Composite Structure
&
Component Diagrams

Greg Guyles
csci5448
Prof. Anderson
November 16, 2012

Executive Summary: Composite Structure & Component Diagrams

This presentation will describe two diagrams defined in the UML 2.0 specification and explain their strengths in featuring some of the important aspects of Object-Oriented design. These diagrams are not better or worse than other UML diagrams you may have worked with, but they offer specialized views of a system that may better serve to express the concepts you are attempting to model.

Composite Structure diagrams explore the internal organization of classes. It can be used to explicitly describe a class as a composition of other classes. The model can also show how the contained classes interact in the working implementation.

Component diagrams allow modelers to provide a simplified, high-order view of of a large system. Classifying groups of classes into components supports the interchangeability and reuse of code. This diagram documents how these components are composed and how they interact in a system.

"As there can be no Classification or Recognition of objects without Perception of them; so there can be no Perception of them without Classification or Recognition."

-Herbert Spencer, *The Principles of Psychology* 1855

Themes

The diagrams covered in this presentation are used to highlight two main themes in Object Oriented Design:

Aggregation

The act or result of forming an object configured from its component parts*

Classification

The act or result of removing certain distinctions between objects, so that we can see commonalities**

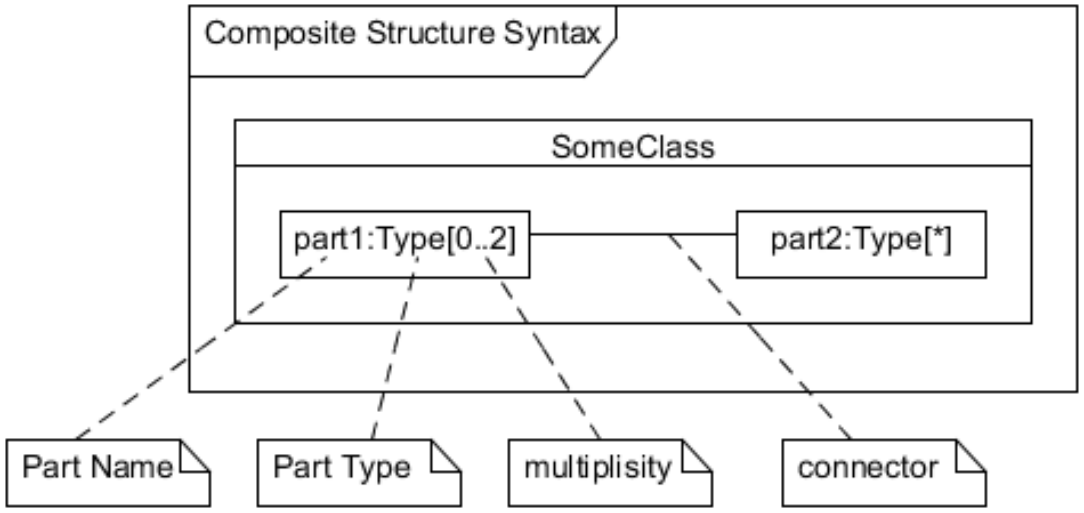
*Odell, 130

**Odell, 122

Composite Structure Diagrams and Aggregation

- Composite Structure Diagrams allow the users to "Peek Inside" an object to see exactly what it is composed of.
- The internal actions of a class, including the relationships of nested classes, can be detailed.
- Objects are shown to be defined as a composition of other classified objects.

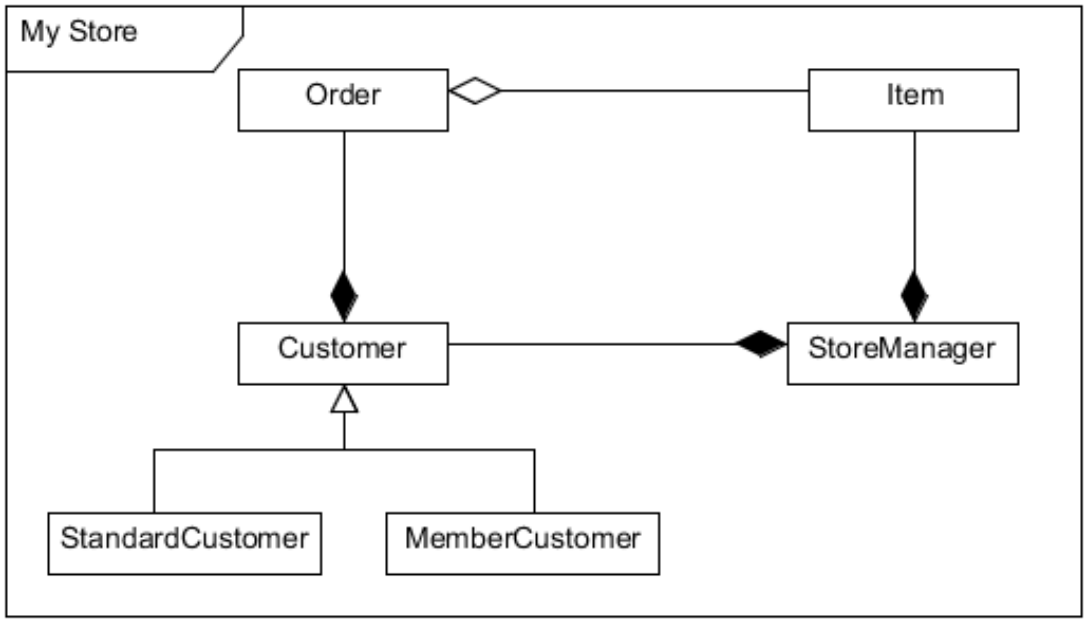
Syntax of a Composite Structure Diagram



Composite Structure Diagrams show the internal parts of a class.

Parts are named: partName:partType[multiplicity]

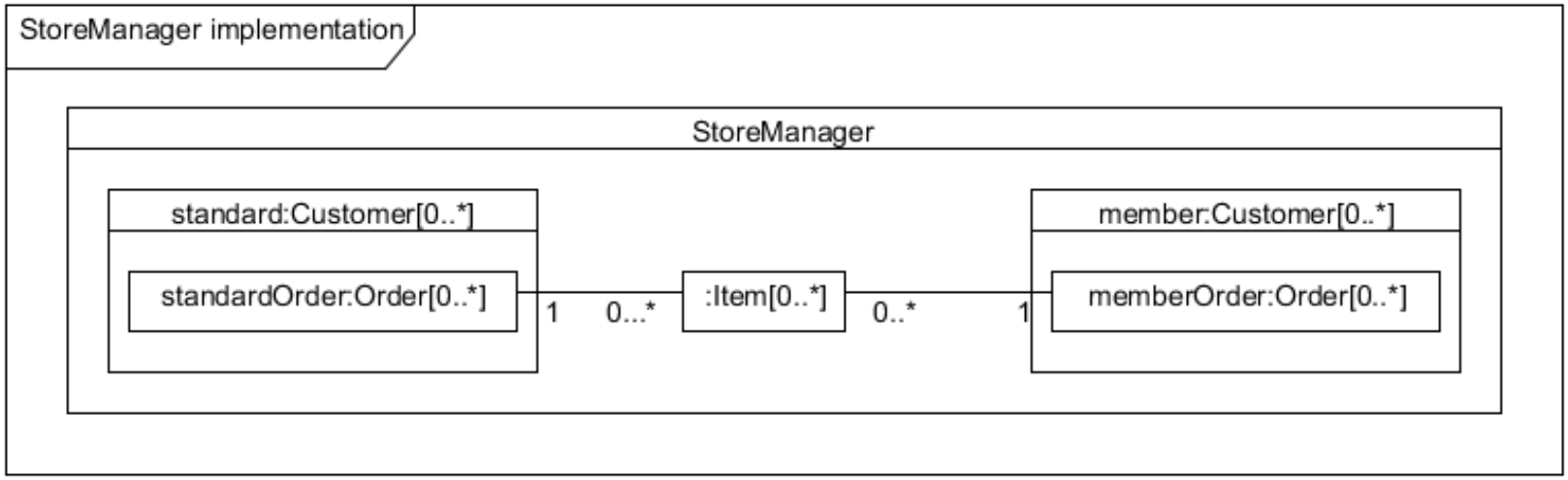
Aggregated classes are parts of a class but parts are not necessarily classes, a part is any element that is used to make up the containing class.



We are modeling a system for an online store. The client has told us that customers may join a membership program which will provide them with special offers and discounted shipping, so we have extended the customer object to provide a member and standard option.

We have a class for `Item` which may be aggregated by the `Order` class, which is composed by the `Customer` class which itself is composed by the `StoreManager` class. We have a lot of objects that end up within other objects.

Everything looks like it ends up inside `StoreManager`, so we'll create a composite structure diagram to really see what it's made of.

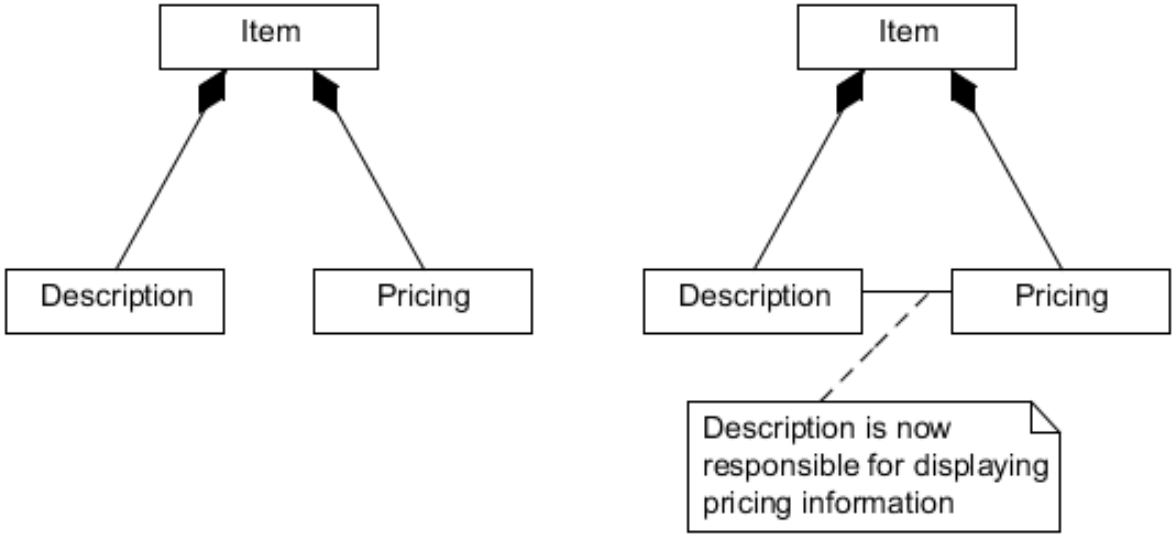


Here we see `StoreManager` from its own perspective, instead of the system as a whole. `StoreManager` directly contains two types of objects (`Customer` and `Item`) as is indicated by the two composition arrows on the class diagram.

What this diagram shows more explicitly is the inclusion of the subtypes of `Customer`. Notice that the type of both of these parts is `Customer`, as the store sees both as `Customer` objects.

We also see a connector which shows the relation between `Item` and `Order`. `Order` is not directly contained within the `StoreManager` class but we can show relations to parts nested within the objects it aggregates.

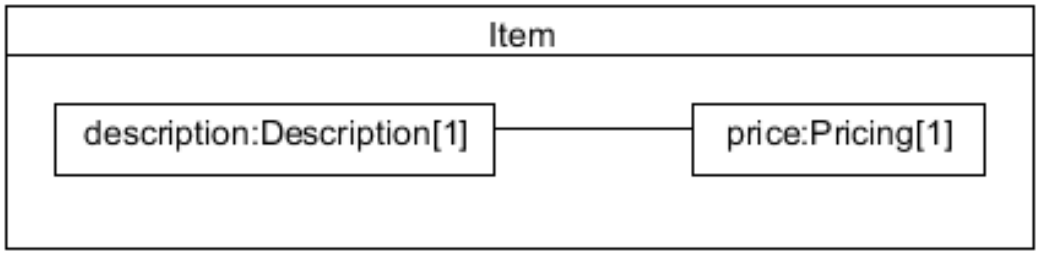
What a Class Diagram can't show



We now have expanded our model to define the `Item` object as one which is composed of a `Description` object and a `Pricing` object. We then realize the implementation may be simplified if `Description` can access the pricing information, so we draw a reference to the `Pricing` object.

The problem is that this diagram is wrong. In a class diagram the reference between `Description` and `Pricing` is ambiguous. This does show that `Description` will have a reference to a `Pricing` object but this diagram does not specify that it be the `Pricing` object contained within the same `Item` object as itself.

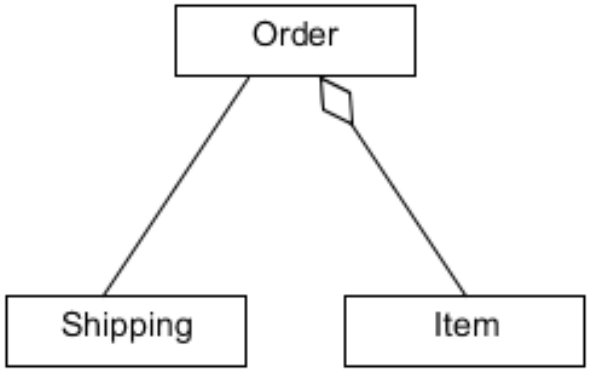
Composite Structure Diagrams are Contained

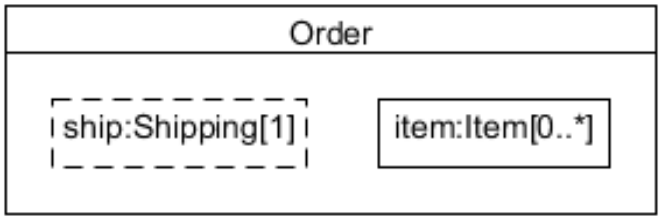


The reference between the `Description` and `Pricing` objects is contained to objects that are composed by `Item`.

The specific implementations of an object's activity can be clearly modeled.

References to External parts





We have seen examples of how Composite Structure diagrams are great at describing aggregation, but your models will also need to contain references to objects outside of the class you are modeling.

References to external objects are shown as a part with a dashed rectangle. Even though they reference object is outside of the class, the reference itself is within the modeled class and is an important step in showing its implementation.

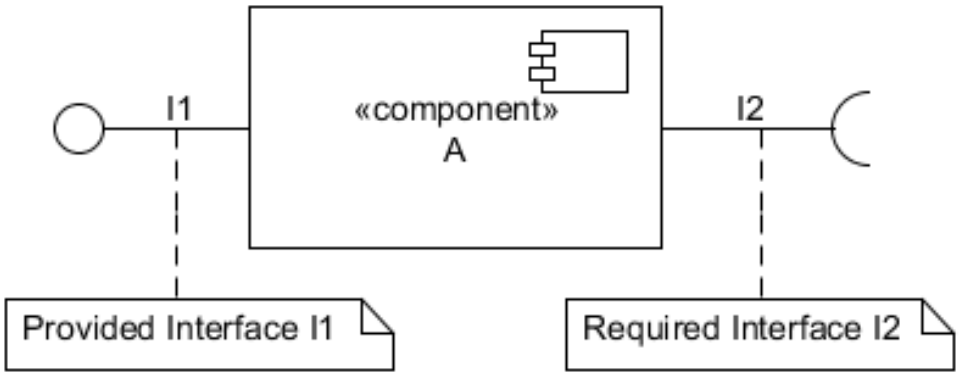
Component Diagrams and Classification

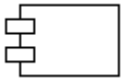
- Components are made up of software objects that have been classified to serve a similar purpose.
- By classifying a group of classes as a component the entire system becomes more modular as components may be interchanged and reused.
- This diagram documents the encapsulation of the component and the means by which the component interacts via interfaces.

“A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.”

-The UML 2.0 specification

Syntax of a Component Diagram

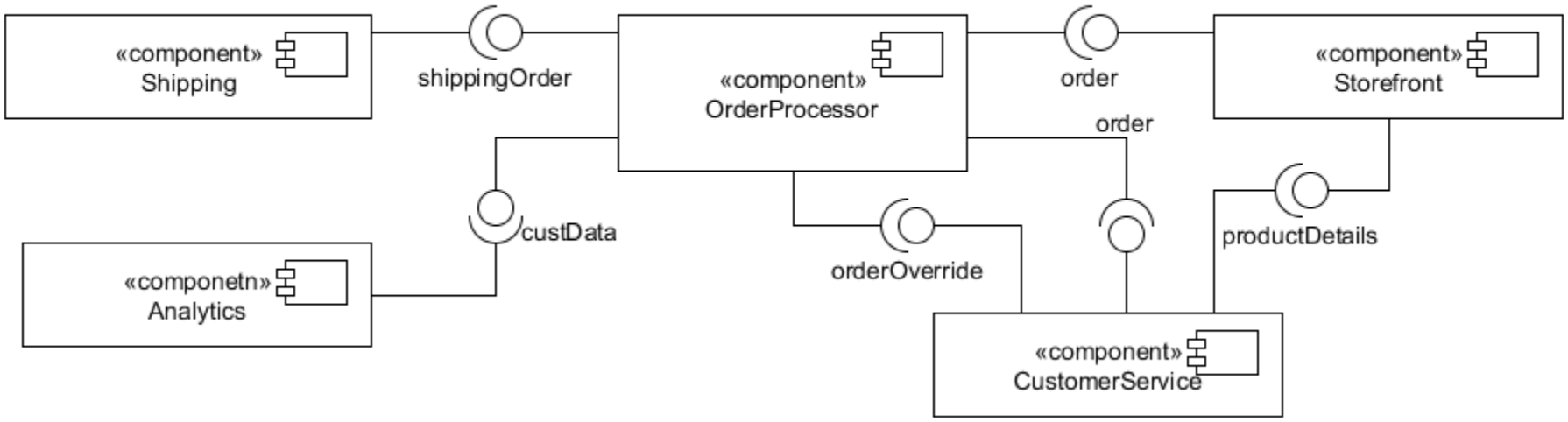


Components are denoted by the use the special symbol,  in the upper right corner.

Components interact via interfaces, shown here using the "lollipop" notation. This component provides interface `I1` meaning it outputs information in the form of `I1`, and requires interface `I2`, meaning it requires input in the form of `I2`.

This specific diagram does not show which class contained within the component require or provide interfaces `I1` and `I2`, only that the component as a whole enforces them.

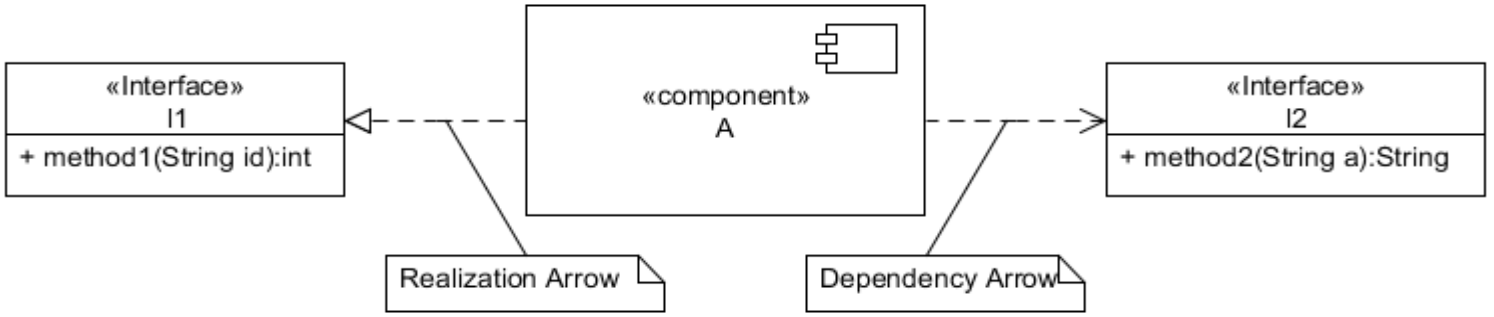
High-Level System View



One main benefit of Component diagrams if to simplify the high-level view of the system. Shown above is a much larger view of what is involved in our online store. By using a component diagram we see the system as a group of nearly independent subsystems that interact with each other in a specifically defined way.

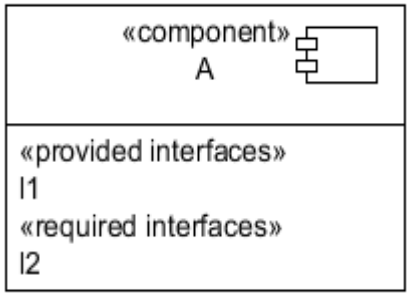
Each component is responsible for the action for which it is named and interface(s) it provides. As long as those requirements are maintained changes to one component will not percolate to other components.

Interfaces: The Component's Contract



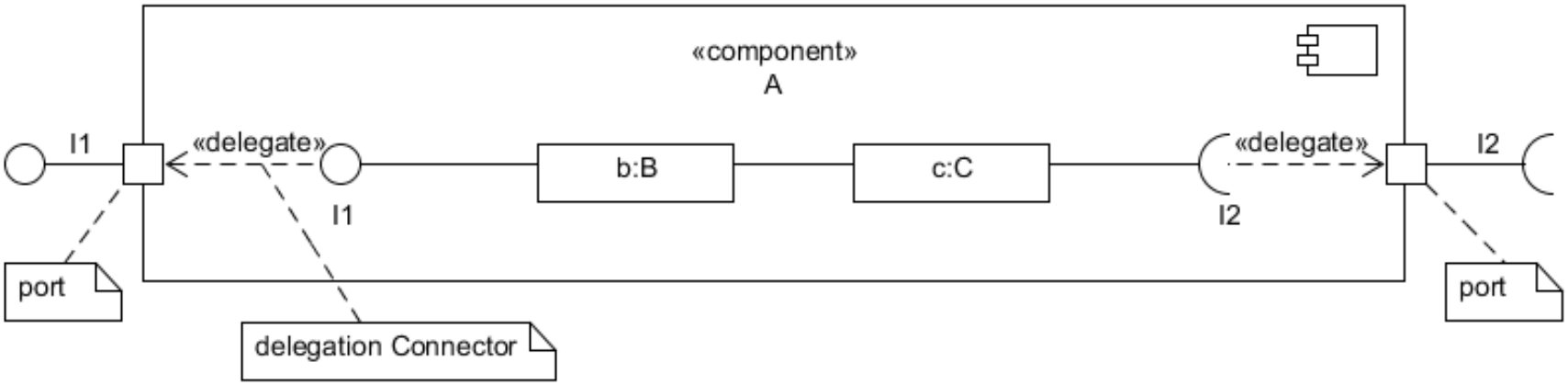
Interfaces shown using Realization and Dependency arrows

The component interacts with the system by way of the interface. This is what allows components to be interchanged and reused. The interface is the contract by which the component must comply. On the previous slide the ball and socket notation was used to show required and provided interfaces. Here, two alternative styles are shown.



Interfaces show in the component body

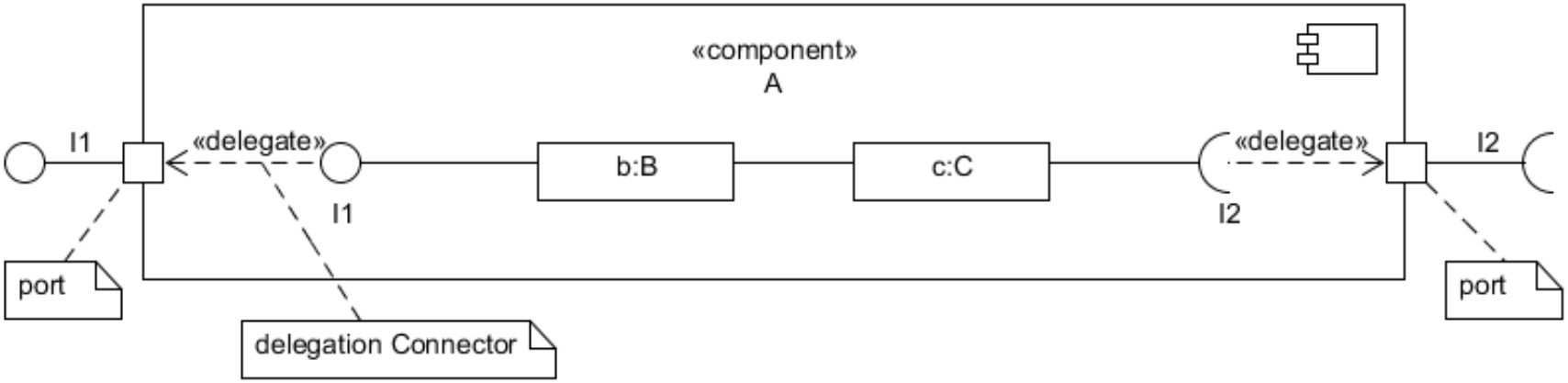
It's what's on the inside that matters



The internal composition of components can also be modeled using component diagrams, this is called a *white-box view* of the diagram because we can see inside. Conversely, the examples on the previous slides are called *black-box views*.

Ports are shown as squares bordering the component, these indicate how the interfaces of the component are used internally. Objects implementing a required interface are received via a port and objects implementing a provided interface are shared via a port.

Delegation connectors

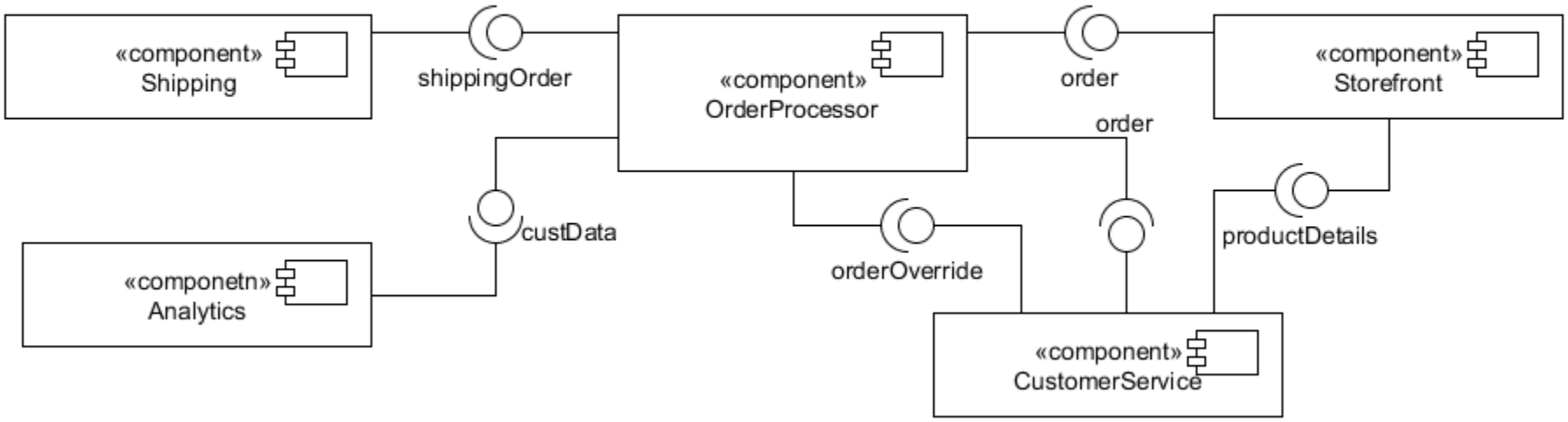


Delegation connectors are used to show which internal part receives or provides shared objects. This is used to distinguish the interface requirements of the internal class from that of the component as a whole.

Though these appear to be the same in this diagram, they may not necessarily be so, a component may require a specialized object and pass it to an internal part that only requires a more generalized object.

Delegation Connectors are one of the UML elements which are not consistently represented. Arrows may or may not be used, they have been seen to either point to the port, or in the direction data moves through the component. Lines of delegation connectors have been shown to be dashed or solid. Using the <<delegate>> title will clarify your intention.

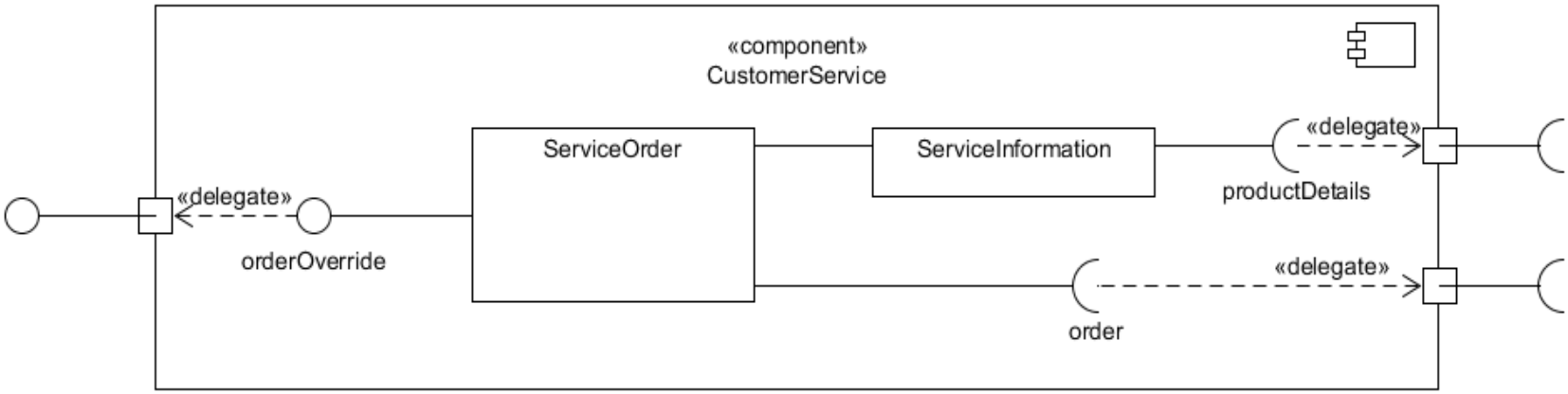
Customer Service Component



Lets review the details of the `CustomerService` component shown earlier in the high-level system view.

This component serves the customer service department of the store and as such must be able to retrieve product information form the `Storefront`, retrieve order information from the `OrderProccessor` and it must be able to issue an `orderOverride` object to the `OrderProcessor` which can change order details, give discounts, expedite processing, etc.

Customer Service Component



This is the *white-box view* of the `CustomerService` Component.

The required and provided objects all pass through the component via its ports and the delegation connectors indicate which internal classes handle them. A reference between the internal objects `ServiceOrder` and `ServiceInformation` is also shown.

We can gain an abstract view of how the component processes information. We see what type of information comes in, what classes are involved and how they interact, and what output is shared.

Conclusion

Composite Structure diagrams are a useful way to examine the internal composition and interactions of a class. They can show distinctions within the class such as specialized classes which are all treated as the same general class by the modeled class. The interactions shown in a Composite Structure diagram are confined to the containing class which allows users to show specifics that class diagrams cannot.

Component Diagrams provide a clear view of how components interact via interfaces. They describe the requirements of the component and its content and give an abstract view of how the component processes information.

These tools provide useful perspectives by which to model software. By looking at a project from multiple points of view we gain a fuller understanding of its implementation and abilities. UML models are often used when trying to convey a point or concept about the system to another person who may or may not be fluent in Computer Science terminology. Communicating with models increases your vocabulary and will help you to convey points which may otherwise seem too abstract to understand.

References

Arlow, Jim, and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. 2nd ed.
Upper Saddle River, NJ: Addison-Wesley, 2005. Print.

Miles, Russ, and Kim Hamilton. *Learning UML 2.0*.
Sebastopol, CA: O'Reilly, 2006. Print.

Odell, James J. *Advanced Object-Oriented Analysis and Design Using UML*.
Cambridge: Cambridge UP, 1998. Print.