Figure 5–8 also shows the Planning package performing a private import of the Plans package, as illustrated by the dependency labeled with «access». This is necessary to allow the PlanAnalyst class to access the GardeningPlan and PlanMetrics classes with unqualified names. But, since an access dependency is private, the Greenhouse package's import of the Planning package doesn't provide the Greenhouse package elements, such as the Gardener class, with the ability to reference GardeningPlan and PlanMetrics with unqualified names. In addition, the elements of the Greenhouse package can't even see the PlanAnalyst class because it has private visibility.

Looking inside the Greenhouse package, the Gardener class must use the qualified names of the elements within the StorageTank package because its namespace does not import the package. For example, it must use the name StorageTank::WaterTank to reference the WaterTank class. Taking this one more step, we look at the elements within the EnvironmentalController package. They all have private visibility. This means they are not visible outside their namespace, that is, the EnvironmentalController package.

To summarize, an unqualified name (often called a *simple name*) is the name of the element without any path information telling us how to locate it within our model. This unqualified name can be used to access the following elements in a package [64, 65]:

- Owned elements
- Imported elements
- Elements within outer packages

A nested package can use an unqualified name to reference the contents of its containing package, through all levels of nesting. However, if an element in an outer package is of the same type and has the same name as one within the inner package, a qualified name must be used. The access situation from a containing package's perspective is quite different, though—the package is required to import its nested packages to reference their elements with unqualified names [66, 67].

5.3 Component Diagrams

A component represents a reusable piece of software that provides some meaningful aggregate of functionality. At the lowest level, a component is a cluster of classes that are themselves cohesive but are loosely coupled relative to other clusters. Each class in the system must live in a single component or at the top level of the system. A component may also contain other components.

Components are a type of structured classifier whose collaborations and internal structure can be shown on a component diagram. A component, collaborating with other components through well-defined interfaces to provide a system's functionality, may itself be comprised of components that collaborate to provide its own functionality. Thus, components may be used to hierarchically decompose a system and represent its logical architecture. This logical perspective of a component is new with UML 2.0. Previously, a component was regarded as a physical item that was deployed within a system. Now, a component may be manifested by an artifact that is deployed on a node [68, 69].

The essential elements of a component diagram are components, their interfaces, and their realizations.

Essentials: The Component Notation

Since a component is a structured classifier, its detailed assembly can be shown with a composite structure using parts, ports, and connectors. Figure 5–9 shows the notation used to represent a component. Its name, `EnvironmentalControlSystem` in this case, is included within the classifier rectangle in bold lettering, using the specific naming convention defined by the development team. In addition, one or both of the component tags should be included: the keyword label «`component`» and the component icon shown in the upper right-hand corner of the classifier rectangle [70, 71].

On the boundary of the classifier rectangle, we have seven ports, which are denoted by small squares. Ports have public visibility unless otherwise noted. Components may also have hidden ports, which are denoted by the same small squares, but they are represented totally inside the boundary of the composite structure, with only one edge touching its internal boundary. Hidden ports may be used for capabilities such as test points that are not to be publicly available. Ports

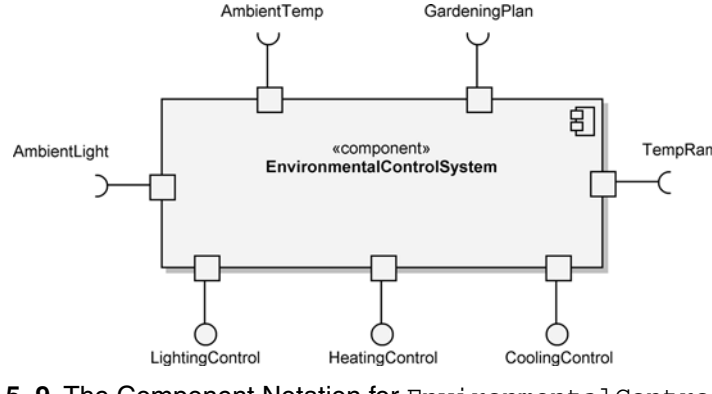

Figure 5–9 The Component Notation for `EnvironmentalControlSystem`

are used by the component for its interactions with its environment, and they provide encapsulation to the structured classifier. These seven ports are unnamed but should be named, in the format of *port name* : *Port Type*, when needed for clarity. The port type is optional when naming a port [72, 73].

To the ports shown in Figure 5–9, we have connected interfaces, which define the component's interaction details. The interfaces are shown in the ball-and-socket notation. Provided interfaces use the ball notation to specify the functionality that the component will provide to its environment; `LightingControl` is an example of a provided interface. Required interfaces use the socket notation to specify the services that the component requires from its environment; `AmbientTemp` is one of the required interfaces [74, 75].

This representation of `EnvironmentalControlSystem` is considered a black-box perspective since we see only the functionality required or provided by the component at its boundary. We are not able to peer inside and see the encapsulated components or classes that actually provide the functionality.

A one-to-one relationship between ports and interfaces is not required; ports can be used to group interfaces, as shown in Figure 5–10. This may be done, for example, to provide clarity in a very complex diagram or to represent the intention of having one port through which certain types of interactions will take place. In Figure 5–10, the ambient measurements of light and temperature are received at one port. Similarly, the gardening plan and temperature ramp information provided by the staff of the Hydroponics Gardening System are received at a single port. Note that the interface names are separated by a comma when using this notation. On these complex ports, we could alternately show separate interfaces such that one port would contain two required interfaces, one named `AmbientLight` and the other named `AmbientTemp`, and the other port would contain a required interface named `GardeningPlan` and another one named `TempRamp` [76, 77].

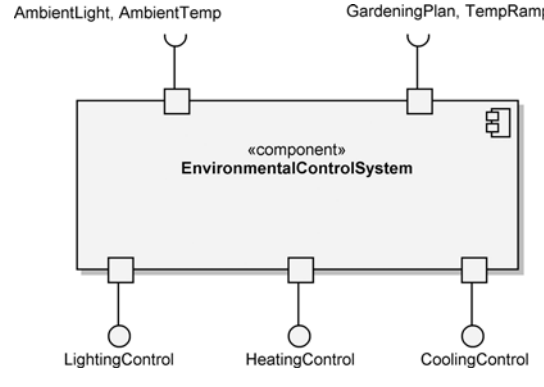

Figure 5–10 The Component Notation with Interface Grouping

Essentials: The Component Diagram

During development, we use component diagrams to indicate the logical layering and partitioning of our architecture. In them, we represent the interdependencies of components, that is, their collaborations through well-defined interfaces to provide a system's functionality. Figure 5–11 shows the component diagram for `EnvironmentalControlSystem`. This white-box perspective shows the four encapsulated components that provide its functionality: `Environmental Controller`, `LightingController`, `HeatingController`, and `CoolingController` [78, 79].

As in Figure 5–9, the ball-and-socket notation is used to specify the required and provided interfaces of each of the components. The interfaces between the components are called *assembly connectors*; they are also known as *interface connectors*. Though the assembly connectors are shown in the ball-and-socket notation, we could have used a straight line to represent each connection. However, this would not be as informative. The interface between `Environmental Controller` and `CoolingController` is shown with a dependency to illustrate another form of representation. This dependency is actually redundant because the interface names are the same: `CoolControl` [80, 81].

Previously, we mentioned the reusable nature of components. For example, as long as another component fulfills the requirements of `LightingController`'s

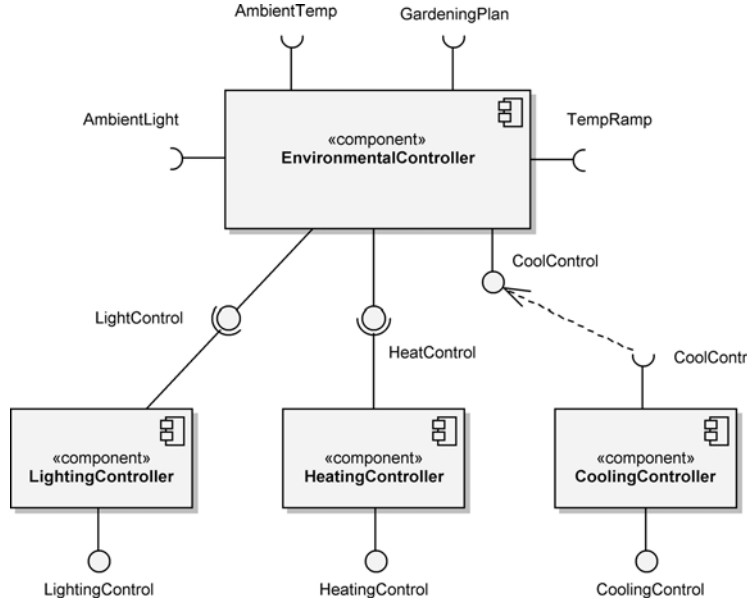

Figure 5–11 The Component Diagram for `EnvironmentalControlSystem`

interfaces, it may replace `LightingController` within `EnvironmentalControlSystem`. This property of components means that we may more easily upgrade our system as needed. In fact, the entire contents of `EnvironmentalControlSystem` may be replaced, as long as its required and provided interface requirements are met by its contained components.

Essentials: Component Interfaces

If we need to show more details about a component's interfaces, we may provide an interface specification, as shown in Figure 5–12. In our case, the specification focuses on only two of the seven interfaces of `EnvironmentalController`: `CoolControl` and `AmbientTemp` [82, 83].

`EnvironmentalController` realizes the `CoolControl` interface; this means that it provides the functionality specified by the interface. This functionality is starting, stopping, setting the temperature, and setting the fan speed for any component using the interface, as shown by the contained operations. These operations may be further detailed with parameters and return types, if needed. The `CoolingController` component (shown in Figure 5–11) requires the functionality of this interface.

Figure 5–12 also shows the dependency of the `EnvironmentalController` component on the `AmbientTemp` interface. Through this interface, `EnvironmentalController` acquires the ambient temperature that it requires to fulfill its responsibilities within the `EnvironmentalControlSystem` component.

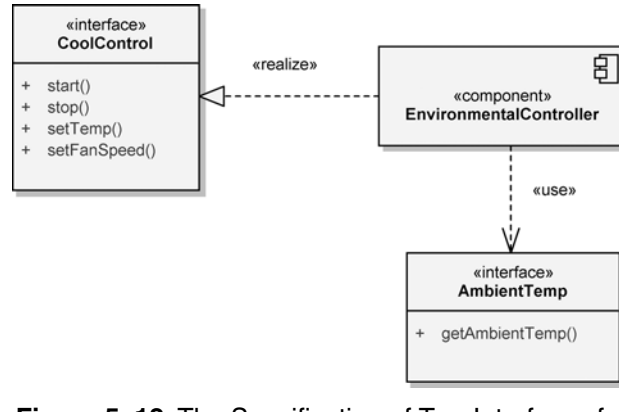

Figure 5–12 The Specification of Two Interfaces for `EnvironmentalController`

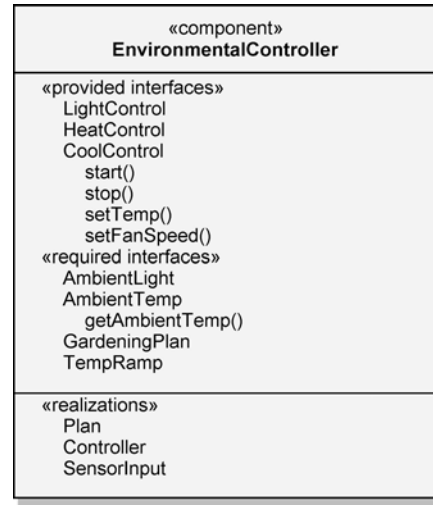

Figure 5–13 An Alternate Notation for `EnvironmentalController`'s Interfaces and Realizations

In Figure 5–13, we show an alternate notation for the interfaces of `EnvironmentalController`. Here we see the three provided interfaces listed under the heading `«provided interfaces»`. For the `CoolControl` interface specified in Figure 5–12, we have provided the associated operations. Likewise, the required interfaces are shown under the heading `«required interfaces»`, along with three classes listed under the `«realizations»` heading [84, 85]. We discuss the concept of realizations in the next section.

Essentials: Component Realizations

Figure 5–13 specifies that the `EnvironmentalController` component is realized by the classes `Plan`, `Controller`, and `SensorInput`. These three classes provide all of the functionality advertised by its provided interfaces. But, in doing so, they require the functionality specified by its required interfaces [86, 87].

This realization relationship between the `EnvironmentalController` component and the `Plan`, `Controller`, and `SensorInput` classes is shown in Figure 5–14. Here, we see a realization dependency from each of the classes to `EnvironmentalController`. This same information may be represented with a containment relationship, as shown in Figure 5–15; each of the classes is physically contained by the `EnvironmentalController` component. The naming convention used for these internal classifiers is tool-specific. Also, note the associations between the classes and the specification of multiplicity [88].

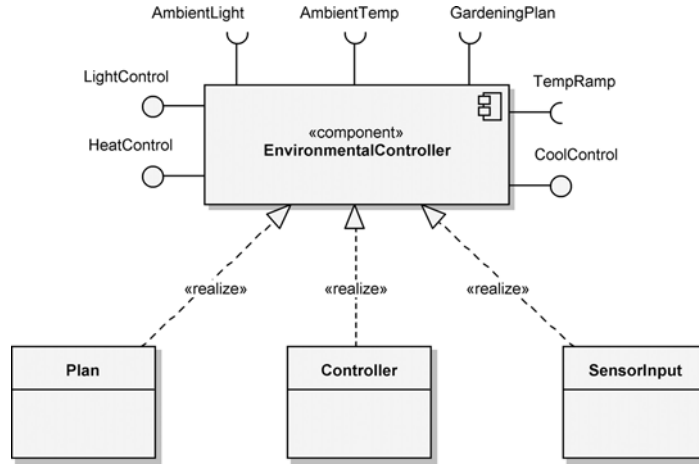

Figure 5–14 The Realization Dependencies for `EnvironmentalController`

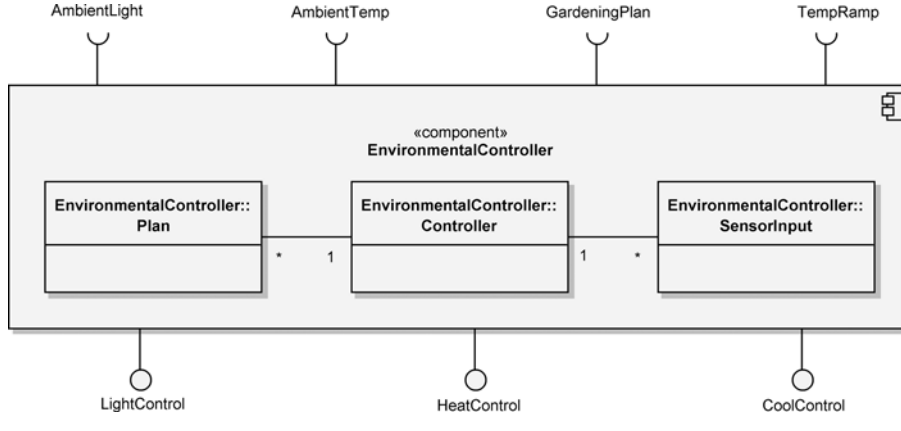

Figure 5–15 The Containment Representation of `EnvironmentalController`'s Realization

Advanced Concepts: A Component's Internal Structure

The internal structure of a component may be shown by using an internal structure diagram; Figure 5–16 shows just such a diagram for the `EnvironmentalControlSystem` subsystem. In this example, we have changed its keyword label from `«component»`, as shown in Figure 5–9, to `«subsystem»` because it is comprised of four components of some complexity that are logically related. Of course, this is a judgment call; the label could reasonably be left as `«component»`. In addition, Figure 5–16 contains a notation that we haven't

encountered yet, the «delegate» label on the lines between the interfaces of the internal components and the ports on the edge of the `EnvironmentalControlSystem`. These connections provide the means to show which internal component fulfills the responsibility of the provided interfaces and which internal component needs the services shown in the required interfaces [89, 90].

Subsystems partition the logical model of a system. A subsystem is an aggregate containing other subsystems and other components. Each component in the system must live in a single subsystem or at the top level of the system. In practice, a large system has one top-level component diagram, consisting of the subsystems at the highest level of abstraction. Through this diagram a developer comes to understand the general logical architecture of a system.

Here we have more of a sense of `EnvironmentalControlSystem` as a reusable component (or subsystem, if you wish) than we did with Figure 5–11. We use ports at its boundary and show that the responsibility for fulfilling the "contract" of an interface has been delegated to one or more of the component's

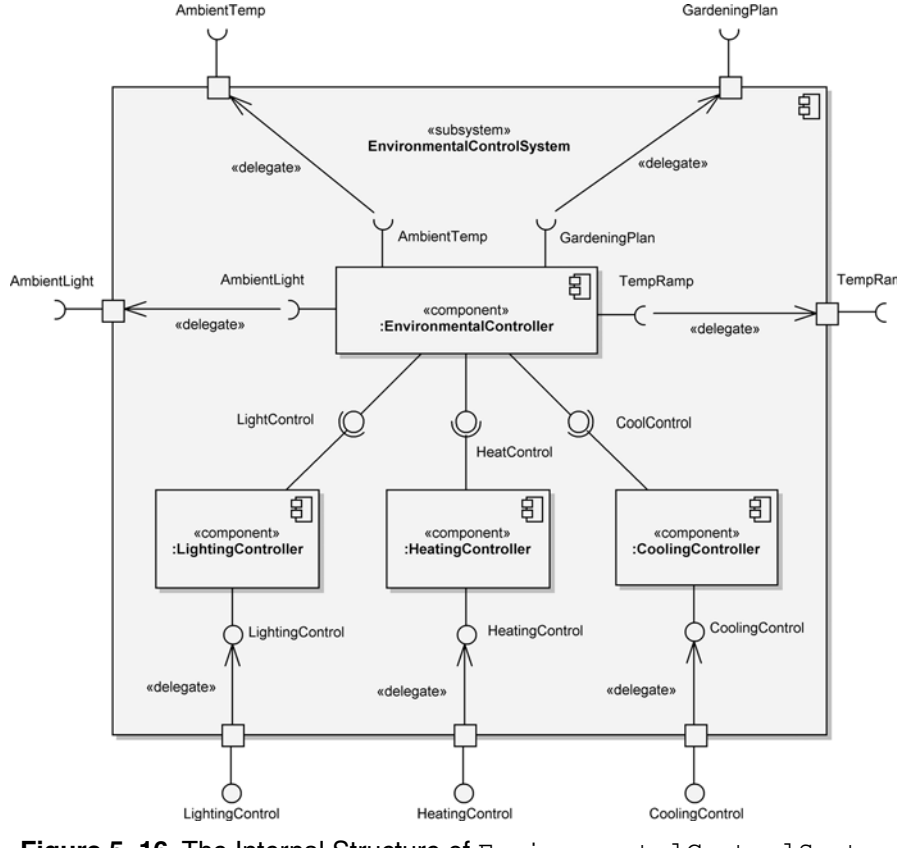

Figure 5–16 The Internal Structure of `EnvironmentalControlSystem`

contained parts. However, remember that these contained parts may require services from the environment of the `EnvironmentalControlSystem` component, such as a gardening plan to meet this contract.

To be specific, `:EnvironmentalController` requires `GardeningPlan`, which specifies the environmental needs (lighting, heating, and cooling) of the Hydroponics Gardening System. The needs of this required interface are delegated to an unnamed port, to which is attached the `GardeningPlan` interface. In this manner, we know that we must provide the `EnvironmentalControlSystem` component with a gardening plan if we intend to use its services. We also recognize that we must provide it with `AmbientLight`, `AmbientTemp`, and `TempRamp` services.

The connectors of `EnvironmentalControlSystem` provide its communication links to its environment, as well as the means for its parts to communicate internally. In Figure 5–16, the type of connectors new to our view of `EnvironmentalControlSystem` are the *delegation connectors* to which we've alluded. Through these connectors, the responsibilities (provided interfaces) of `EnvironmentalControlSystem`, as well as its requirements (required interfaces), are communicated. For example, the `:LightingController` component opaquely provides the `LightingControl` services. A user of `EnvironmentalControlSystem` would not likely have this white-box perspective of the subsystem [91, 92].

5.4 Deployment Diagrams

A deployment diagram is used to show the allocation of artifacts to nodes in the physical design of a system. A single deployment diagram represents a view into the artifact structure of a system. During development, we use deployment diagrams to indicate the physical collection of nodes that serve as the platform for execution of our system.

The three essential elements of a deployment diagram are artifacts, nodes, and their connections.

Essentials: The Artifact Notation

An artifact is a physical item that implements a portion of the software design. It is typically software code (executable) but could also be a source file, a document, or another item related to the software code. Artifacts may have relationships with other artifacts, such as a dependency or a composition [20, 21].