- Section 5.5: use case diagrams
- Section 5.6: activity diagrams
- Section 5.7: class diagrams
- Section 5.8: sequence diagrams
- Section 5.9: interaction overview diagrams
- Section 5.10: composite structure diagrams
- Section 5.11: state machine diagrams
- Section 5.12: timing diagrams
- Section 5.13: object diagrams
- Section 5.14: communication diagrams

In Section III of this book, each chapter presents a different type of application and focuses on a certain set of diagrams that would be most appropriate for the application at the point in the overall lifecycle that we describe.

UML 2.0 Information Sources

The UML 2.0 notation is quite extensive and complex, as a review of the OMG UML 2.0 Specification clearly confirms. Effective practical use of the specification requires turning to additional resources, such as this chapter. If, after reading the OMG UML 2.0 Specification and this chapter, you desire a further level of explanation or detail, we suggest reviewing *The Unified Modeling Language Reference Manual, Second Edition*. Appendix B lists other UML 2.0 resources.

5.2 Package Diagrams

While performing object-oriented analysis and design, you need to organize the artifacts of the development process to clearly present the analysis of the problem space and the associated design. The specific reasons will vary but will focus either on physically structuring the visual model itself or on clearly representing the model elements through multiple views. The benefits of organizing the OOAD artifacts include the following [42]:

- Provides clarity and understanding in a complex systems development
- Supports concurrent model use by multiple users
- Supports version control
- Provides abstraction at multiple levels—from systems to classes in a component
- Provides encapsulation and containment; supports modularity

The primary means to accomplish this organization is the UML package diagram, which provides us the ability to represent grouped UML elements.

The essential elements of a package diagram are packages, their visibility, and their dependencies.

Essentials: The Package Notation

The UML package is one of the two primary notations used on a package diagram. The other one is the dependency relationship, which we will describe later. The notation for the package is a rectangle with a tab on the top left. UML 2.0 specifies that the name of the package is placed in the interior of the rectangle if the package contains no UML elements. If it does contain elements, the name should be placed within the tab. A tool-specific implementation of the naming guidelines appears in Figure 5–2, which provides a black-box perspective of the `HydroponicsGardeningSystem` package that does not show its contained elements [45, 46].

When there are fewer elements to be shown because fewer exist or because we have a focused concern, we can use the appropriate notation (package, use case, class, component, and so on) to show these constituent pieces within the containing package. Figure 5–3 again shows the `HydroponicsGardeningSystem` package, but with two of its contained elements represented as packages themselves. In the representation on the left, we show the `Planning` and `Greenhouse`

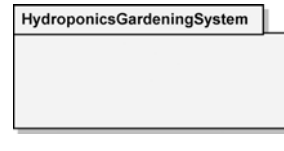

Figure 5–2 The Package Notation for `HydroponicsGardeningSystem`

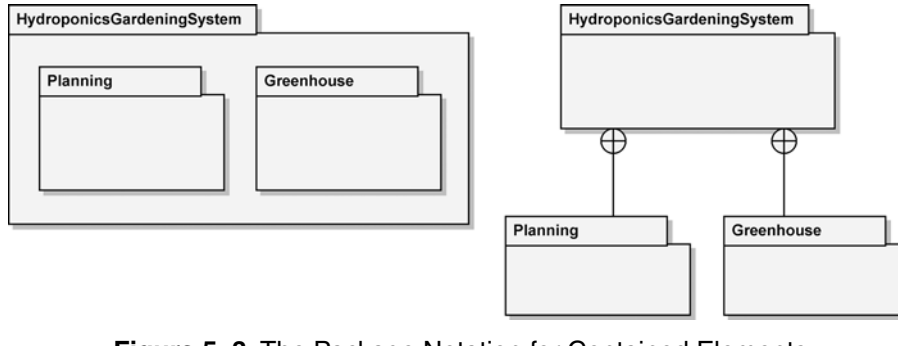

Figure 5–3 The Package Notation for Contained Elements

packages as physically contained packages inside the `HydroponicsGardening-System` package. On the right appears an alternate notation for the containment relationship [47, 48].

Essentials: Visibility of Elements

Access to the services provided by a group of collaborating classes within a package—or more generically, to any elements within a package—is determined by the visibility of the individual elements, including nested packages. The visibility of the elements, defined by the containing package to be either public or private, applies both to contained elements and to those that are imported. The concept of importing elements will be discussed later in this section.

Visibility is defined from the perspective of the containing package, which provides the namespace for its contained elements. Because the package provides the namespace, every contained element has a unique name, at least among other elements of its type. As an example, this means that no two classes contained within the same namespace may have the same name [49, 50]. We will discuss this concept further when we look at import and access.

Elements with public visibility can be thought of as part of the package's interface because these elements are visible to all other elements. Those elements with private visibility are not visible outside the containing package. The definition of public and private visibility is provided here, along with the corresponding notation shown in parentheses [51, 52]:

- Public (+) Visible to elements within its containing package, including nested packages, and to external elements
- Private (-) Visible only to elements within its containing package and to nested packages

On a visual diagram, this visibility notation is placed in front of the element name, as shown in Figure 5–4. The `GardeningPlan` class has public visibility

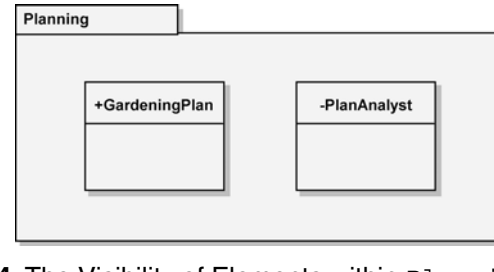

Figure 5–4 The Visibility of Elements within `Planning` Package

to permit other elements to access it, while the `PlanAnalyst` class has private visibility.

Essentials: The Dependency Relationship

If an element has the appropriate visibility to afford access, a dependency relationship to it can be shown representing this access. This dependency relationship is the other primary notation on a package diagram, as we mentioned earlier when discussing the package notation itself. A dependency shows that an element is dependent on another element as it fulfills its responsibilities within the system.

Dependencies between UML elements (including packages), as shown in Figure 5–5, are represented as a dashed arrow with an open arrowhead. The tail of the arrow is located at the element having the dependency (*client*), and the arrowhead is located at the element that supports the dependency (*supplier*). Dependencies may be labeled to highlight the type of dependency between the elements by

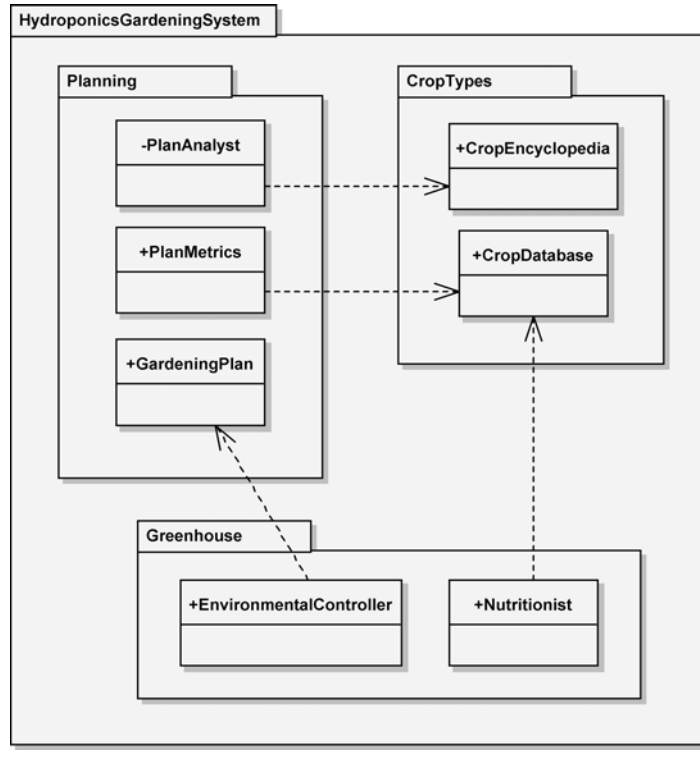

Figure 5–5 The Dependency Notation for `HydroponicsGardeningSystem`

placing the dependency type—denoted by a keyword—within guillemets (« »), for example, «import». Package-specific dependencies include import, access, and merge; dependencies between packages due to the relationships of contained elements include trace, derive, refine, permit, and use [53].

If multiple contained element dependencies exist between packages, these dependencies are aggregated at the package level. A package-level dependency may be labeled with a keyword, denoting type, inside guillemets (« ») ; however, if the contained dependencies are of different types, the package-level dependency is not labeled. Figure 5–6 shows the dependencies of Figure 5–5 elevated to the containing package level. Note that the two individual element dependencies between the Planning and CropTypes packages have been aggregated to the level of the containing package [54].

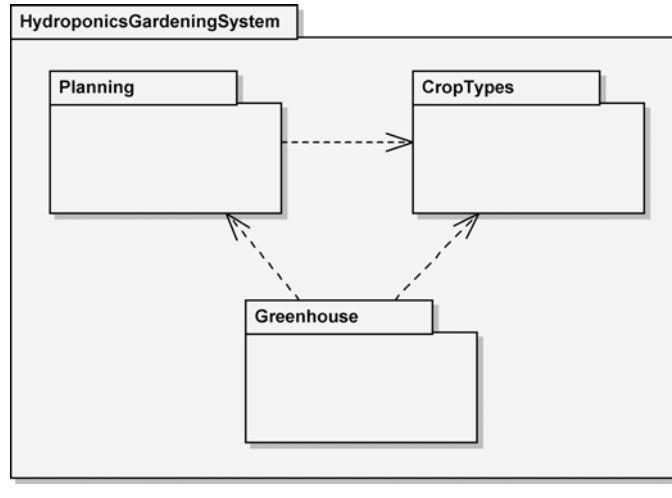

Figure 5–6 Aggregation of Contained Element Dependencies

Essentials: Package Diagrams

So far, we've discussed what could be referred to as the constituent pieces of a package diagram:

- Package notation
- Element visibility
- Dependency relationship

The package diagram is the UML 2.0 structure diagram that contains packages as the primary represented UML element and shows dependencies between the packages.

However, the package notation can be used to show the structuring and containment of many different model elements, such as classes, as shown earlier in Figures 5–4 and 5–5. It can also be used on UML diagrams that are not structure diagrams. We alluded to this earlier when we mentioned that a package can be used to organize use cases. This might be done for the sake of clarity in a very large system or to partition work. An example appears in Figure 5–7, where packages are used to group use cases of the `HydroponicsGardeningSystem` to facilitate their specification among two groups with different expertise—operations and support [55]. We'll discuss actors and use cases in depth later in this chapter.

The elements grouped in a package should typically be related in some manner, such as the subsystems within a system, use cases related to a particular aspect of

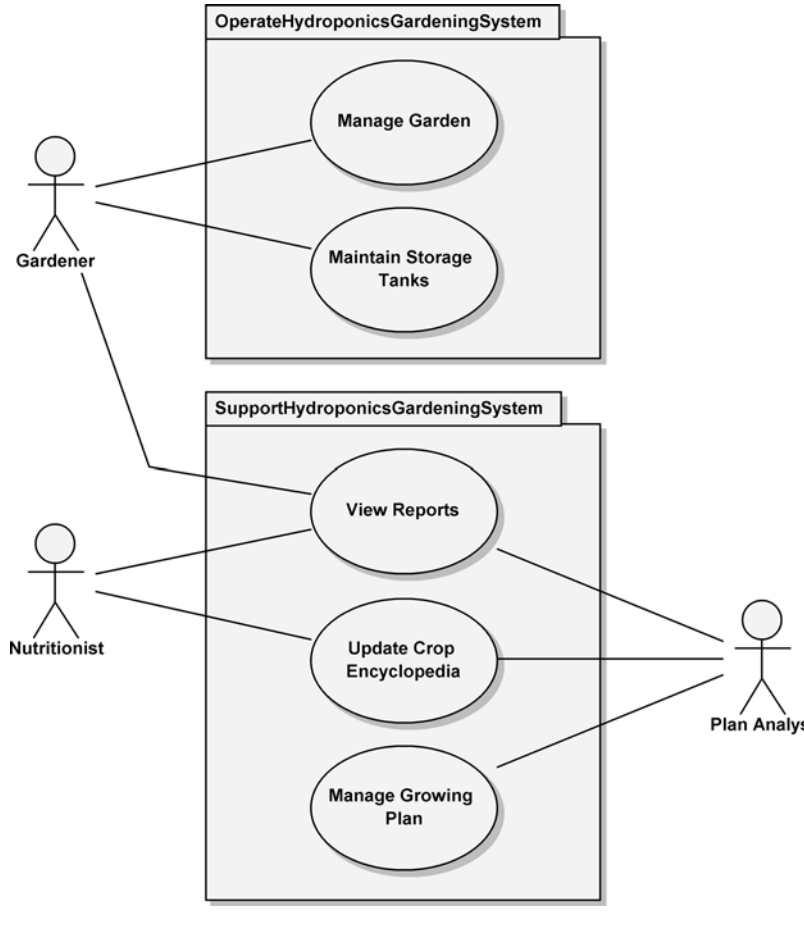

Figure 5–7 The Package Notation Used for Partitioning

the system, or classes that collaborate to provide a usable subset of system functionality [56].

What are the criteria for deciding how to package elements? There are many different ways to organize a system with packages—by architectural layer, by subsystem, by user (for use cases), and so on. Good packages are loosely coupled and highly cohesive; that is, we should see more interaction among the elements within a package and less between the packages. We should also strive not to extend generalization hierarchies or aggregations across packages. Similarly, don't break use case include or extend relationships across packages [57]—again, this will become clearer after our discussion of classes and use cases later in this chapter.

Every element contained within a namespace may be referred to with a qualified name in the format of *`package name`*`::`*`element name`*. Elements are permitted to have the same name as long as they belong to different namespaces (reside in different packages) [58, 59]. This leads us into the advanced concepts of import and access.

Advanced Concepts: Import and Access

Import and access are really two sides of the same coin—import is a public package import, whereas access is a private package import. What this really means is that in an import, other elements that have visibility into the importing package can see the imported items. But, when a package performs an access, no other elements can see those elements that have been added to the importing package's namespace. These items are private; they are not visible outside the package that performed the access [60, 61].

At this point, you're probably wondering why we would perform a package import or package access. Doing so gives us the ability to refer to the public elements of another namespace by using unqualified names; the importing package adds the names of the imported elements to its namespace. However, if any of the imported elements are of the same type and have the same name as an owned element, they are not added to the importing namespace. Similarly, if any elements imported from multiple different namespaces are of the same type and have the same name, they are not added to the importing namespace [62, 63].

The import of a package's elements can be broad or focused—all the elements or just selected ones may be imported. Look back at Figure 5–5, which shows the `PlanAnalyst` class with a dependency on the `CropEncyclopedia` class. Because the `Planning` package does not import or access the `CropTypes` package, `PlanAnalyst` must use the qualified name `HydroponicsGardeningSystem::CropTypes::CropEncyclopedia` to reference the `CropEncyclopedia` class.

To indicate that the `Planning` package imports the `CropTypes` package, a dependency is shown from `Planning` to `CropTypes` and is labeled with «`import`», for a *public* package import, as shown in Figure 5–8. This means that both `PlanAnalyst` and `PlanMetrics` can access the `CropEncyclopedia` and `CropDatabase` classes with their unqualified names. This is true for `PlanMetrics` because its namespace (`Plans` package) provides it access to the elements of outer packages within which it is nested.

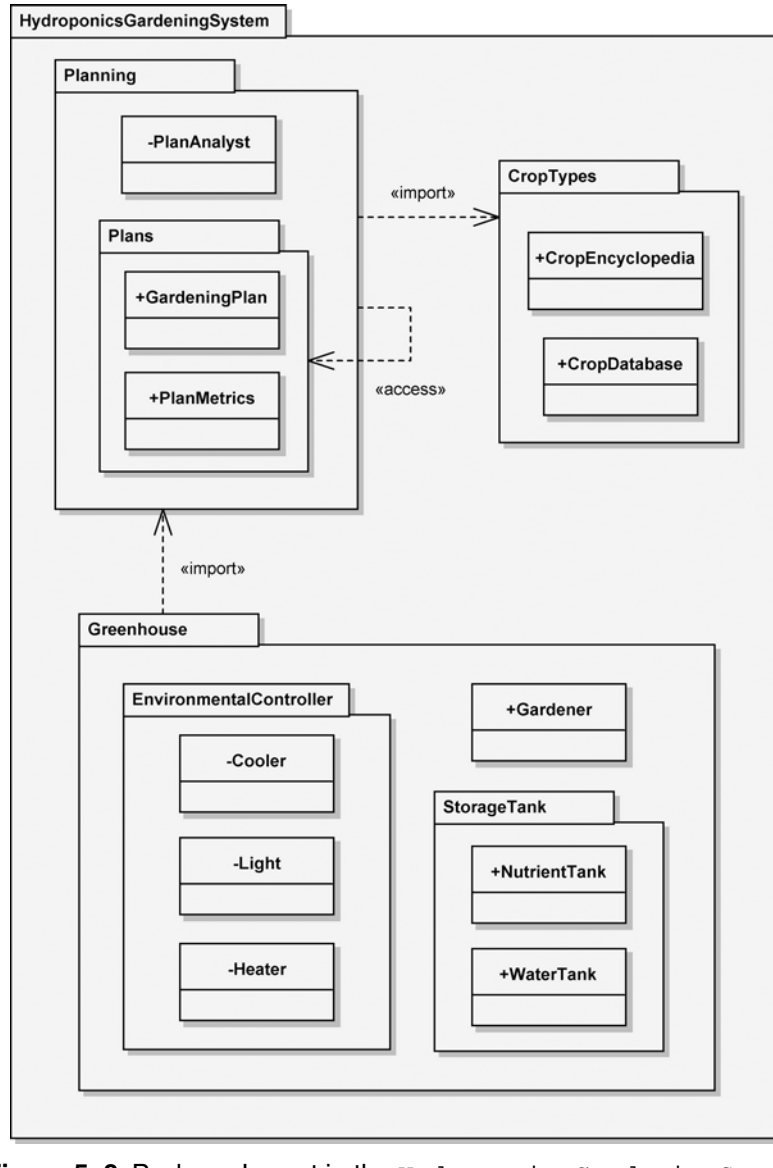

Figure 5–8 Package Import in the `HydroponicsGardeningSystem`

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.