12. Managing and Reusing Your System's Parts: Component Diagrams

Shaoning Zeng, http://zsn.cc

What we learnt?

- 11. Modeling a Class's Internal Structure: Composite Structures 组合结构
  - 11.1. Internal Structure 内部结构
  - 11.2. Showing How a Class Is Used 类的用法
  - 11.3. Showing Patterns with Collaborations

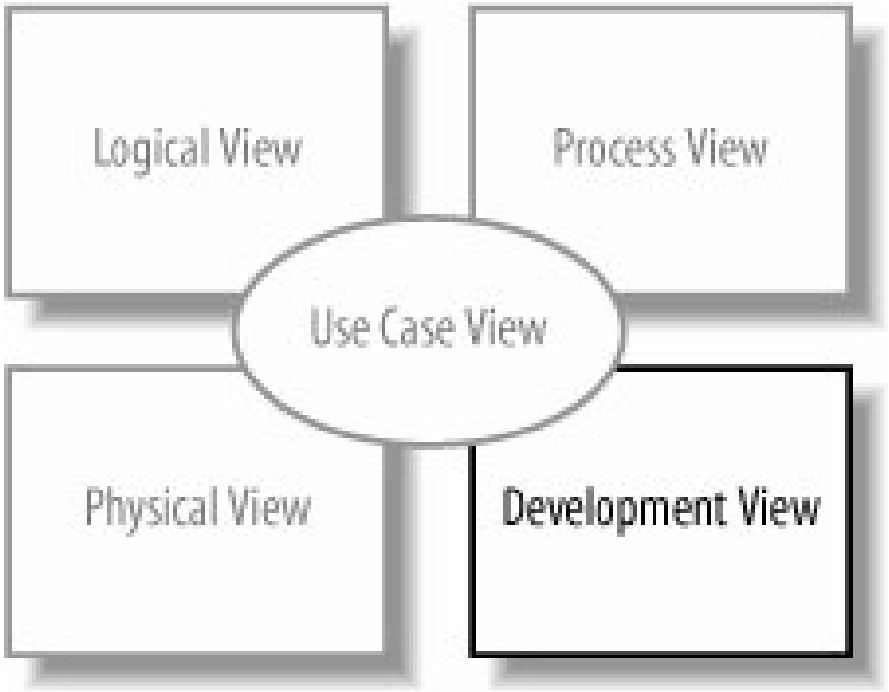
12. Managing and Reusing Your System's Parts: Component Diagrams 组件图

- 12.1. What Is a Component? 组件的概念
- 12.2. A Basic Component in UML
- 12.3. Provided and Required Interfaces of a Component
- 12.4. Showing Components Working Together
- 12.5. Classes That Realize a Component 实现组件的类
- 12.6. Ports and Internal Structure 端口与内部结构
- 12.7. Black-Box and White-Box Component Views

12. Managing and Reusing Your System's Parts: Component Diagrams

- With all but the most trivial systems, it's helpful to plan out the high-level pieces of your system to establish the architecture and manage complexity and dependencies among the parts. 体系架构
  - Components are used to organize a system into manageable, reusable, and swappable pieces of software.
    可管理|可重用|可替换
- UML component diagrams model the components in your system and as such form part of the development view , as shown in Figure 12-1. 开发视图
  - 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.
    系统各部分如何构成模块与组件 – 系统架构

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



12.1. What Is a Component?

- A *component* is an encapsulated, reusable, and replaceable part of your software. 组件是软件中封装好的可重用可替换部件
- You can think of components as building blocks: you combine them to fit together (possibly building successively larger components) to form your software. 构建块
  - Because of this, components can range in size from relatively small, about the size of a class, up to a large subsystem.
- Good candidates for components are items that perform a key functionality and will be used frequently throughout your system. 候选：关键功能+频繁使用
  - Software, such as loggers, XML parsers, or online shopping carts, are components you may already be using.
  - These happen to be examples of common third-party components, but the same principles apply to components you create yourself.

Component in practical

- In your own system, you might create a component that provides services or access to data. 服务|数据访问
  - For example, in a CMS you could have a conversion management component that converts blogs to different formats, such as RSS feeds. RSS feeds are commonly used to provide XML-formatted updates to online content (such as blogs). CMS中将内容转换为其他格式的组件

Component in UML

- 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. 与类相同
- Furthermore, as with composite structures (see Chapter 11), they can have ports and show internal structure.
  可以有端口和显示内部结构
- The main difference between a class and a component is that a component generally has bigger responsibilities than a class.
  - For example, you might create a user information class that contains a user's contact information (her name and email address) and a user management component that allows user accounts to be created and checked for authenticity. 用户信息类 vs 用户管理组件
  - Furthermore, it's common for a component to contain and use other classes or components to do its job. 组件可能包含类或组件

Loosely coupled 宽松耦合

- Since components are major players in your software design, it's important that they are loosely coupled so that changes to a component do not affect the rest of your system.
- To promote loose coupling and encapsulation, components are accessed through interfaces. 通过接口访问
  - Recall from Chapter 5 that interfaces separate a behavior from its implementation.
  - By allowing components to access each other through interfaces, you can reduce the chance that a change in one component will cause a ripple of breaks throughout your system.

12.2. A Basic Component in UML

Figure 12-2. The basic component symbol showing a ConversionManagement component

Figure 12-3. You can substitute the <<subsystem>> stereotype to show the largest pieces of your system

12.3. Provided and Required Interfaces of a Component 两种接口

- Components need to be loosely coupled so that they can be changed without forcing changes on other parts of the system, this is where interfaces come in.
  - Components interact with each other through provided and required interfaces to control dependencies between components and to make components swappable.

Two types of interfaces of component

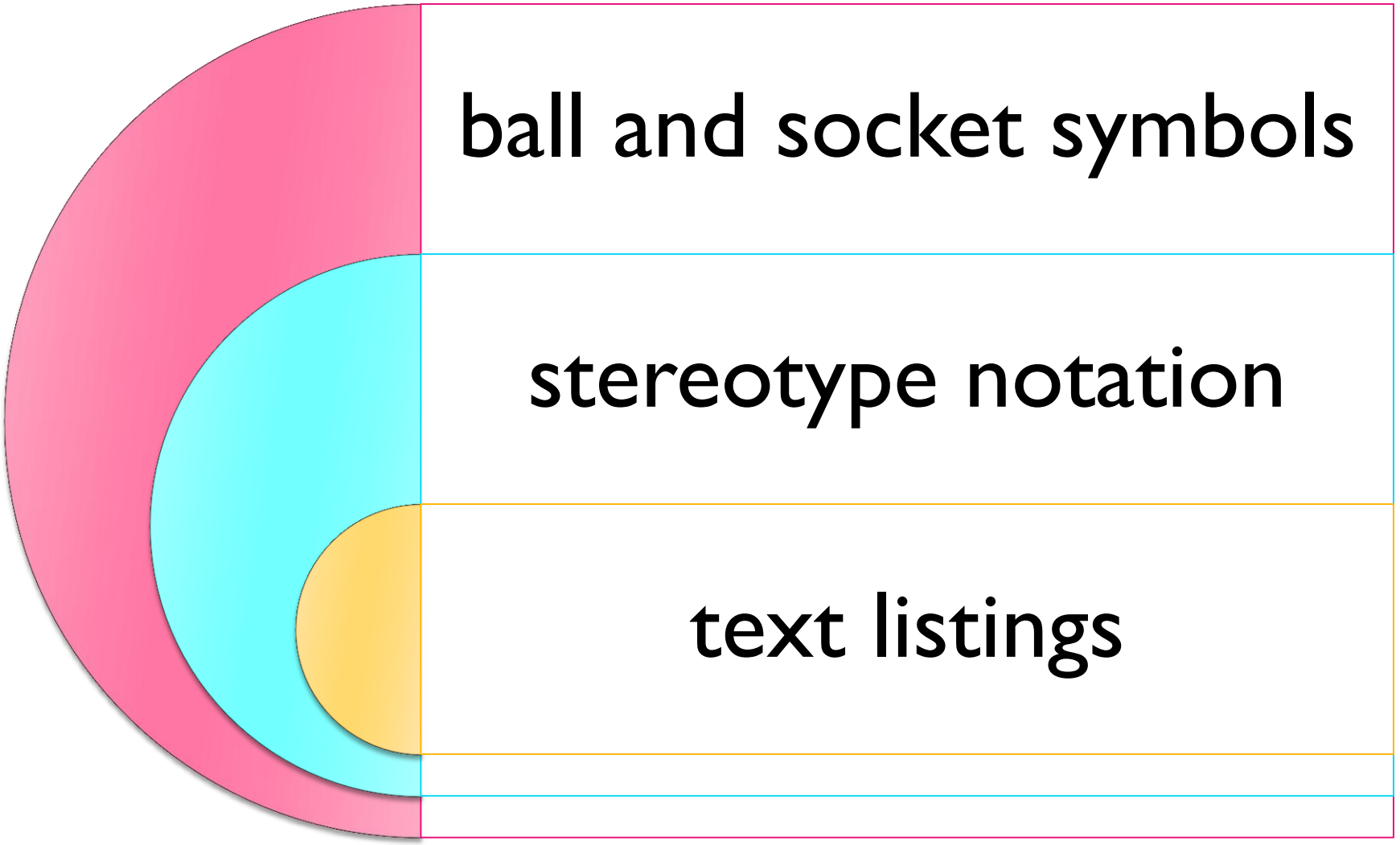
A ***provided interface*** of a component is an interface that the component realizes.

- Other components and classes interact with a component through its provided interfaces . 供其他组件使用
- A component's provided interface describes the services provided by the component. 组件提供的服务

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.

There are three standard ways to show provided and required interfaces in UML:



12.3.1. Ball and Socket Notation for Interfaces

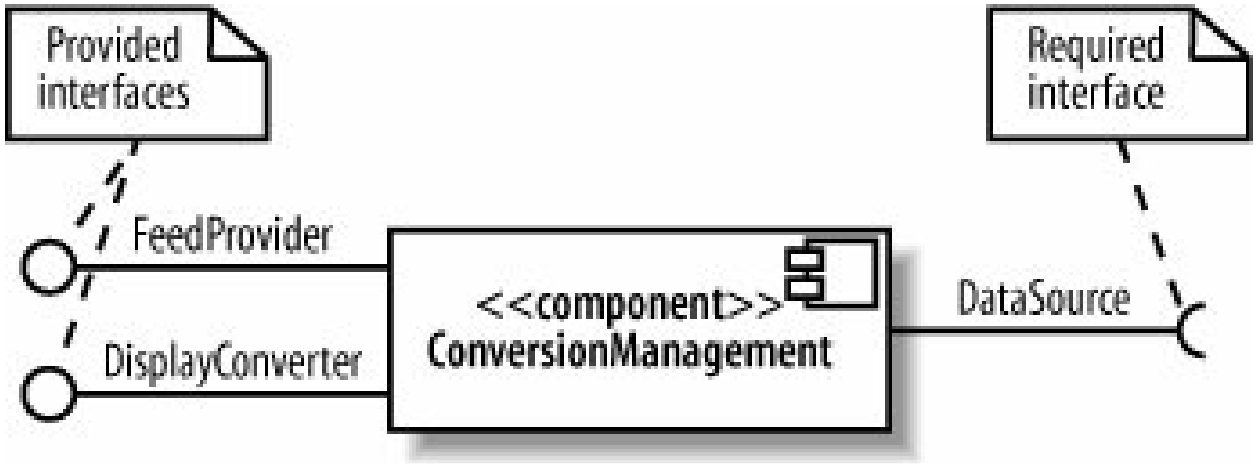


Figure 12-4. The ball and socket notation for showing a component's provided and required interfaces

12.3.2. Stereotype Notation for Interfaces

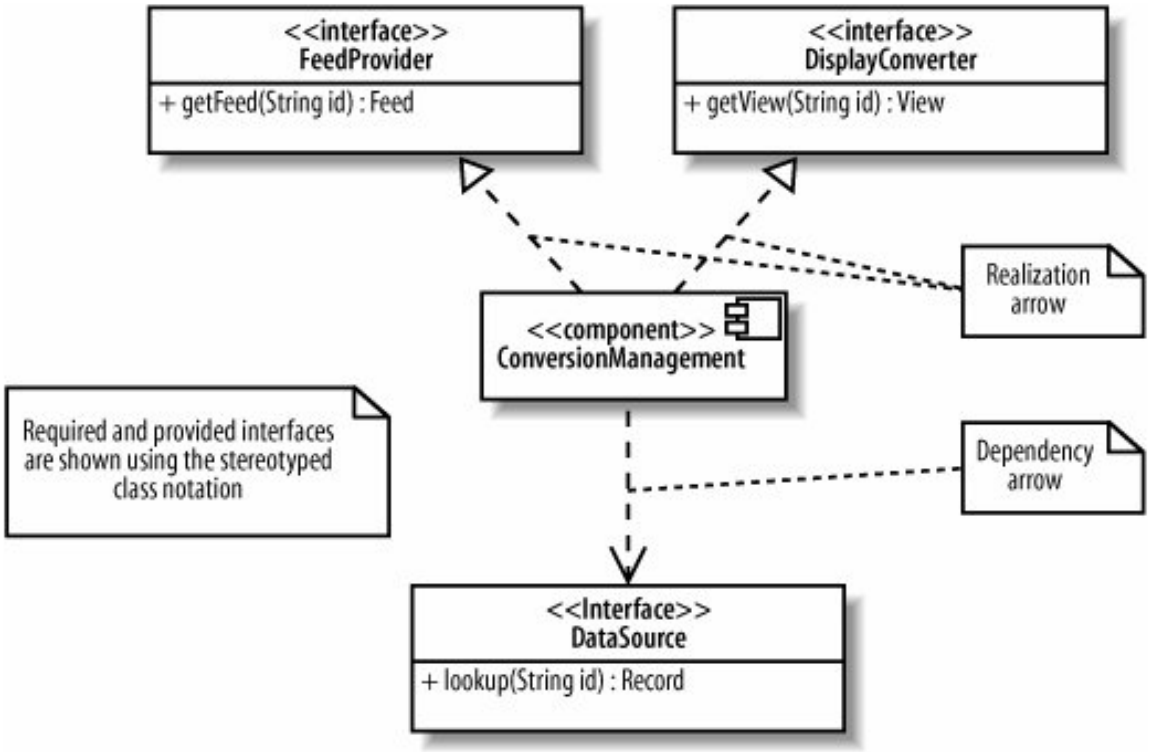


Figure 12-5. The stereotyped class notation, showing operations of the required and provided interfaces

12.3.3. Listing Component Interfaces

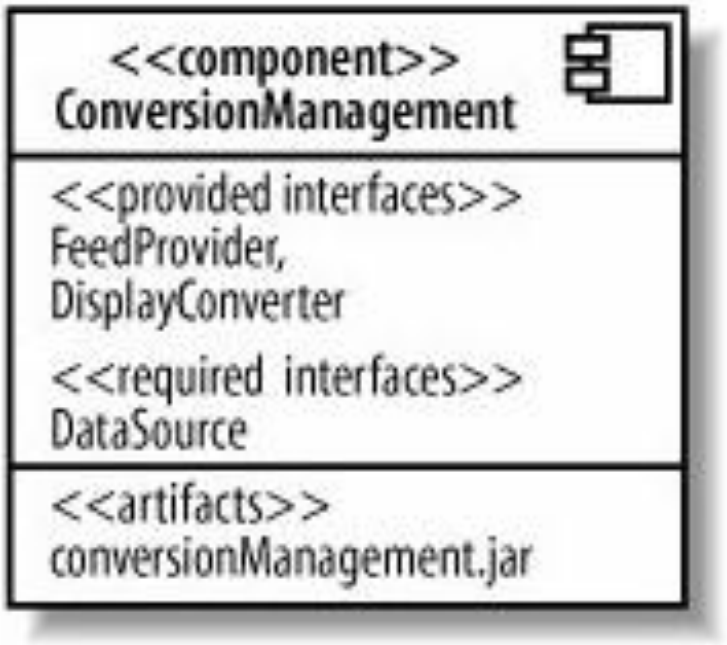


Figure 12-6. Listing required and provided interfaces within the component is the most compact representation

12.4. Showing Components Working Together

- If a component has a required interface, then it needs another class or component in the system to provide it.
- To show that a component with a required interface depends on another component that provides it, draw a dependency arrow from the dependent component's socket symbol to the providing component's ball symbol, as shown in Figure 12-7.

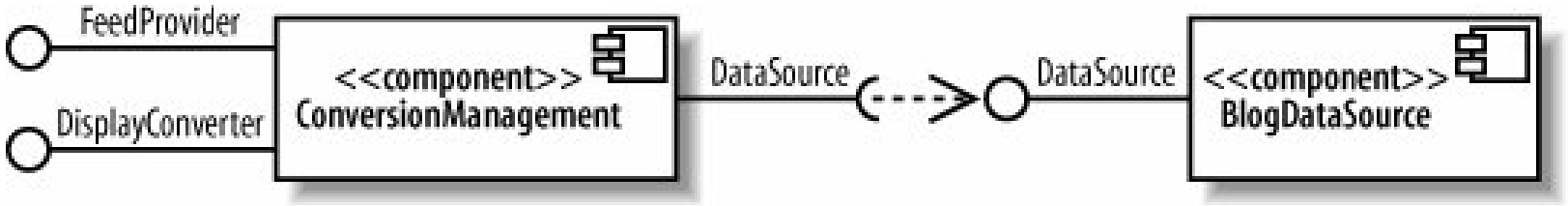


Figure 12-7. The ConversionManagement component requires the DataSource interface, and the BlogDataSource component provides that interface

12.4. Showing Components Working Together

- As a presentation option for Figure 12-7, your UML tool may let you get away with snapping the ball and socket together (omitting the dependency arrow), as shown in Figure 12-8.
- This is actually the assembly connector notation, which is introduced later in this chapter. 集合连接器

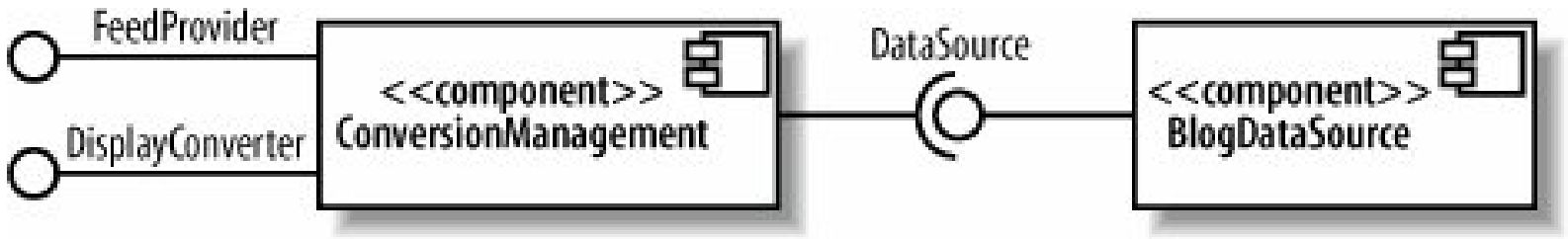


Figure 12-8. Presentation option that snaps the ball and socket together

12.4. Showing Components Working Together

- You can also omit the interface and draw the dependency relationship directly between the components, as shown in Figure 12-9. 简化：直接调用

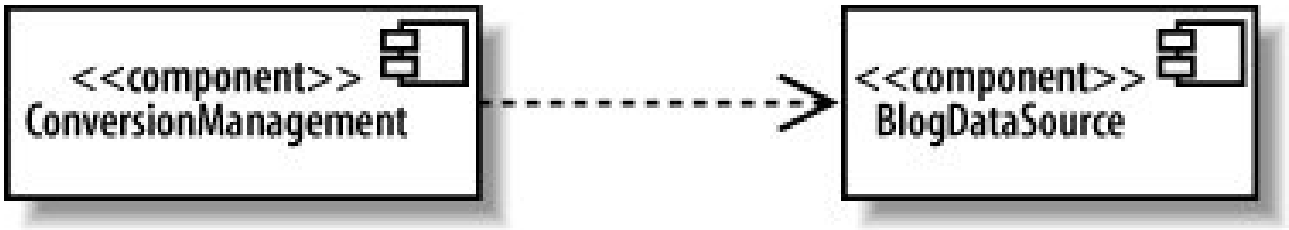


Figure 12-9.You can draw dependency arrows directly between components to show a higher level view

How to choose

- Remember that interfaces help components stay loosely coupled, so they are an important factor in your component architecture.
- Showing the key components in your system and their interconnections through interfaces is a great way to describe the architecture of your system, and this is what the first notation is good at, as shown in Figure 12-10.

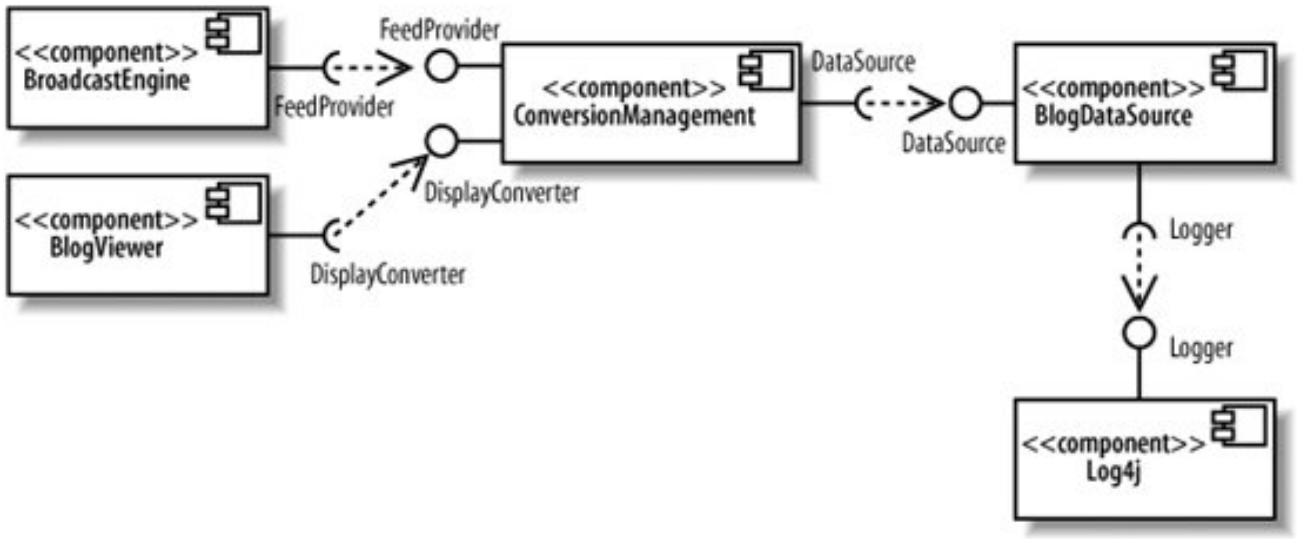


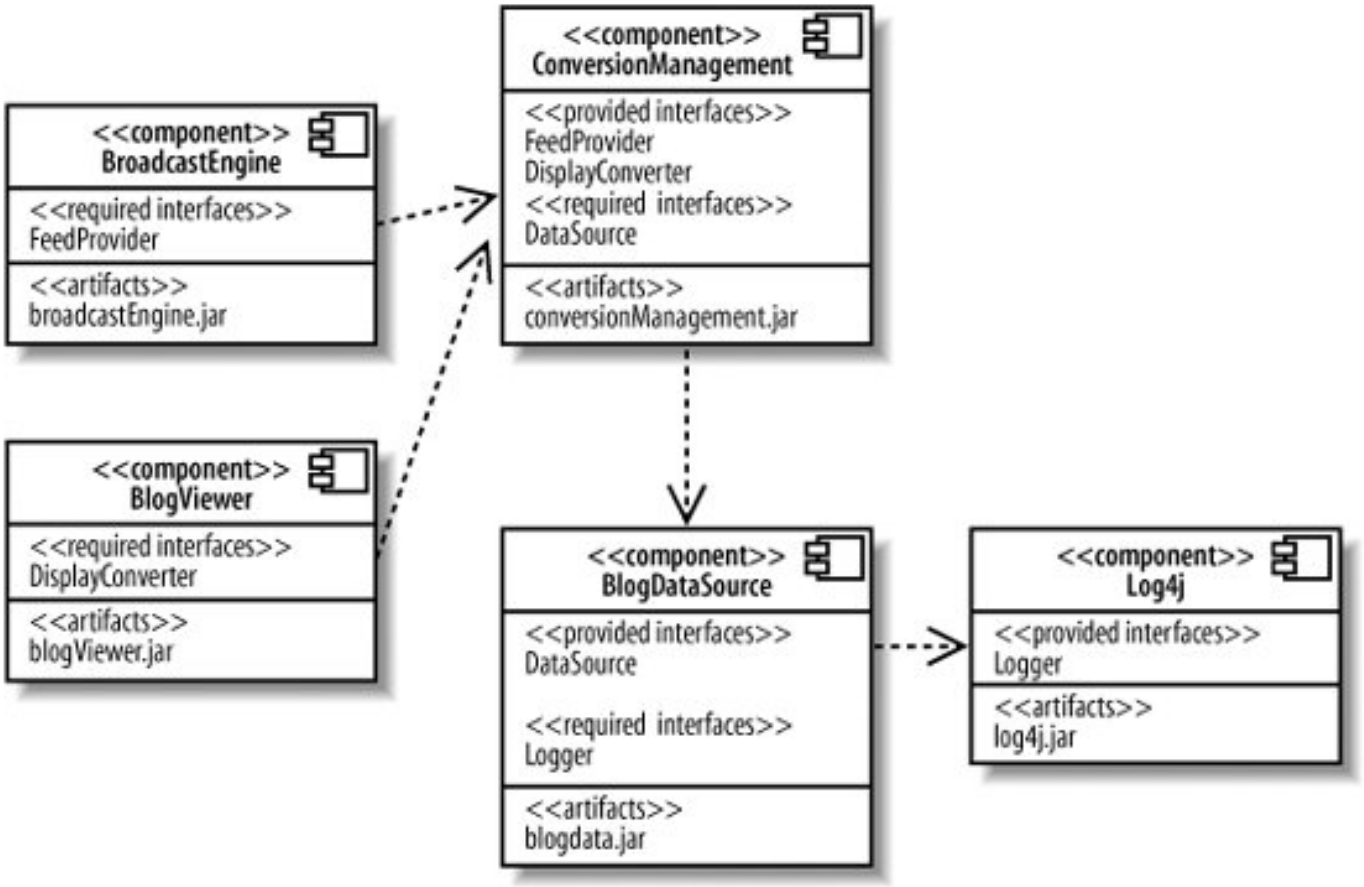
Figure 12-10. Focusing on the key components and interfaces in your system

How to choose

- The second notation is good at showing simplified higher level views of component dependencies.
- This can be useful for understanding a system‘s configuration management or deployment concerns because emphasizing component dependencies and listing the manifesting artifacts allows you to clearly see which components and related files are required during deployment, as shown in Figure 12-11.

配置管理或部署重点

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



12.5. 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, they help the component do its job.
- You can show realizing classes by drawing them (and their relationships) inside the component.
  - Figure 12-12 shows that the BlogDataSource component contains the Blog and Entry classes.
  - It also shows the aggregation relationship between the two classes.
- You can also show a component's realizing classes by drawing them outside the component with a dependency arrow from the realizing class to the component, as shown in Figure 12-13.

Figure 12-12. The Blog and Entry classes realize the BlogDataSource component

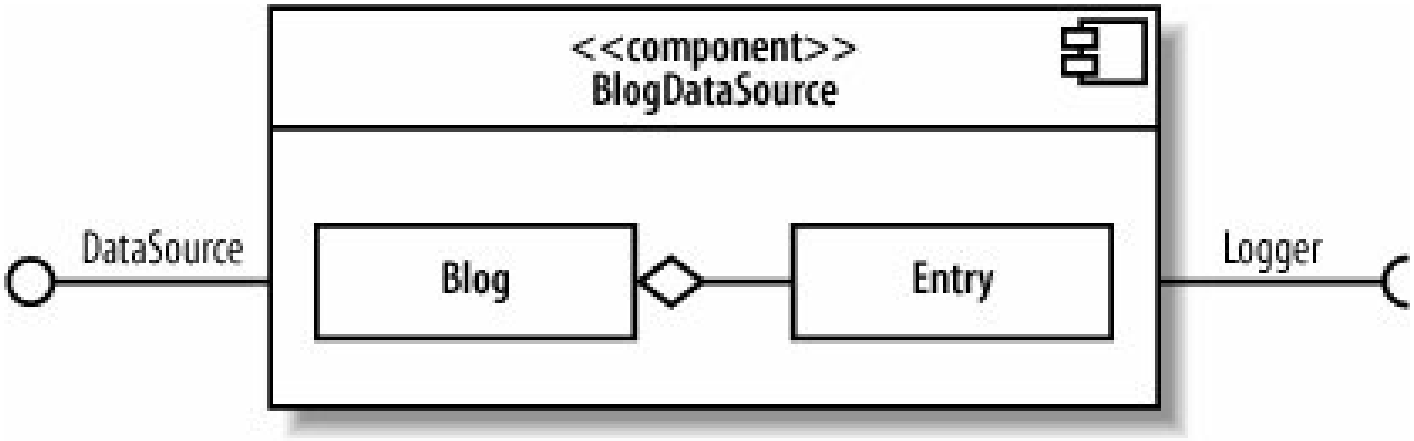


Figure 12-13. Alternate view, showing the realizing classes outside with the dependency relationship

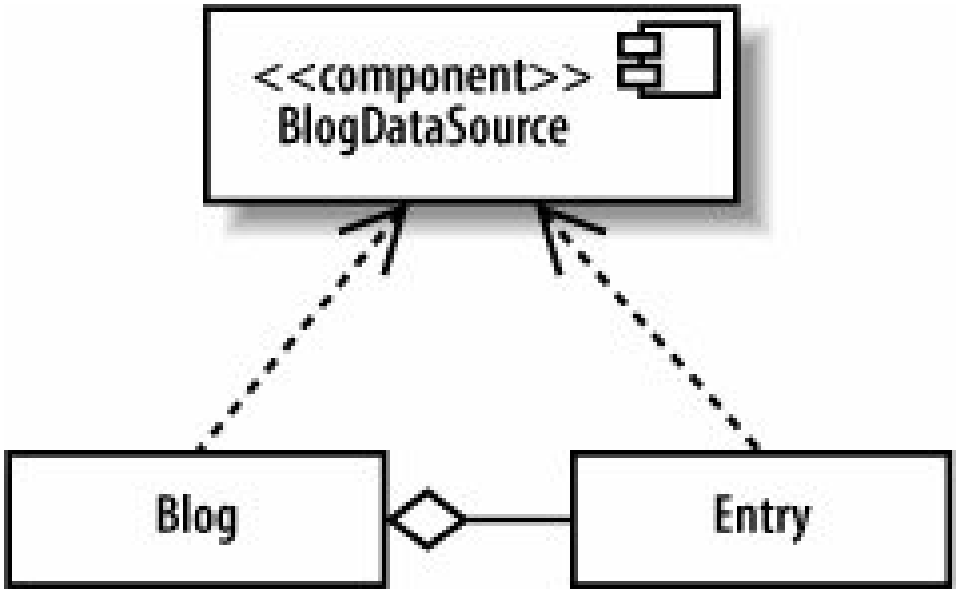
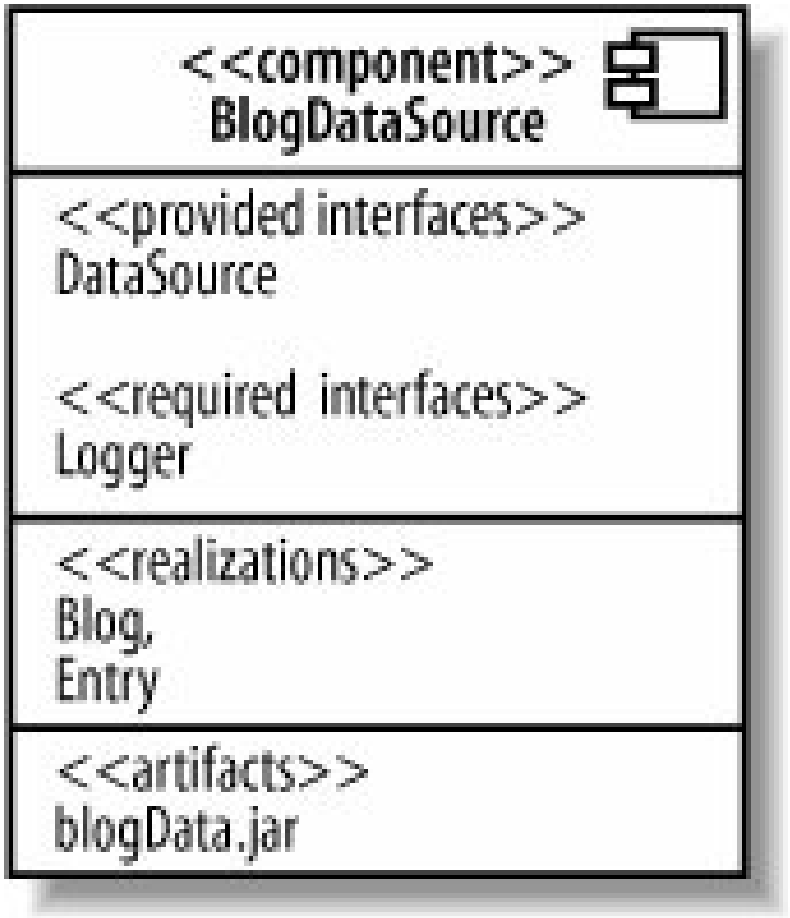


Figure 12-14. You can also list the realizing classes inside the component



12.6. Ports and Internal Structure

- Components can also have ports and internal structure.
- You can use ports to model distinct ways that a component can be used with related interfaces attached to the port.
  - In Figure 12-15, the ConversionManagement component has a Formatting and a Data port, each with their associated interfaces.
- You can show the internal structure of a component to model its parts, properties, and connectors (see Chapter 11 for a review of internal structure).
  - Figure 12-16 shows the internal structure of a BlogDataSource component.

Figure 12-15. Ports show unique uses of a component and group "like" interfaces

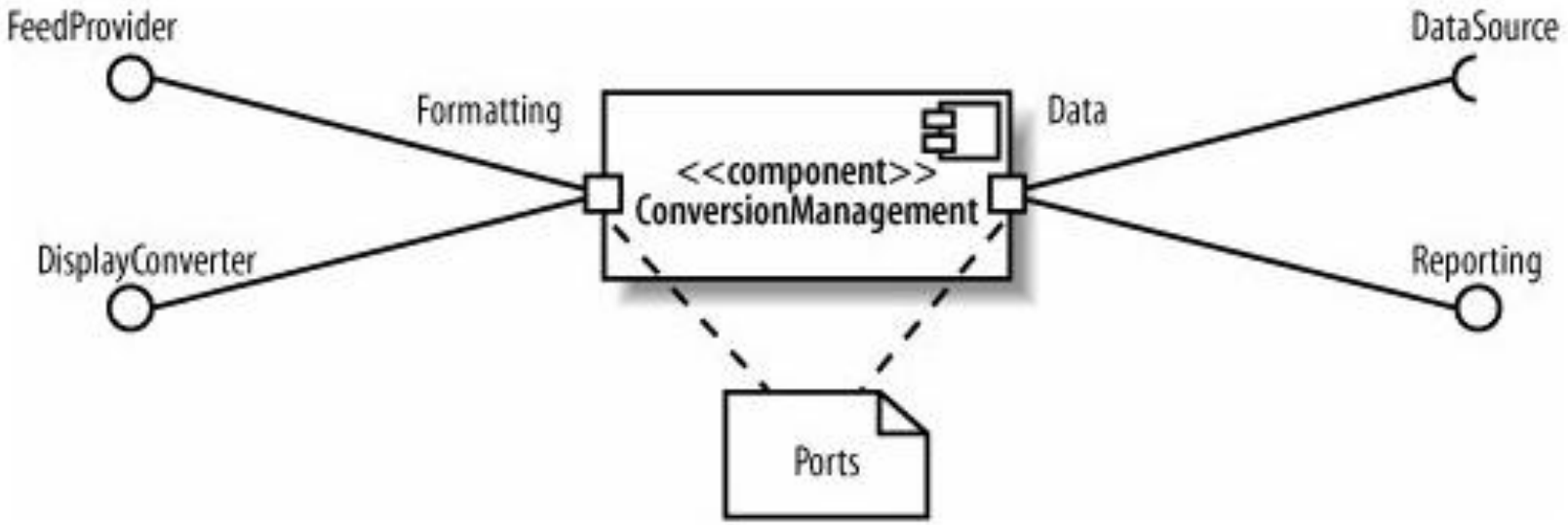
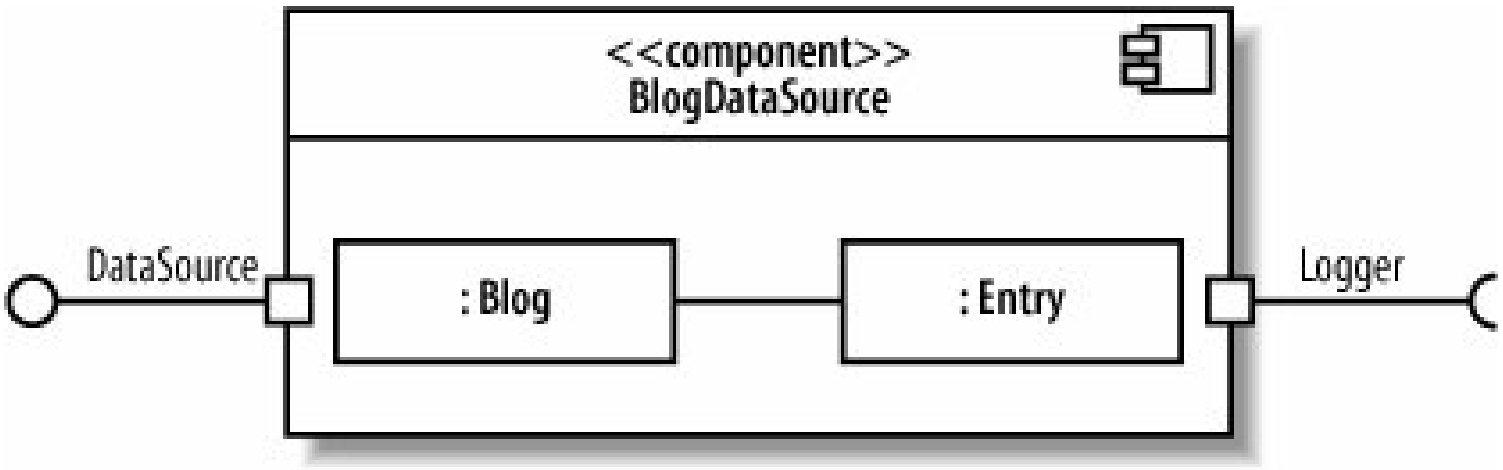


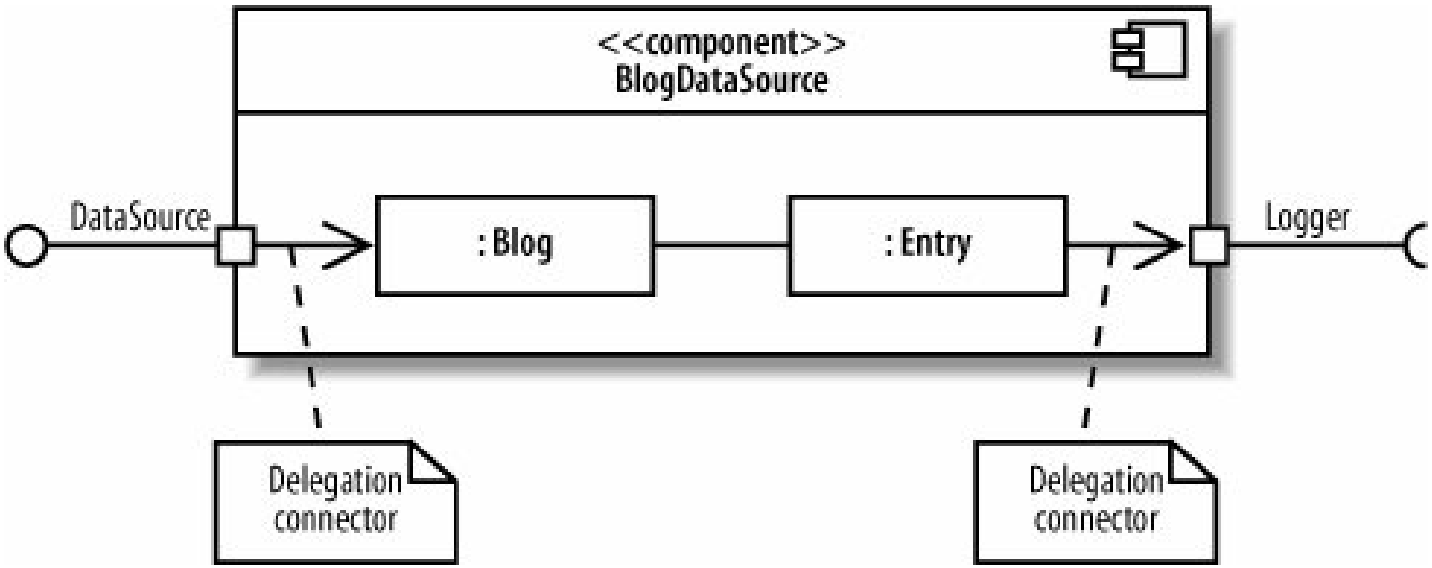
Figure 12-16. Showing the internal structure of a component



12.6.1. Delegation Connectors

- A component's provided interface can be realized by one of its internal parts. Similarly, a component's required interface can be required by one of its parts.
  - In these cases, you can use *delegation connectors* to show that internal parts realize or use the component's interfaces.
- Delegation connectors are drawn with arrows pointing in the "direction of traffic," connecting the port attached to the interface with the internal part.
  - If the part realizes a provided interface, then the arrow points from the port to the internal part.
  - If the part uses a required interface, then the arrow points from the internal part to the port.

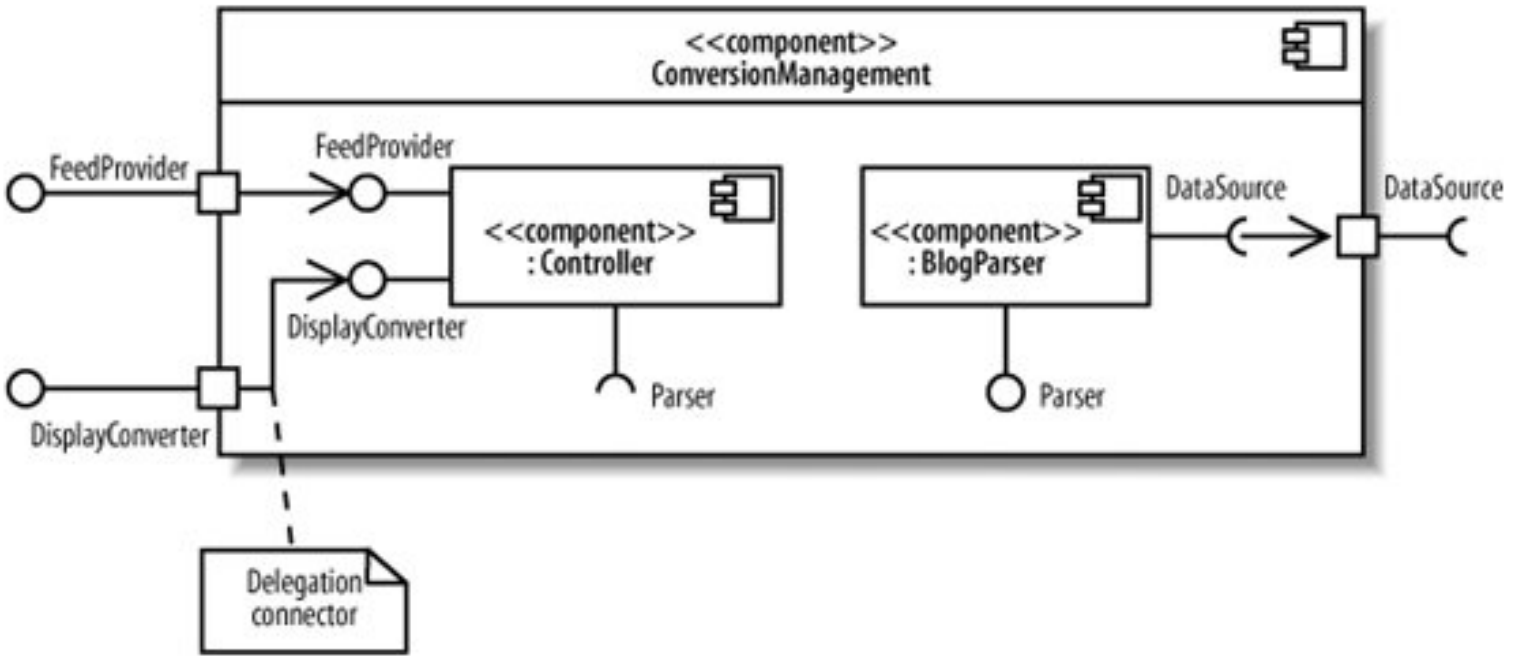
Figure 12-17. Delegation connectors show how interfaces correspond to internal parts: the Blog class realizes the DataSource interface and the Entry class requires the Logger interface



Understanding delegation connector

- You can think of the delegation connectors as follows:
  - the port represents an opening into a component through which communications pass, and delegation connectors point in the direction of communication.
  - So, a delegation connector pointing from a port to an internal part represents messages being passed to the part that will handle it. 消息传递到组件内部

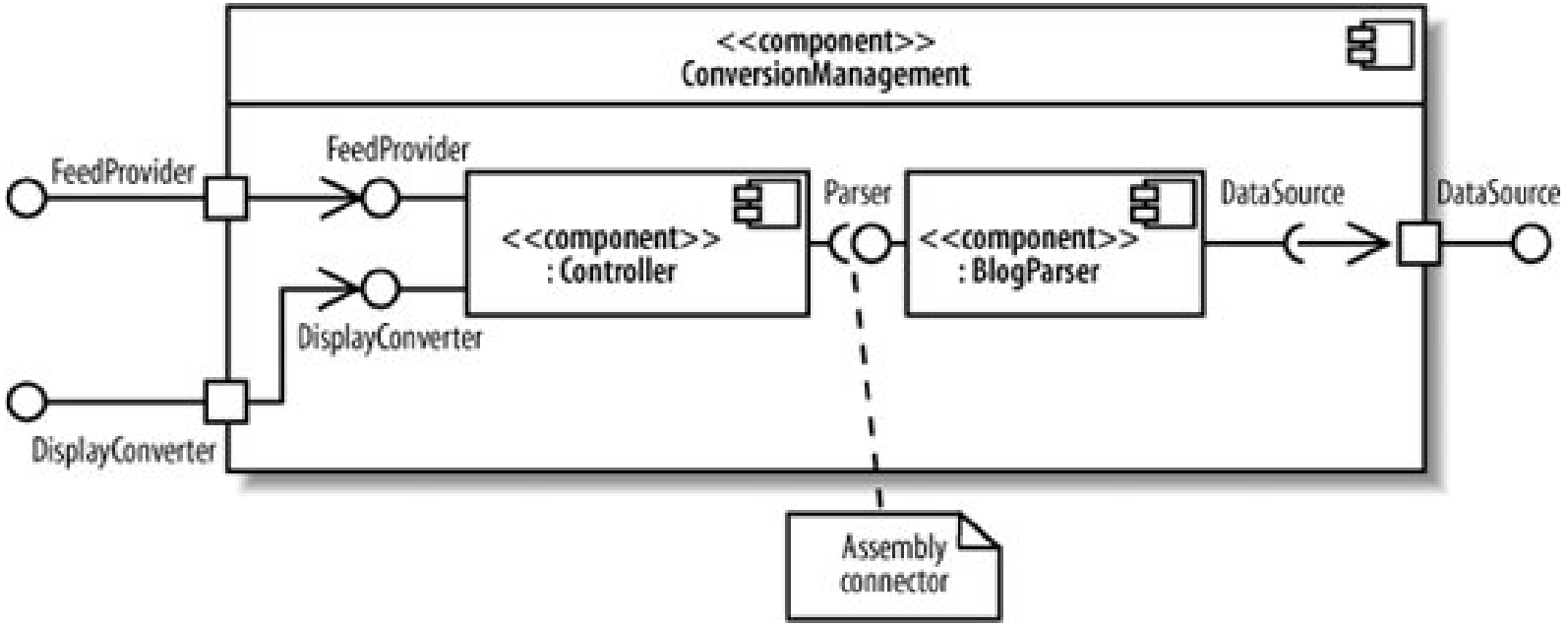
Figure 12-18. Delegation connectors can also connect interfaces of internal parts with ports



12.6.2. Assembly Connectors

- *Assembly connectors* show that a component requires an interface that another component provides.
- Assembly connectors snap together the ball and socket symbols that represent required and provided interfaces.
- Figure 12-19 shows the assembly connector notation connecting the Controller component to the BlogParser component.

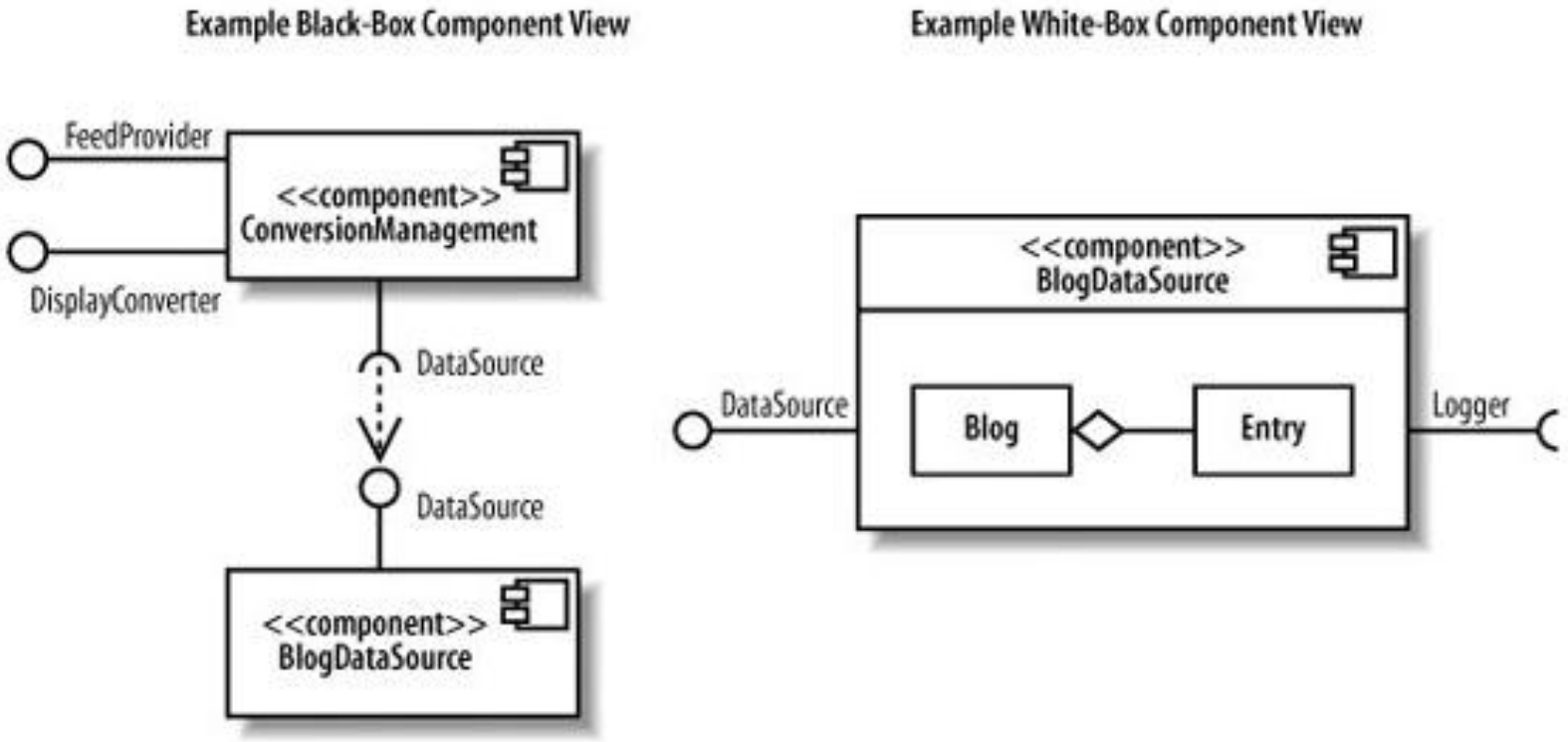
Figure 12-19. Assembly connectors show components working together through interfaces



12.7. Black-Box and White-Box Component Views

- There are two views of components in UML: a black-boxview and a white-box view. 黑盒|白盒视图
  - The black-box view shows how a component looks from the outside, including its required interfaces, its provided interfaces, and how it relates to other components. A black-box view specifies nothing about the internal implementation of a component.
  - The white-box view, on the other hand, shows which classes, interfaces, and other components help a component achieve its functionality.
- So, what's the difference in practical terms?
  - A white-box view is one that shows parts inside a component, whereas a black-box view doesn't, as shown in Figure 12-20.

Figure 12-20. Black-box component views are useful for showing the big picture of the components in your system, whereas white-box views focus on the inner workings of a component



Comparison

- When modeling your system, it's best to use black-box views to focus on large-scale architectural concerns.
  - Black-box views are good at showing the key components in your system and how they're connected.
  - White-box views, on the other hand, are useful for showing how a component achieves its functionality through the classes it uses.
- Black-box views usually contain more than one component, whereas in a white-box view, it‘s common to focus on the contents of one component.

黑盒视图用于描述多个组件的关系；
白盒主要关注于一个组件。

Summary

- 12. Managing and Reusing Your System's Parts: Component Diagrams
  - 12.1. What Is a Component?
  - 12.2. A Basic Component in UML
  - 12.3. Provided and Required Interfaces of a Component
  - 12.4. Showing Components Working Together
  - 12.5. Classes That Realize a Component
  - 12.6. Ports and Internal Structure
  - 12.7. Black-Box and White-Box Component Views

Next …

- 13. Organizing Your Model: Packages
- 14. Modeling an Object's State: State Machine Diagrams
- 15. Modeling Your Deployed System: Deployment Diagrams

See you …