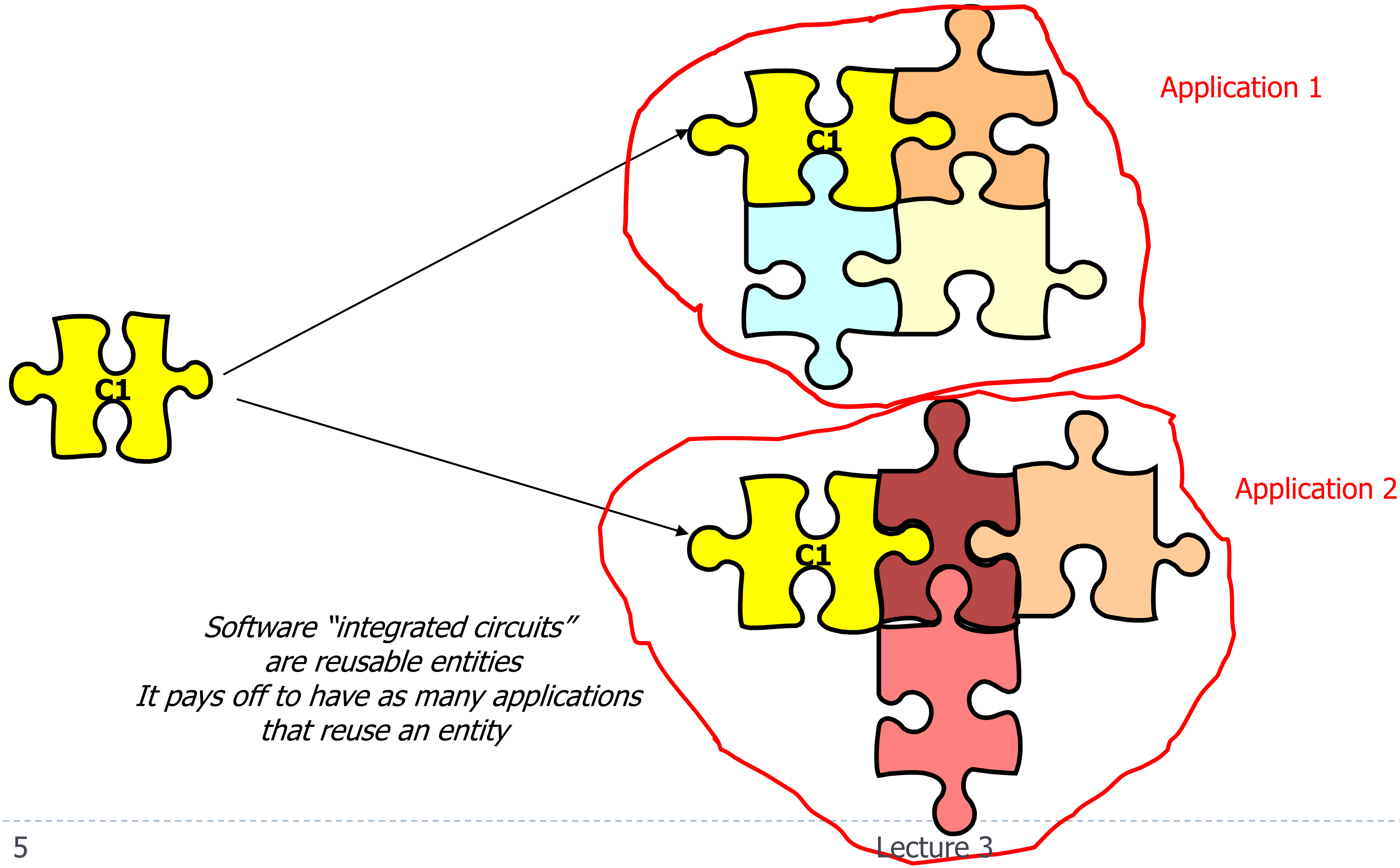
- Component-based software engineering (CBSE) is an approach to software development that relies on software reuse – reusing *artifacts* (*software parts*)
- Advantages of CBSE:
  - **Reuse**: Development of system = assembly of component
  - **Flexibility**: Maintenance,upgrading=customization, replacement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

Advantage 1: Software construction



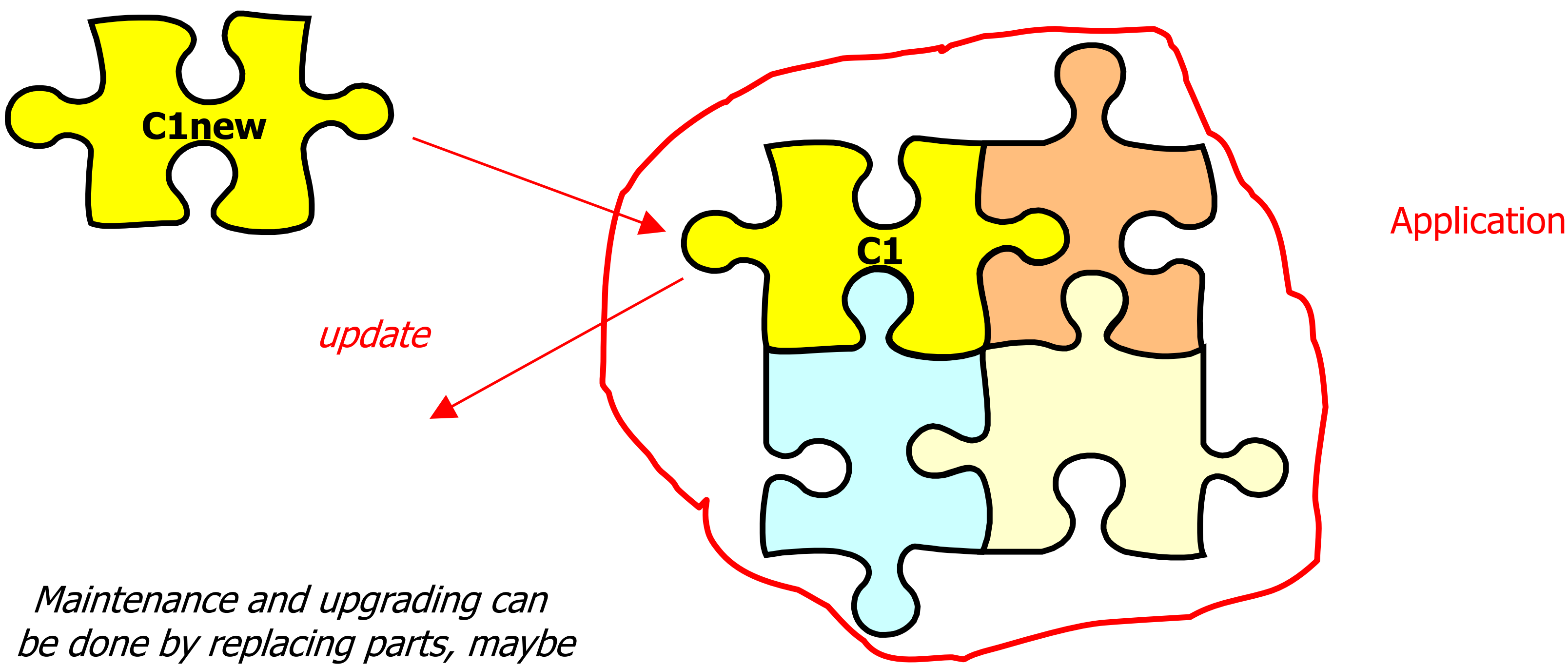
Software construction vs. creation: application is developed as an assembly of "integrated circuits"

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 Produced Software Components*"

1970

1980

1990

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

2000

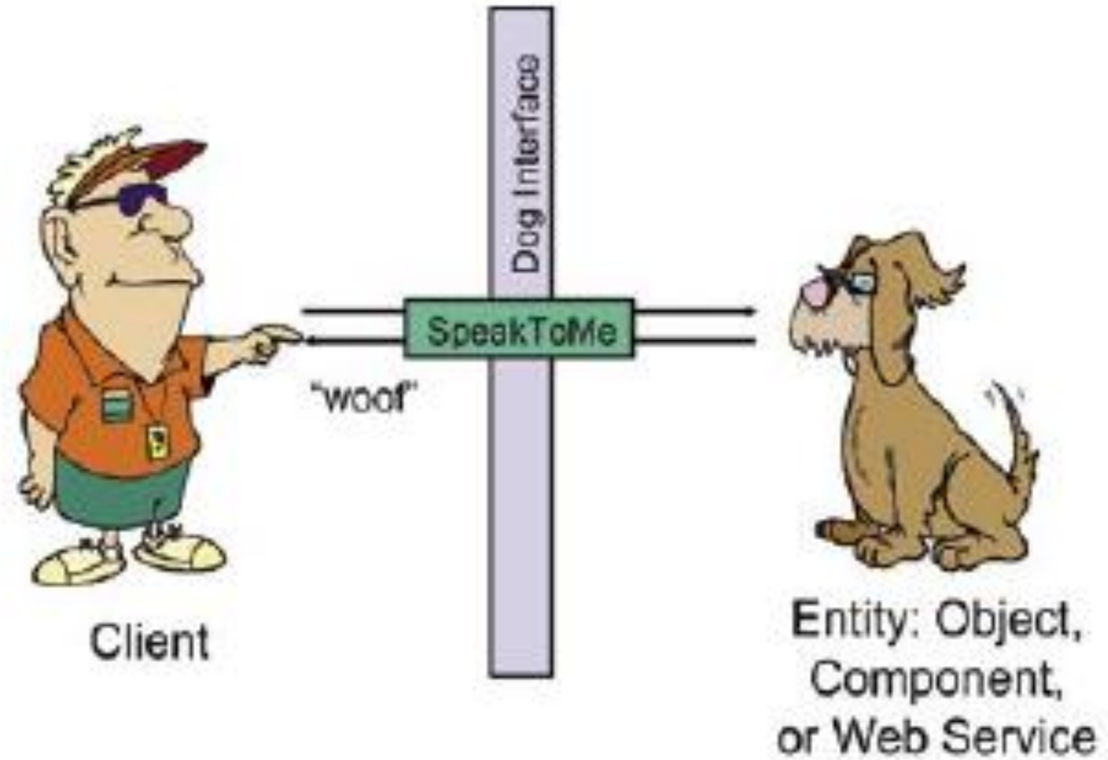
2010

Principles for reuse by composition

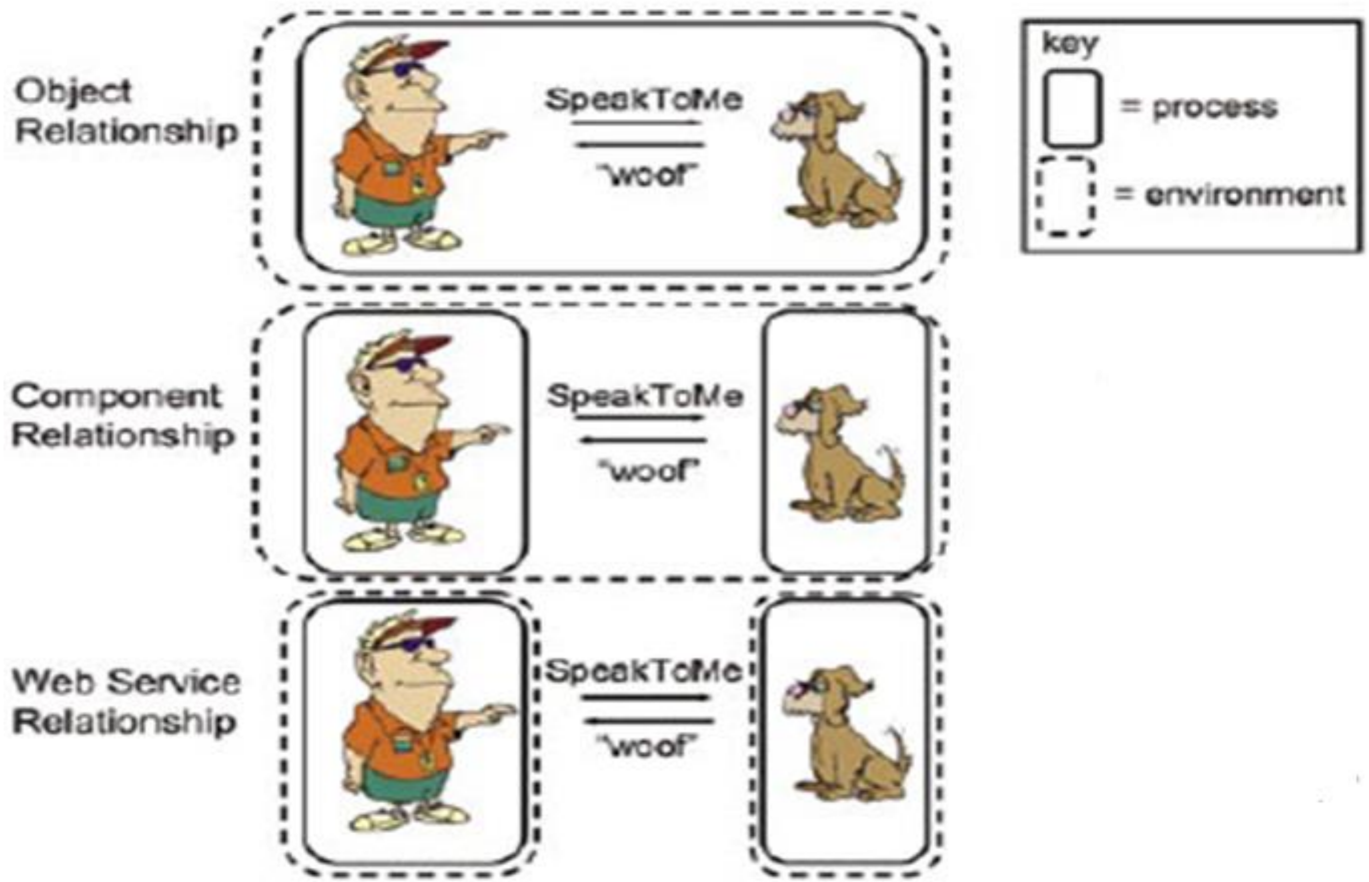
- Key requirements for Black-Box reuse:
  - **Abstraction**: an "Entity" is known by its "interface"
  - **Encapsulation**: the "insides" of an "Entity" are not exposed to the outside

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."



Reusable Entities by Location and Environment



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

Objects-Components-Services

<table>
<tr><th colspan="3">Entities for Reuse and Composition</th></tr>
<tr><td colspan="3">•Abstraction
•Encapsulation</td></tr>
<tr><th>Objects</th><th>Components</th><th>Services</th></tr>
<tr><td>•Location: same process
•Inheritance
•Polymorphism</td><td>•Location: different processes, same environment
•Usually some runtime infrastructure needed
•No state
•No shared variables</td><td>•Location: different environments
•More emphasis on interface/contract/service agreement
•Mechanisms for dynamic discovery
•Dynamically composable</td></tr>
</table>

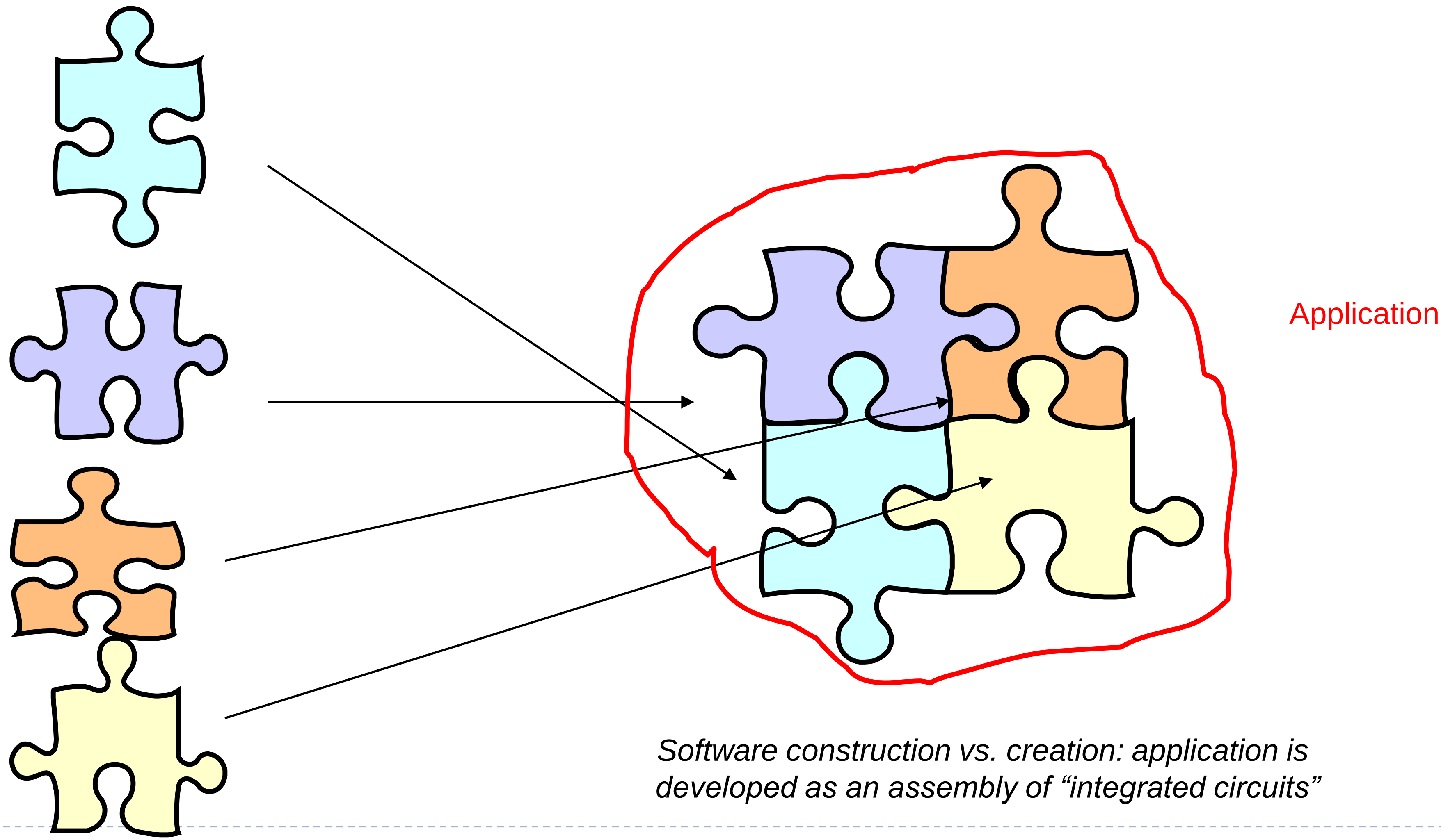
Reusable Entities
made more usable and more composable

- 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

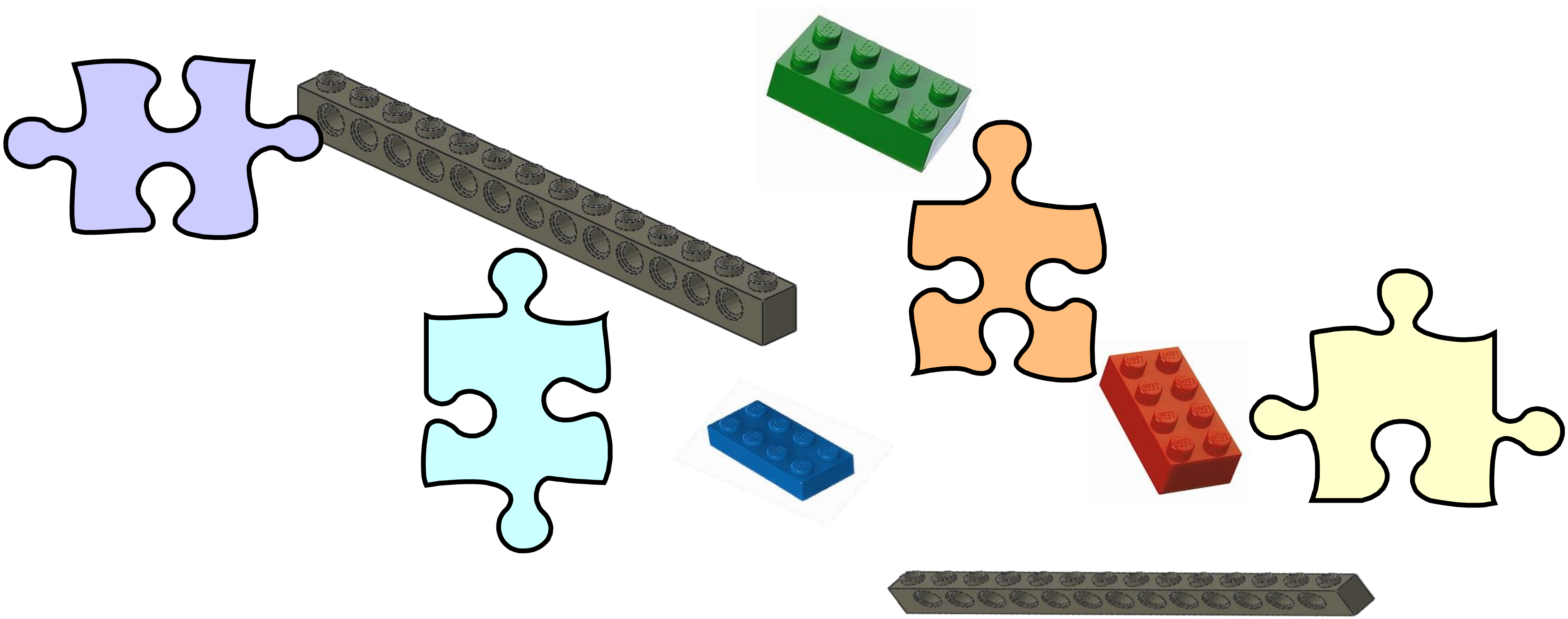
- **Component Based Software Engineering (CBSE) = reuse of:**
  - **Parts (components)**
  - **Infrastructure**

Component based software construction – the ideal case



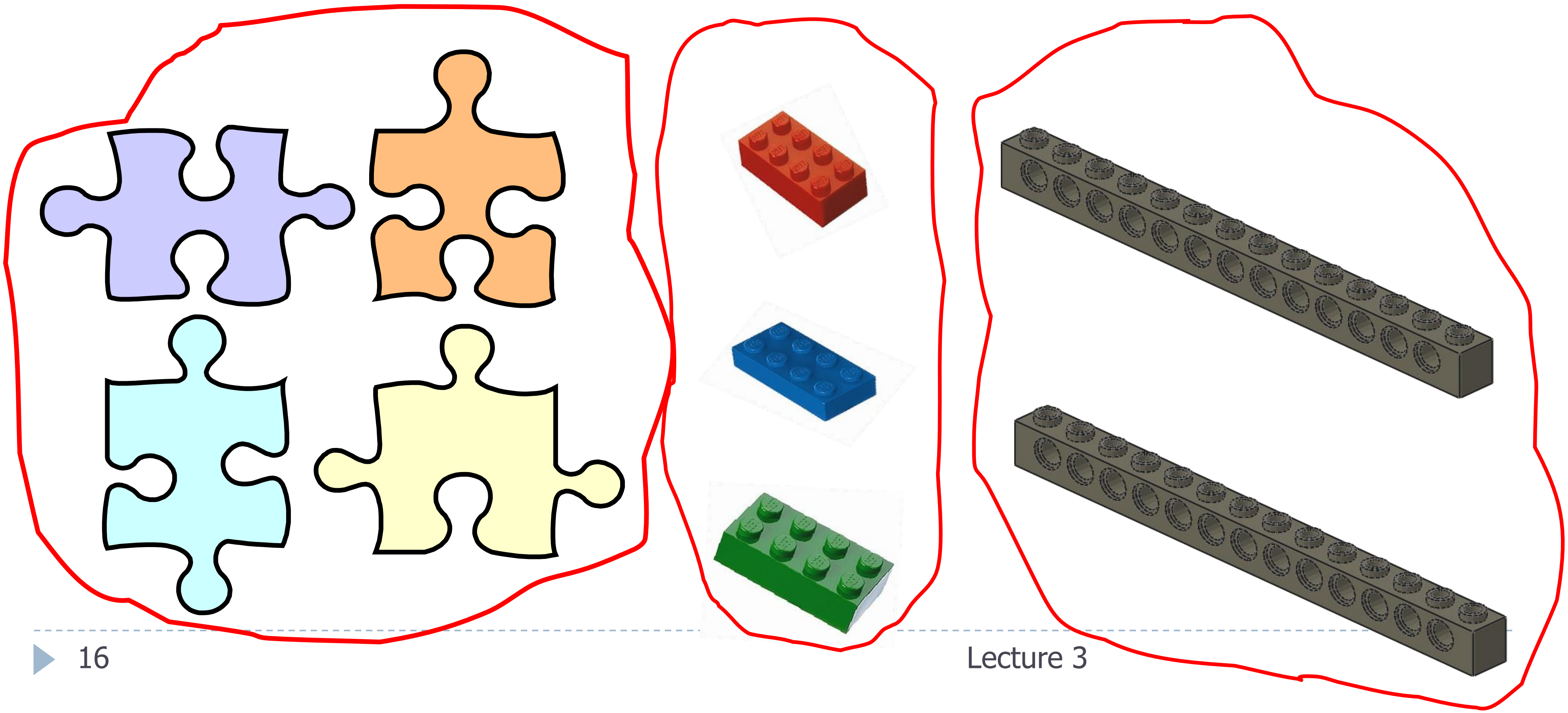
Software construction vs. creation: application is developed as an assembly of "integrated circuits"

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 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."

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

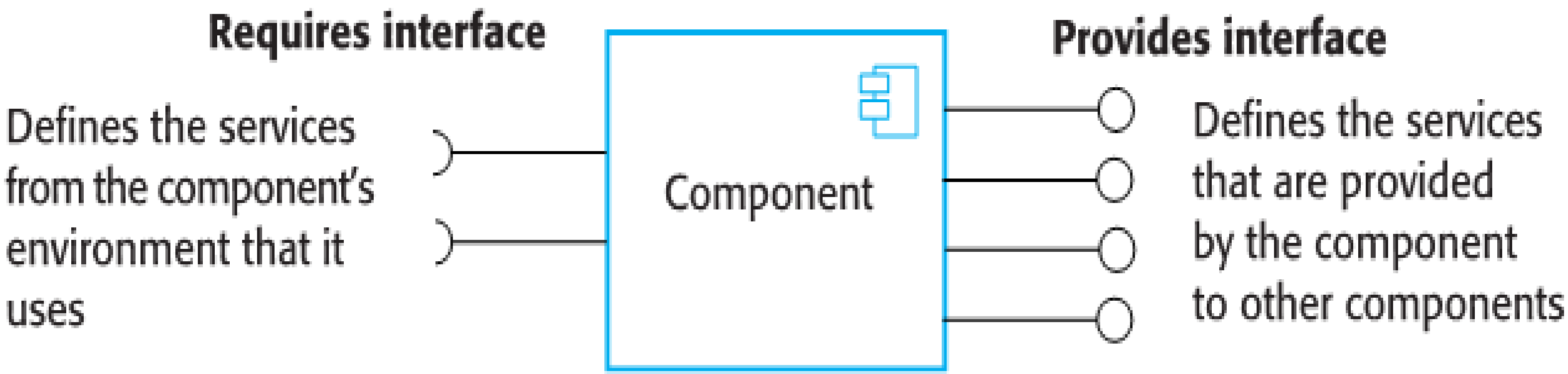


Fig. 19.2 from [Sommerville]

Example: A data collector component

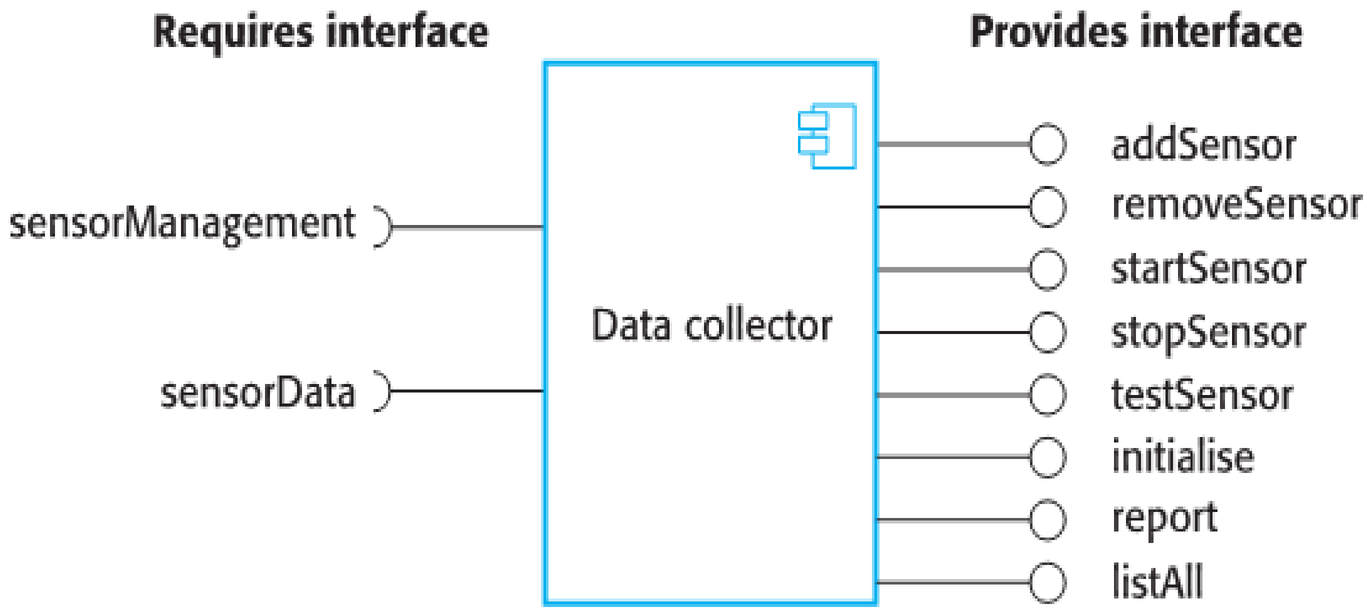


Fig. 19.3 from [Sommerville]

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

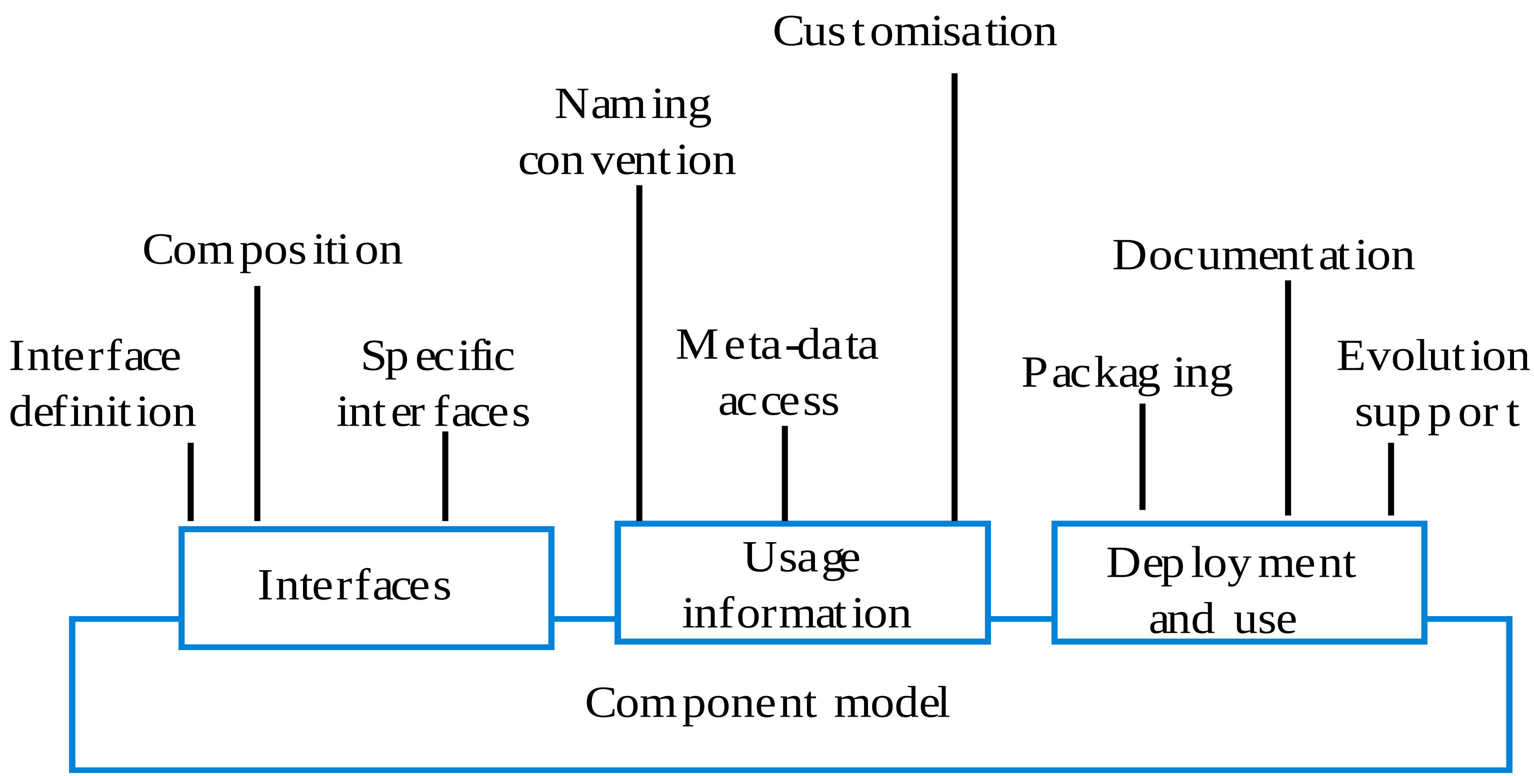


 Fig. 19.4 from [Sommerville]

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 component model infrastructure, components are deployed in a container. This is a set of interfaces used to access the service implementations.

Component model services

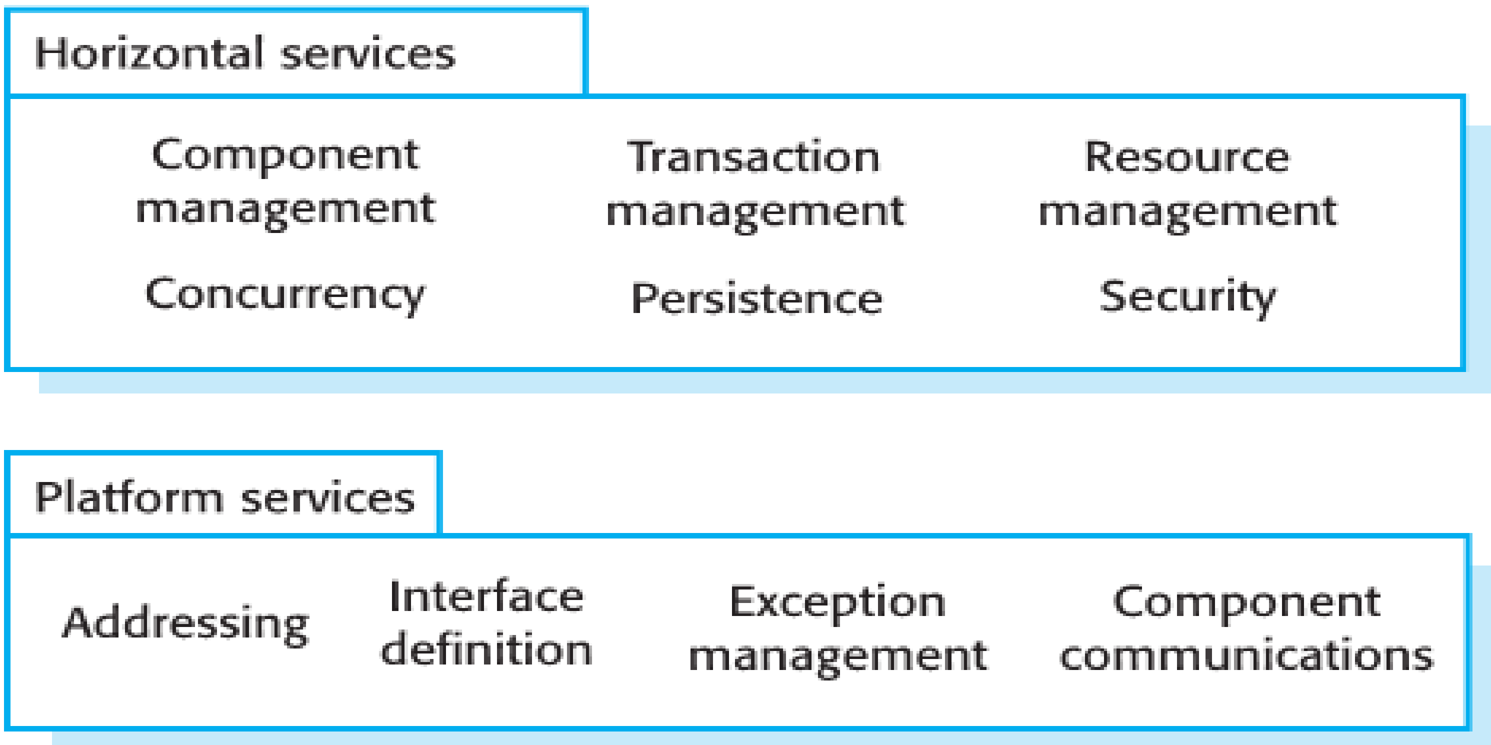


Fig. 19.5 from [Sommerville]

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 (component models)
  - Middleware (component framework)

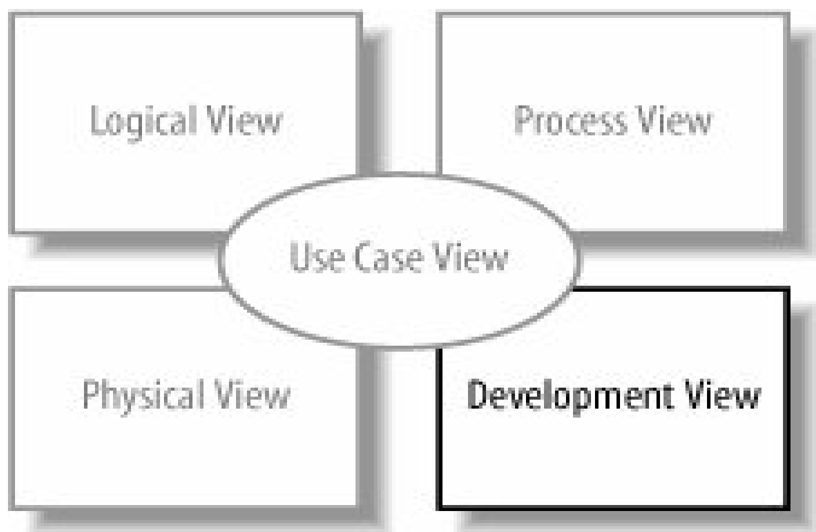
Modelling components in UML

Main References

- Modelling components in UML
  - Main text:
    - Kim Hamilton, Russell Miles, *Learning UML 2.0*, OReilly, 2006 , chapter 12 (*Managing and Reusing Your System's Parts: Component Diagrams*)
  - Additional readings:
    - *Documenting Component and Connector Views with UML 2.0*, Technical Report CMU-SEI-2004-TR-008, http://www.sei.cmu.edu/library/abstracts/reports/04tr008.cfm
    - Kim Hamilton, Russell Miles, *Learning UML 2.0*, OReilly, 2006 , chapter 11 (*Modeling a Class's Internal Structure: Composite Structures*)

UML Components

- Components are used in UML to organize a system into manageable, reusable, and swappable pieces of software.
- UML **component diagrams** model the components in your system and as such form part of the development view
- The **development view** describes how your system's parts are organized into modules and components and is great at helping you manage layers within your system's architecture.



12-1. The Development View of your model describes how your system's parts are organized into modules and components

UML Components

- In UML, a component can do:
  - the same things a class can do: generalize and associate with other classes and components, implement interfaces, have operations, and so on.
  - Furthermore, as with composite structures, components can have **ports** and show **internal structure**. It's common for a component to contain and use other classes or components to do its job.
  - To promote loose coupling and encapsulation, components are accessed through **interfaces**.

UML notation for components

- A component is drawn as a rectangle with the <<component>> stereotype and an optional tabbed rectangle icon in the upper righthand corner.

- In earlier versions of UML, the component symbol was a larger version of the tabbed rectangle icon

- You can show that a component is actually a subsystem of a very large system by replacing <<component>> with <<subsystem>>

Fig. 12.2 and 12.3 from [UML2]

Provided and required interfaces

- Components need to be loosely coupled so that they can be changed without forcing changes on other parts of the system.
- Components interact with each other through provided and required *interfaces* to control dependencies between components and to make components swappable.
- A *provided interface* of a component is an interface that the component realizes. Other components and classes interact with a component through its provided interfaces . A component's provided interface describes the services provided by the component.
- A *required interface* of a component is an interface that the component needs to function. More precisely, the component needs another class or component that realizes that interface to function. But to stick with the goal of loose coupling, it accesses the class or component through the required interface. A required interface declares the services a component will need.

UML notation for provided and required interfaces

- There are three standard ways to show provided and required interfaces in UML:
  - ball and socket symbols
  - stereotype notation
  - text listings.

Ball and socket notation for interfaces

Fig. 12.4 from [UML2]

Stereotype notation for interfaces

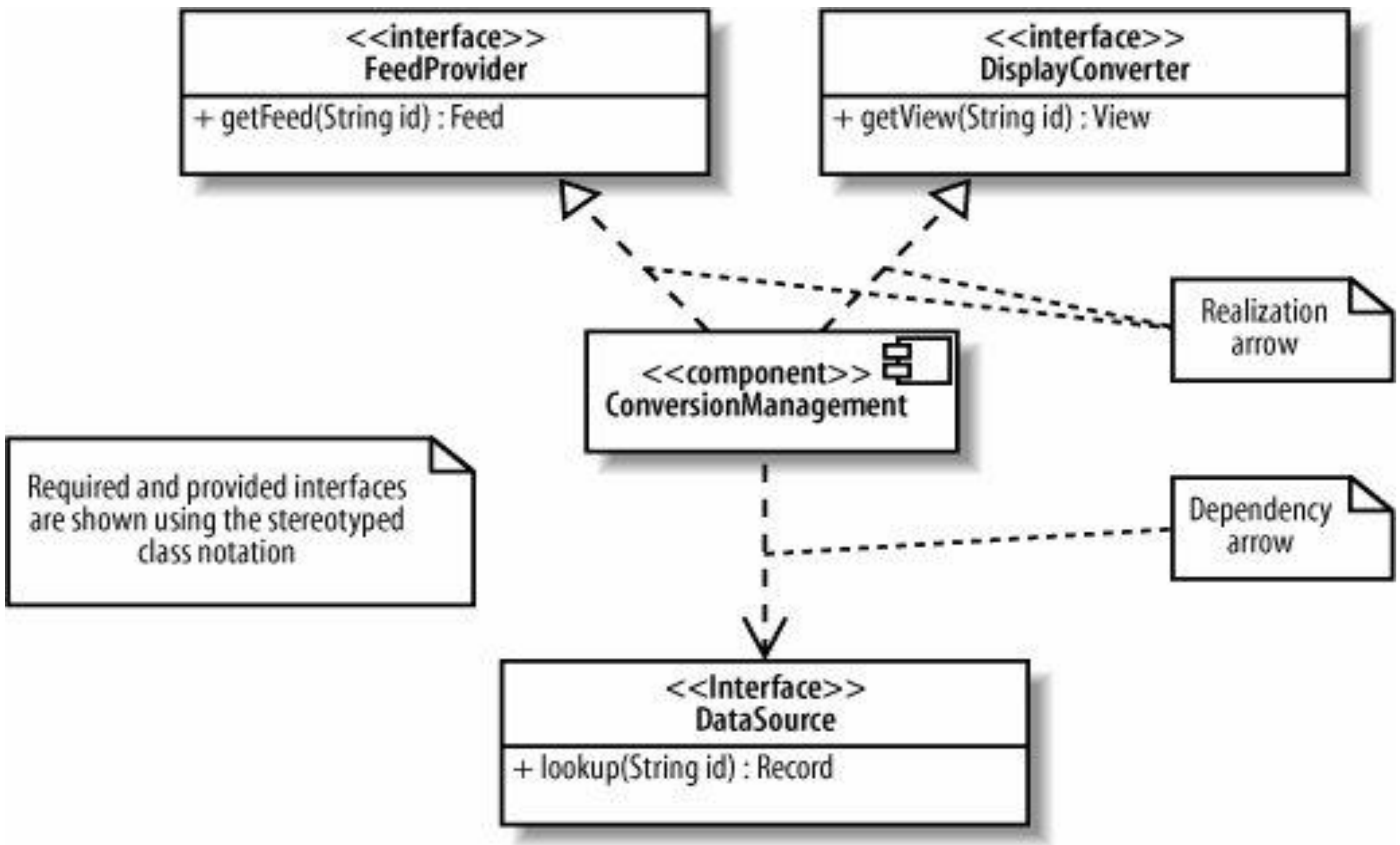
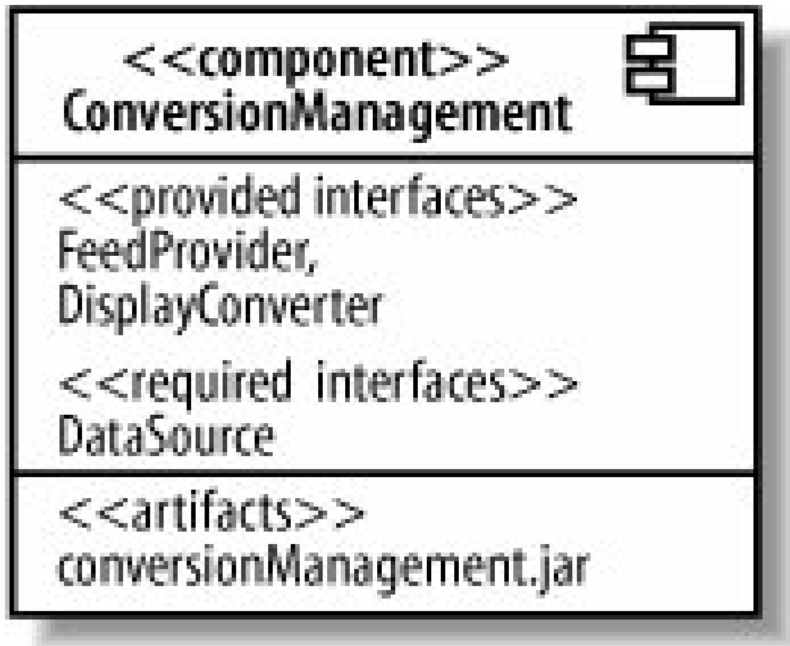


Fig. 12.5 from [UML2]

Listing component interfaces

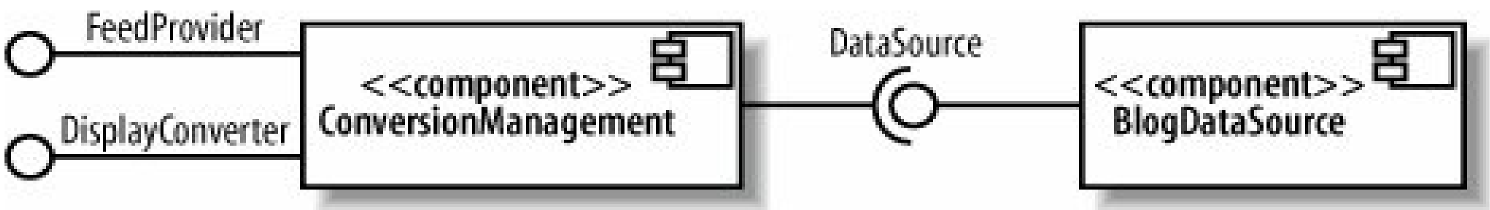


This notation additionaly has an <<artifacts>> section listing the artifacts or physical files manifesting this component

Fig. 12.6 from [UML2]

Showing components working together

If a component has a required interface, then it needs another class or component in the system that provides it.



At a higher level view, this is a dependency relation between the components

Fig. 12.7, 12.8 and 12.9 from [UML2]

Example- component diagram presents system architecture

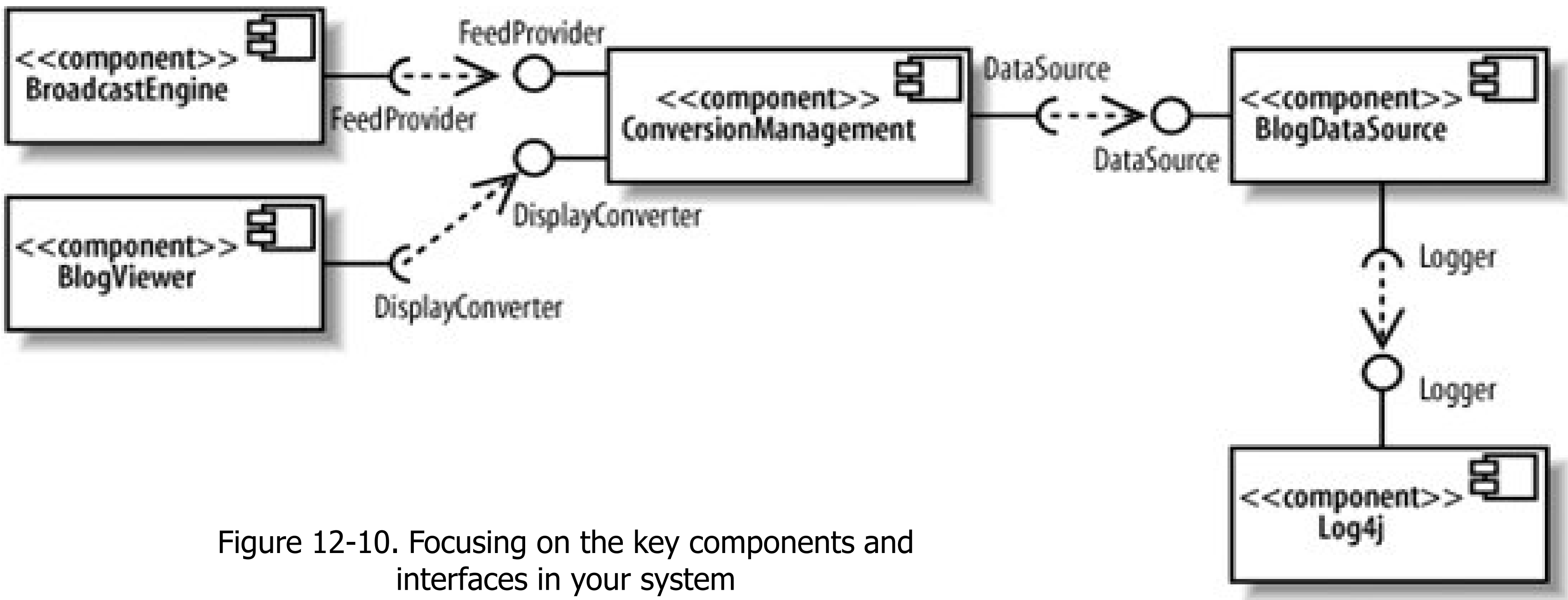


Figure 12-10. Focusing on the key components and interfaces in your system

Example- component diagram presents system architecture

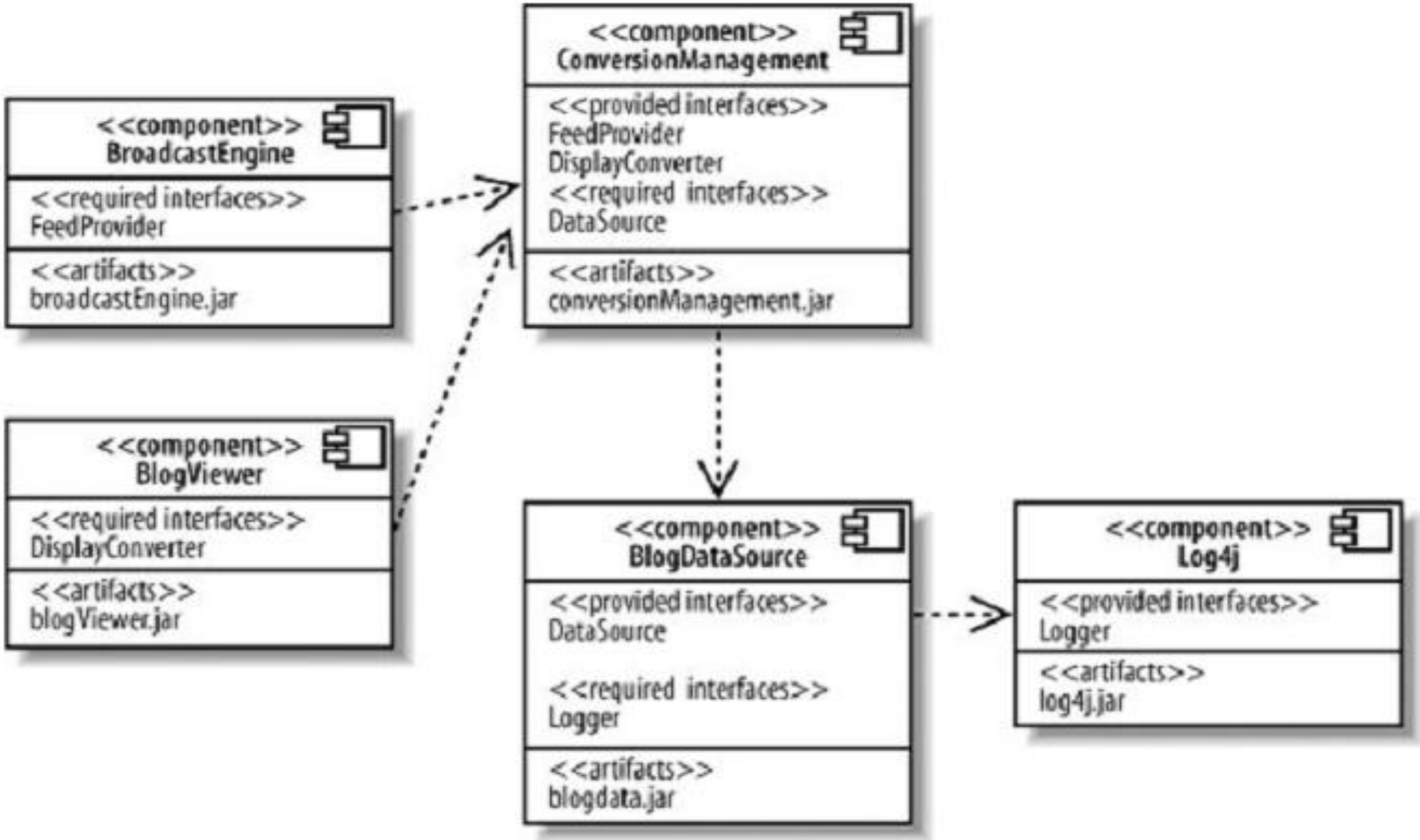
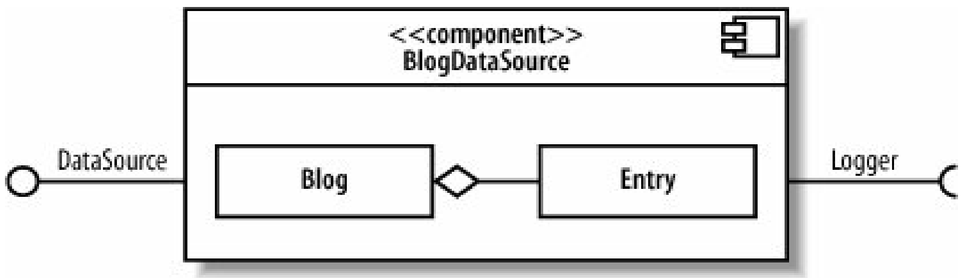
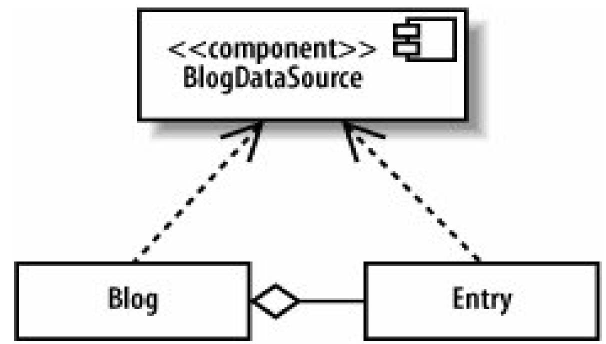


Figure 12-11. Focusing on component dependencies and the manifesting artifacts is useful when you are trying control the configuration or deployment of your system

Classes that realize a component

A component often contains and uses other classes to implement its functionality.
Such classes are said to *realize* a component.
There are 3 ways to depict this:





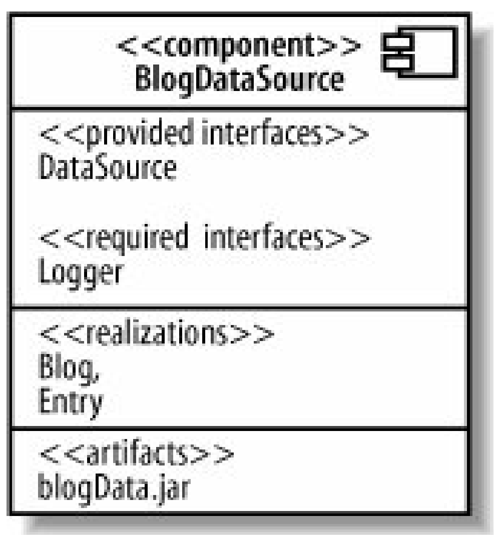


Fig. 12.12 , 12.13, 12.14 from [UML2]

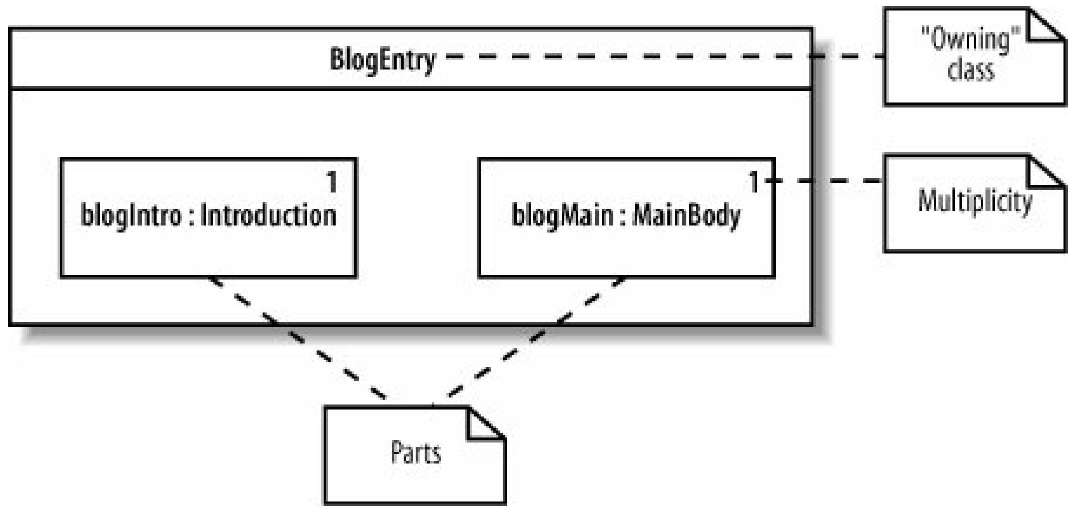
Ports and internal structure

- There is heavy overlap between certain topics in component diagrams and composite structures. The ability to have ports and internal structure is defined for classes in composite structures. Components inherit this capability and introduce some of their own features, such as delegation and assembly connectors.
- The topics of a class's internal structure and ports in the context of composite structures are presented here first (based on Chapter 11 from [UML2] .

Composite structures

- Composite structures show:
  - Internal structures
    - Show the parts contained by a class and the relationships between the parts; this allows you to show context-sensitive relationships, or relationships that hold in the context of a containing class
  - Ports
    - Show how a class is used on your system with ports
  - Collaborations
    - Show design patterns in your software and, more generally, objects cooperating to achieve a goal
- Composite structures provide a view of your system's parts and form part of the *logical view* of your system's model

Parts of a class

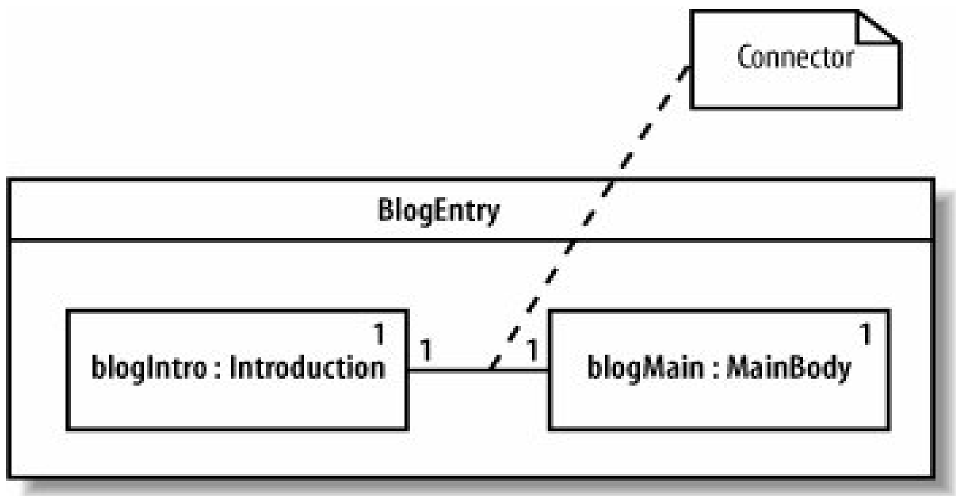


When showing the internal structure of a class, you draw its parts, or items contained by composition, inside the containing class. Parts are specified by the role they play in the containing class

A part is a set of instances that may exist in an instance of the containing class at runtime

Fig. 11.6 from [UML2]

Connectors



Relationships between parts are shown by drawing a connector between them. A connector is a link that enables communication between parts: it means that runtime instances of the parts can communicate

Fig. 11.9 from [UML2]

Ports

A port is a point of interaction between a class and the outside world. It represents a distinct way of using a class, usually by different types of clients.

For example, a Wiki class could have two distinct uses:

•Allowing users to view and edit the Wiki

•Providing maintenance utilities to administrators who want to perform actions such as rolling back the Wiki if incorrect content is provided

Each distinct use of a class is represented with a port, drawn as a small rectangle on the boundary of the class

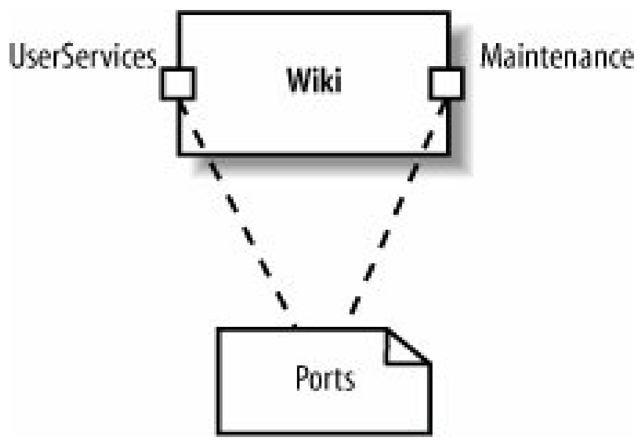


Fig. 11.14 from [UML2]

It's common for classes to have interfaces associated with ports.
You can use ports to group related interfaces to show the services available at that port.

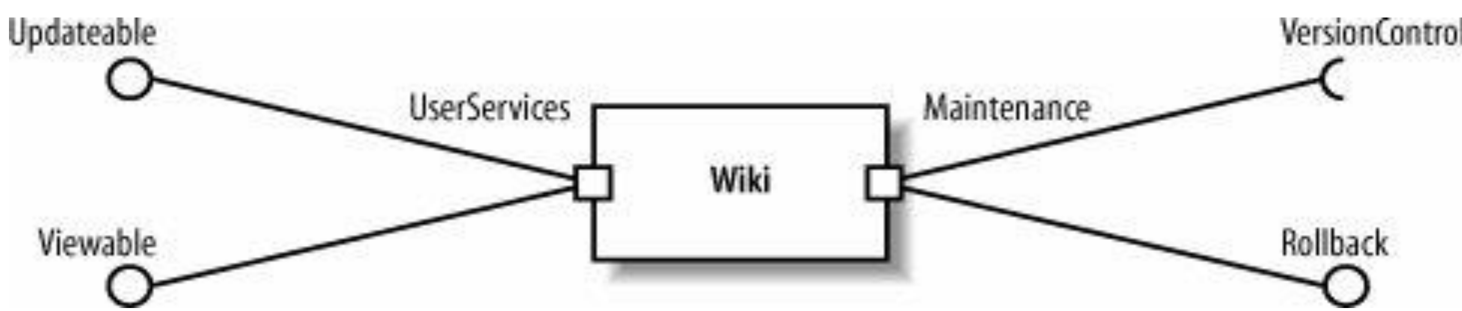
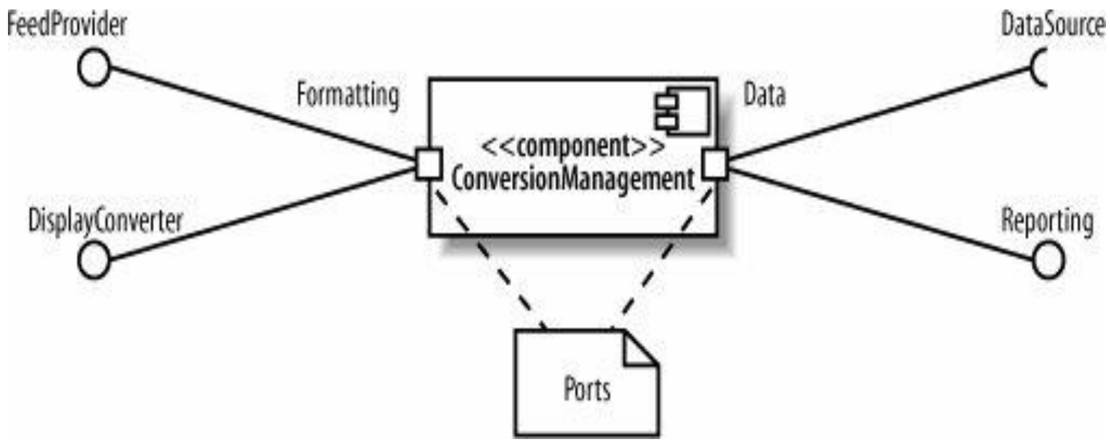


Fig. 11.15 from [UML2]

Ports and internal structure of components



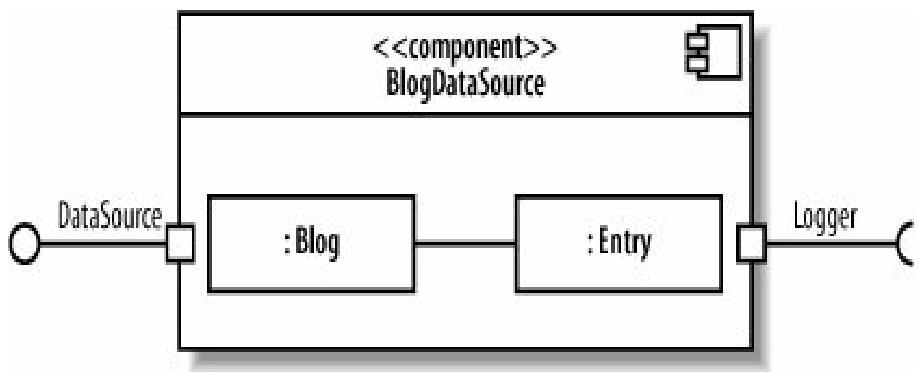
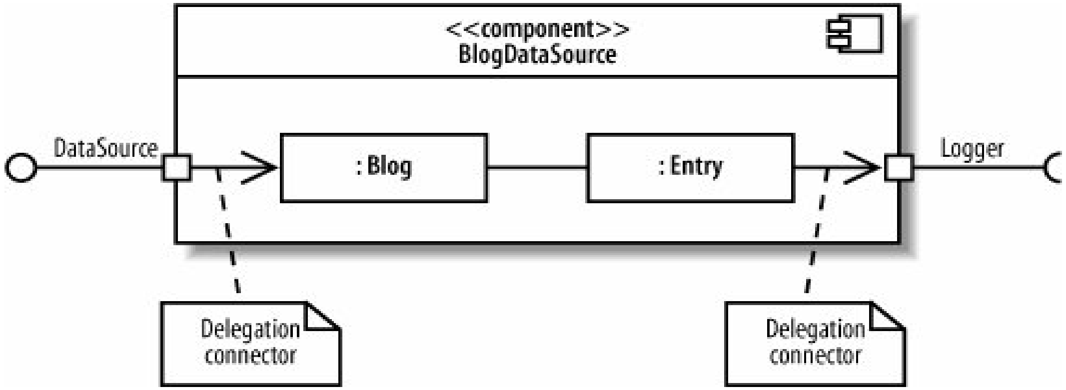


Fig. 12.15 and 12.16 from [UML2]

Delegation connectors



Delegation connectors show how interfaces correspond to internal parts
Delegation connectors can also connect interfaces of internal parts with ports

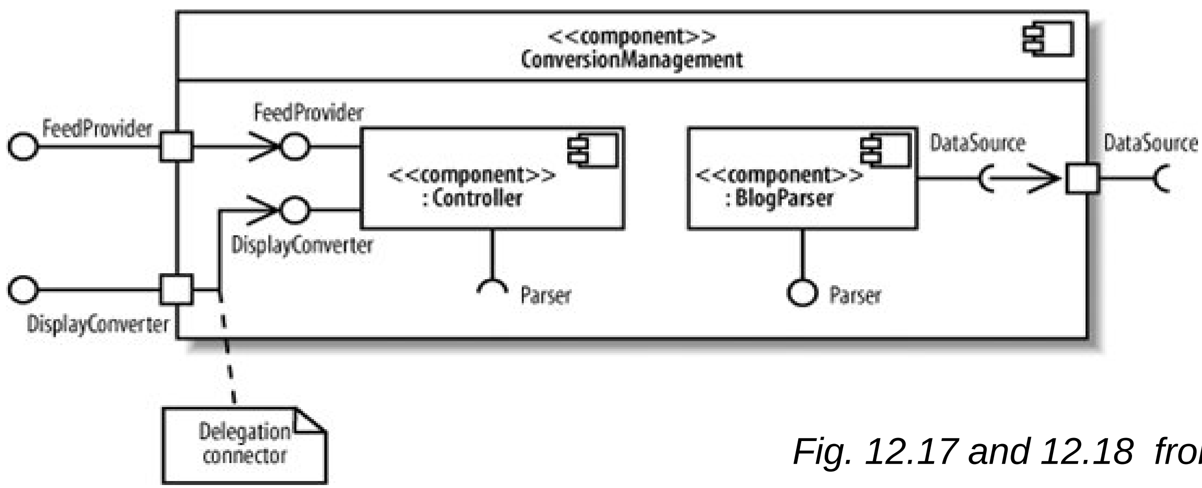
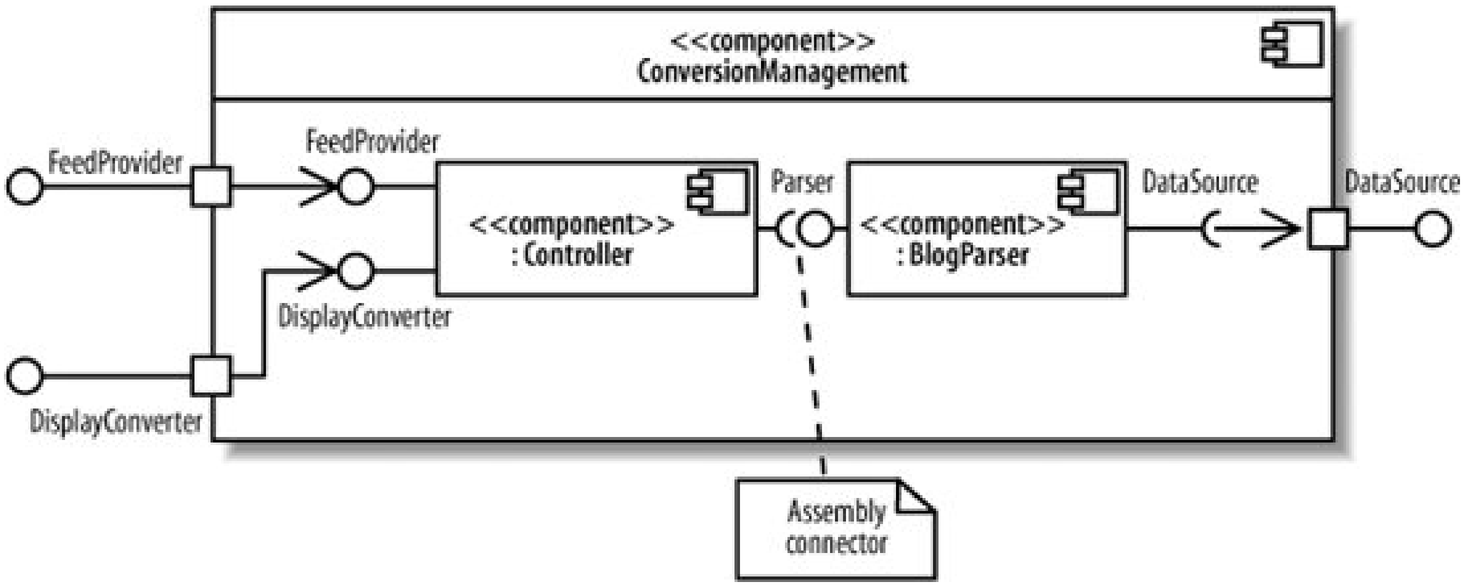


Fig. 12.17 and 12.18 from [UML2]

Assembly connectors



Assembly connectors show that a component requires an interface that another component provides
Assembly connectors are used when showing composite structure of components

Fig. 12.19 from [UML2]

Black-box and white-box views

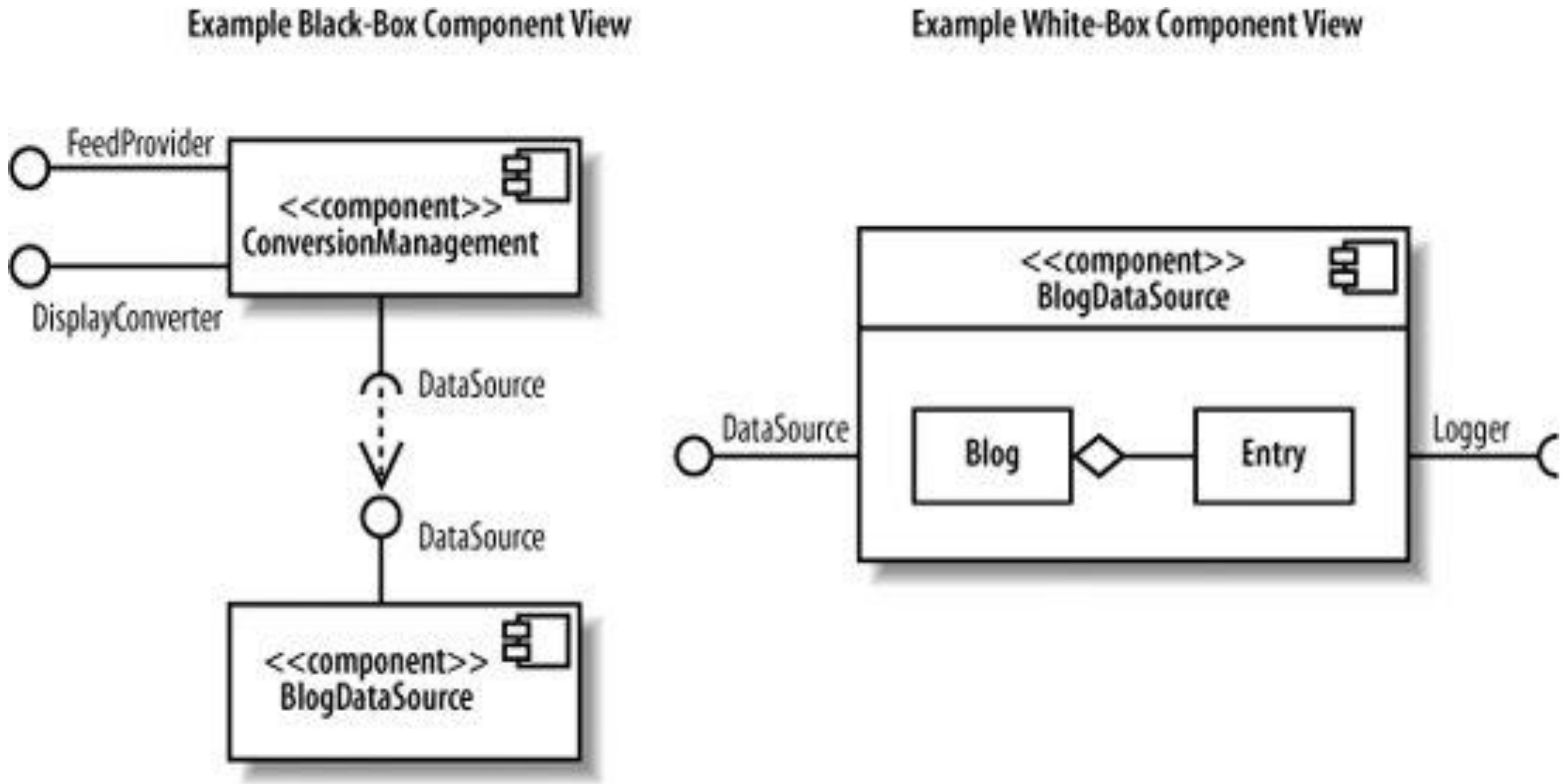


Fig. 12.20 from [UML2]

UML Tools

- Wikipedia List of UML tools
- http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

- StarUML (free StarUML1 version, free StarUML2 Beta version)
- UMLet (free simple UML drawing tool, includes component diagrams)
- MS Visio (60 days trial version)
- IBM Rational Software Architect (30 days trial)

- Questions?