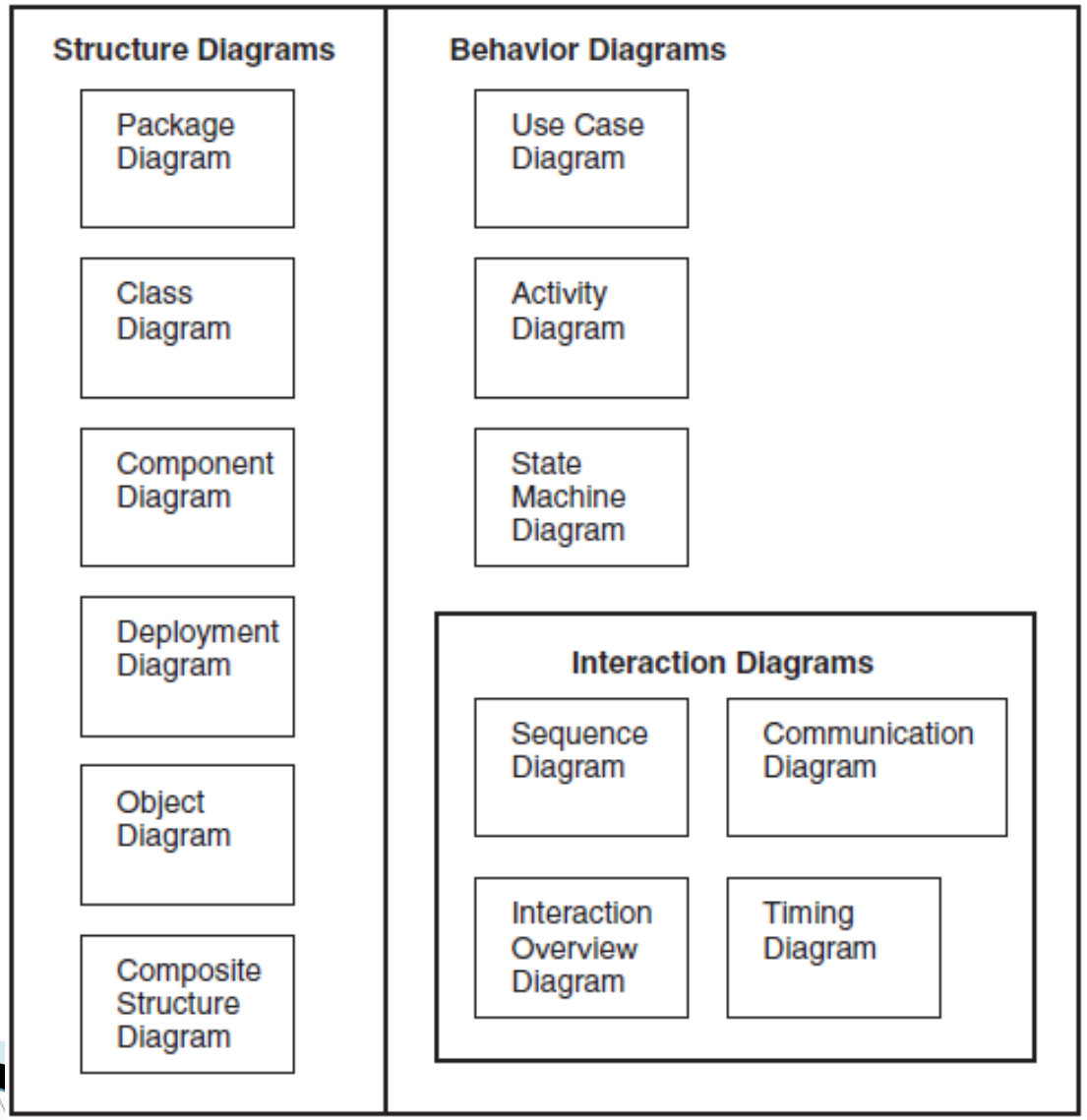
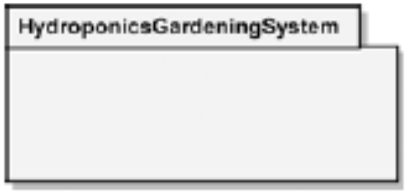
Unified Modeling Language Version 2.0

Notation – UML 2.0



Package Diagrams

- Packages, visibility and their dependencies



Contains packages as the primary represented UML element and shows dependencies between the packages.

Figure 5–2 The Package Notation for `HydroponicsGardeningSystem`

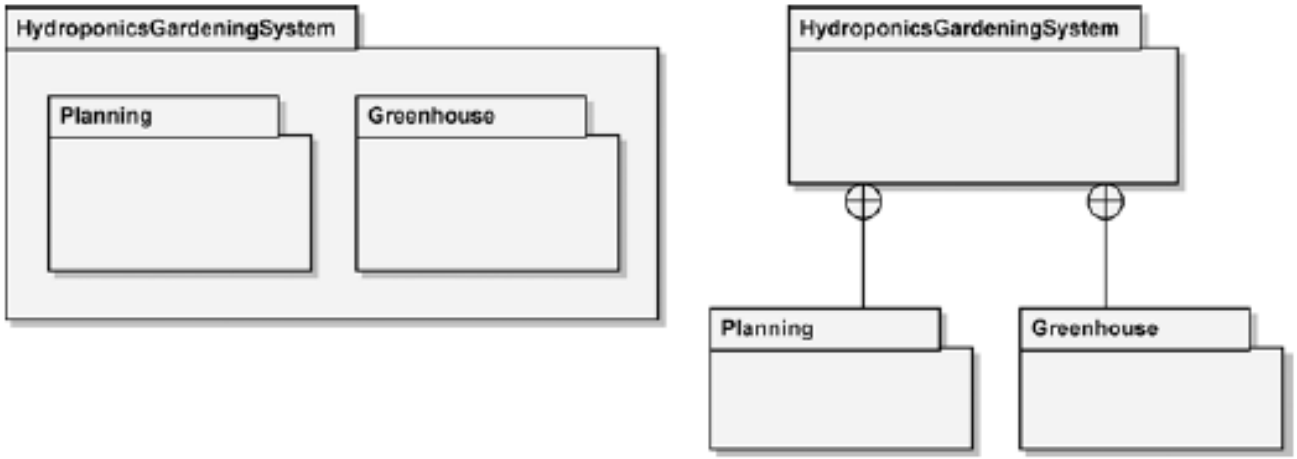


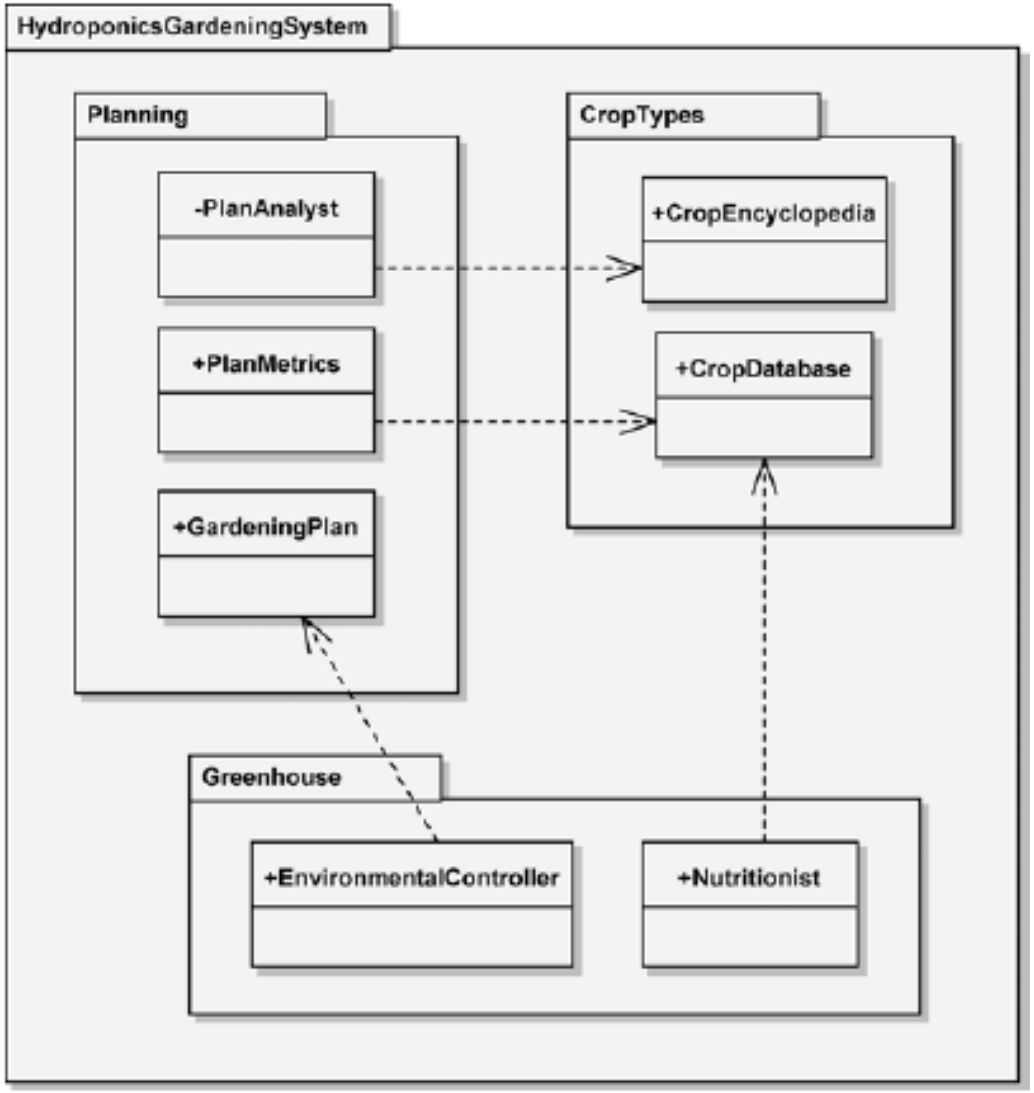
Figure 5–3 The Package Notation for Contained Elements

Package Diagrams

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

- 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

Package Diagrams



- A dependency shows that an element is dependent on another element as it fulfills its responsibilities within the system.

- Can include the type of dependency within angular guillemets. Ex: <<import>>

Package Diagrams

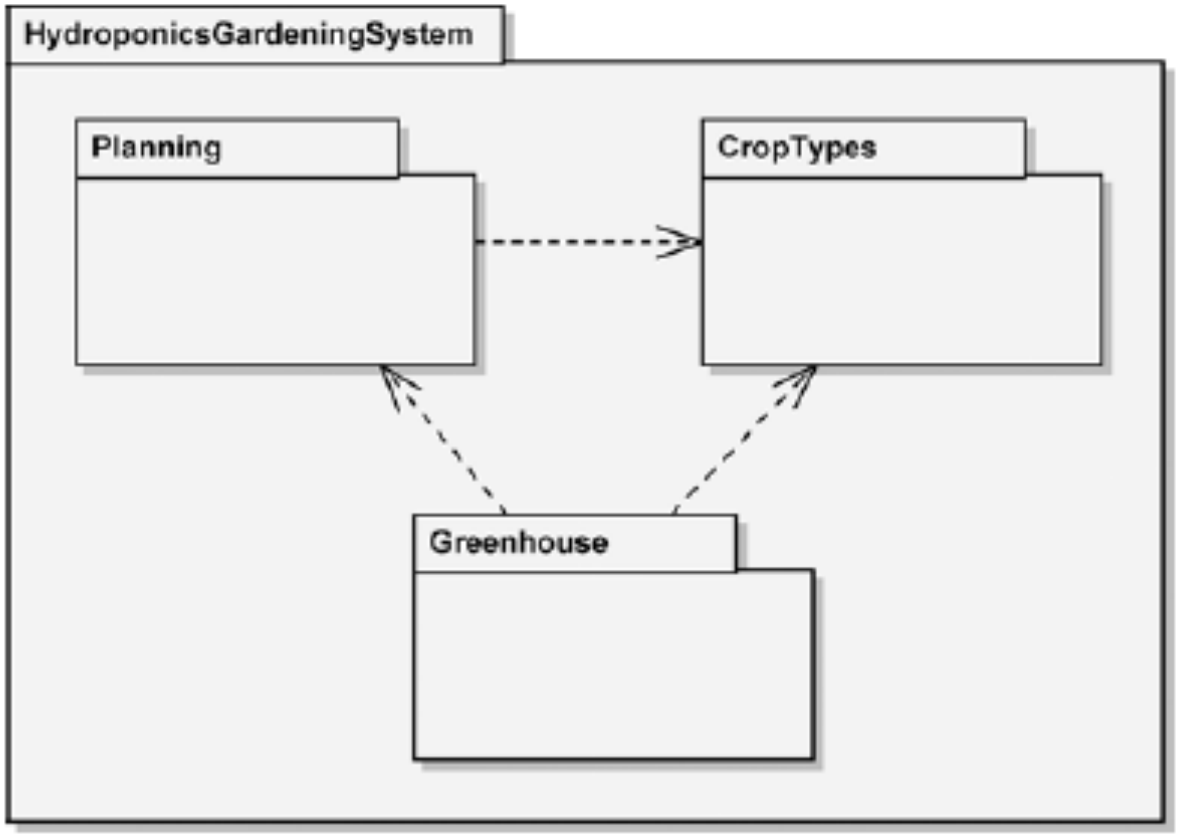
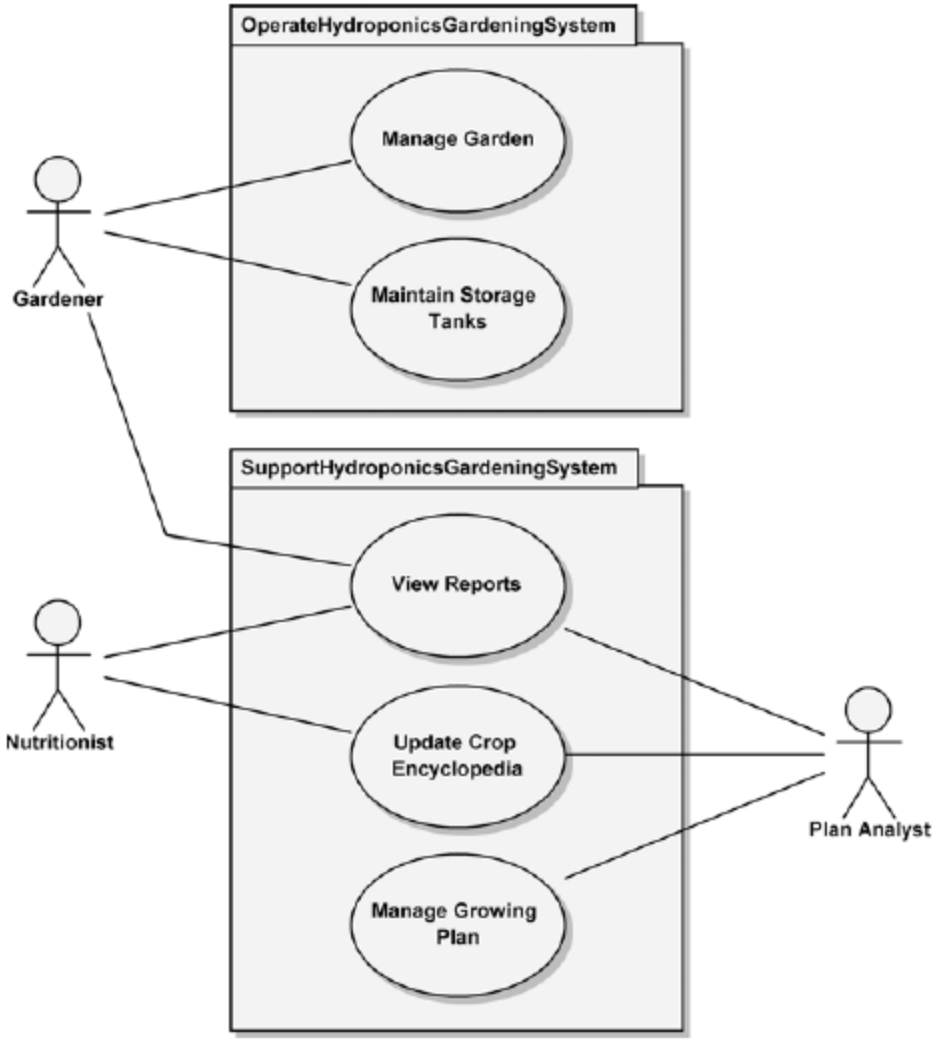


Figure 5–6 Aggregation of Contained Element Dependencies

Package Diagrams

- Package can be used to organize use cases
  - sake of clarity in a very large system or to partition work



Package Diagrams – Best Practices

- The system, or classes that collaborate to provide a usable subset of system functionality
- Ways to organize system with packages
  - by architectural layer, by subsystem, by user (for use cases), and so on
- Good Packages are loosely coupled and highly cohesive
  - more interaction among the elements within a package and less between the packages
- Not to extend generalization hierarchies or aggregations across packages
- Don't break use case include or extend relationships across packages

Component Diagrams

- 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.
- 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

Component Diagrams

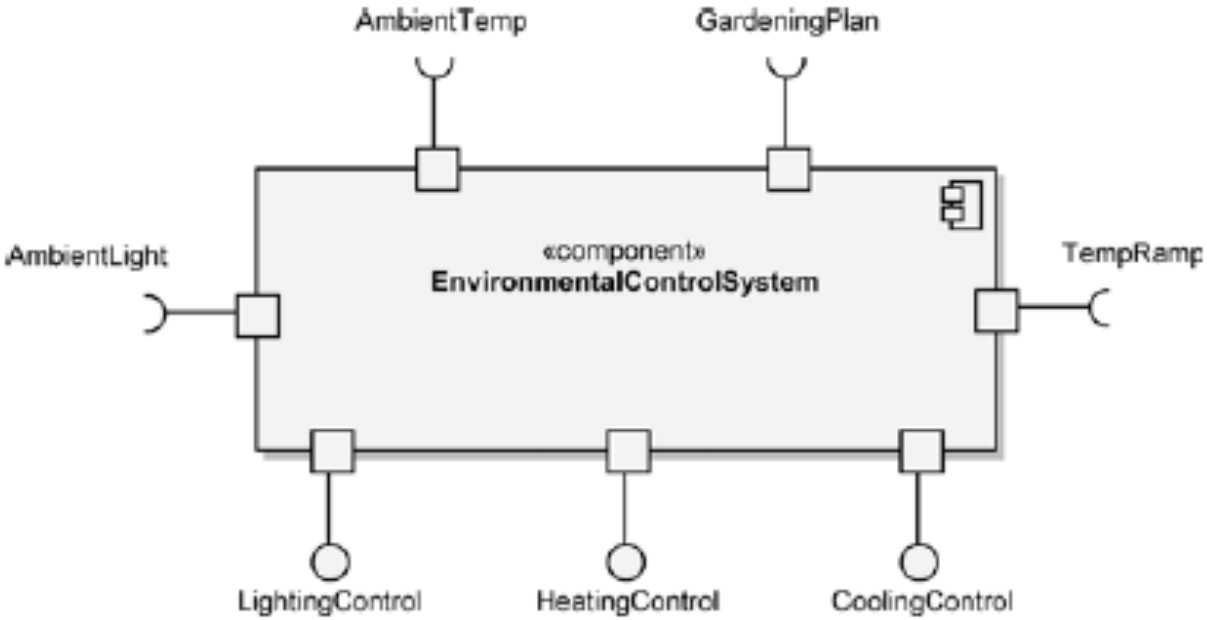


Figure 5–9 The Component Notation for `EnvironmentalControlSystem`

Component Diagrams

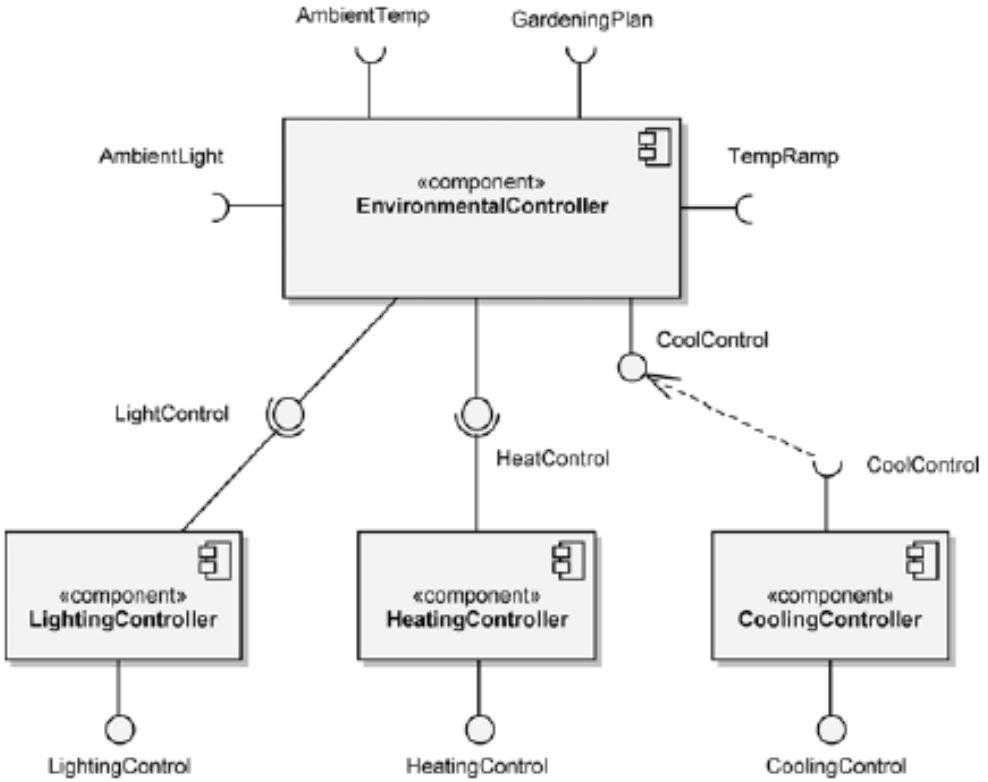


Figure 5–11 The Component Diagram for `EnvironmentalControlSystem`

Component Diagrams

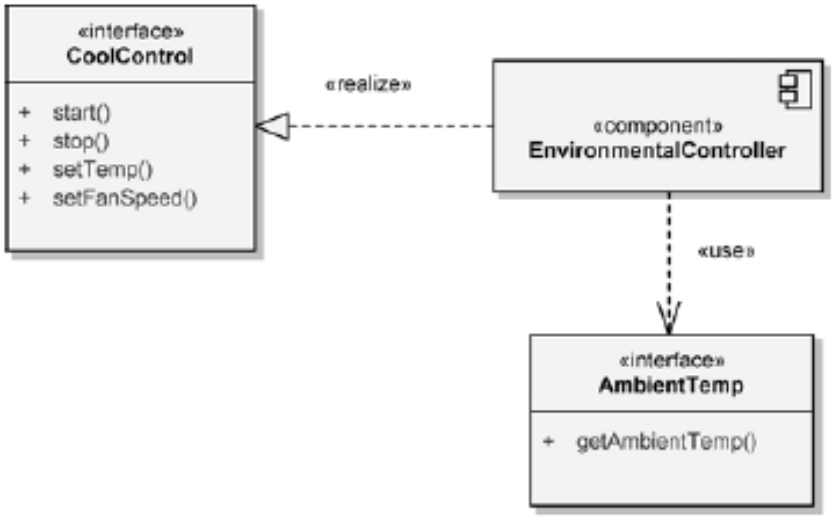


Figure 5–12 The Specification of Two Interfaces for `EnvironmentalController`

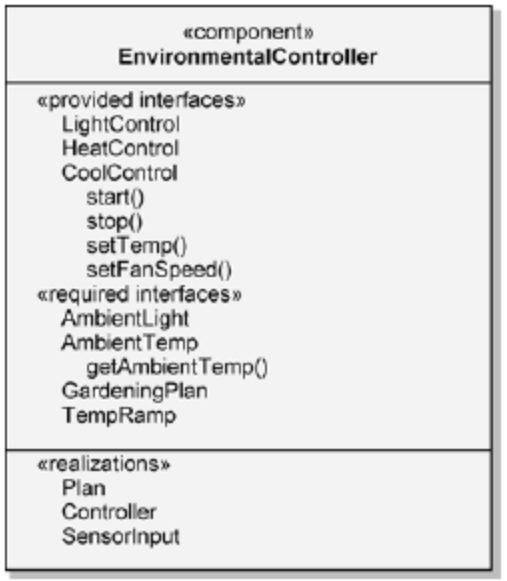


Figure 5–13 An Alternate Notation for `EnvironmentalController`'s Interfaces and Realizations

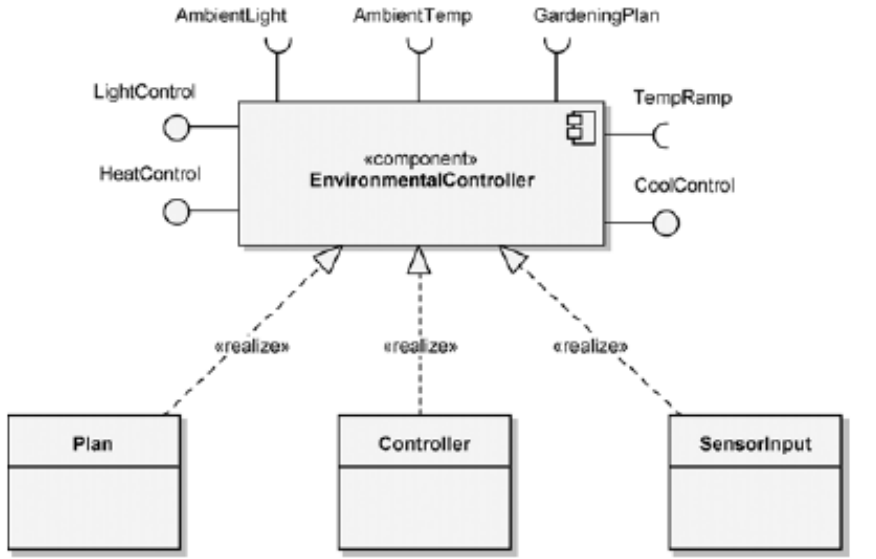


Figure 5–14 The Realization Dependencies for `EnvironmentalController`

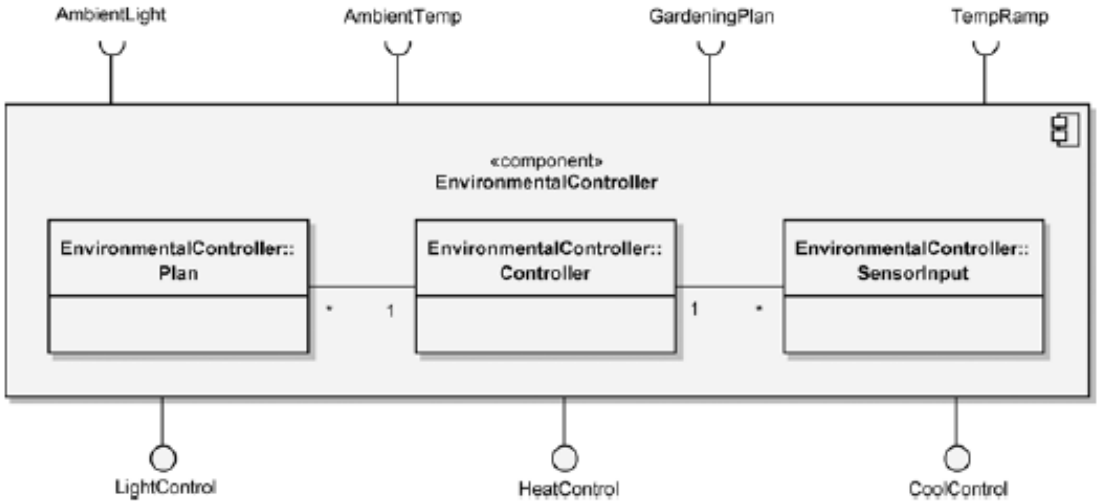


Figure 5–15 The Containment Representation of `EnvironmentalController`'s Realization

Component Diagrams – Internal Structure

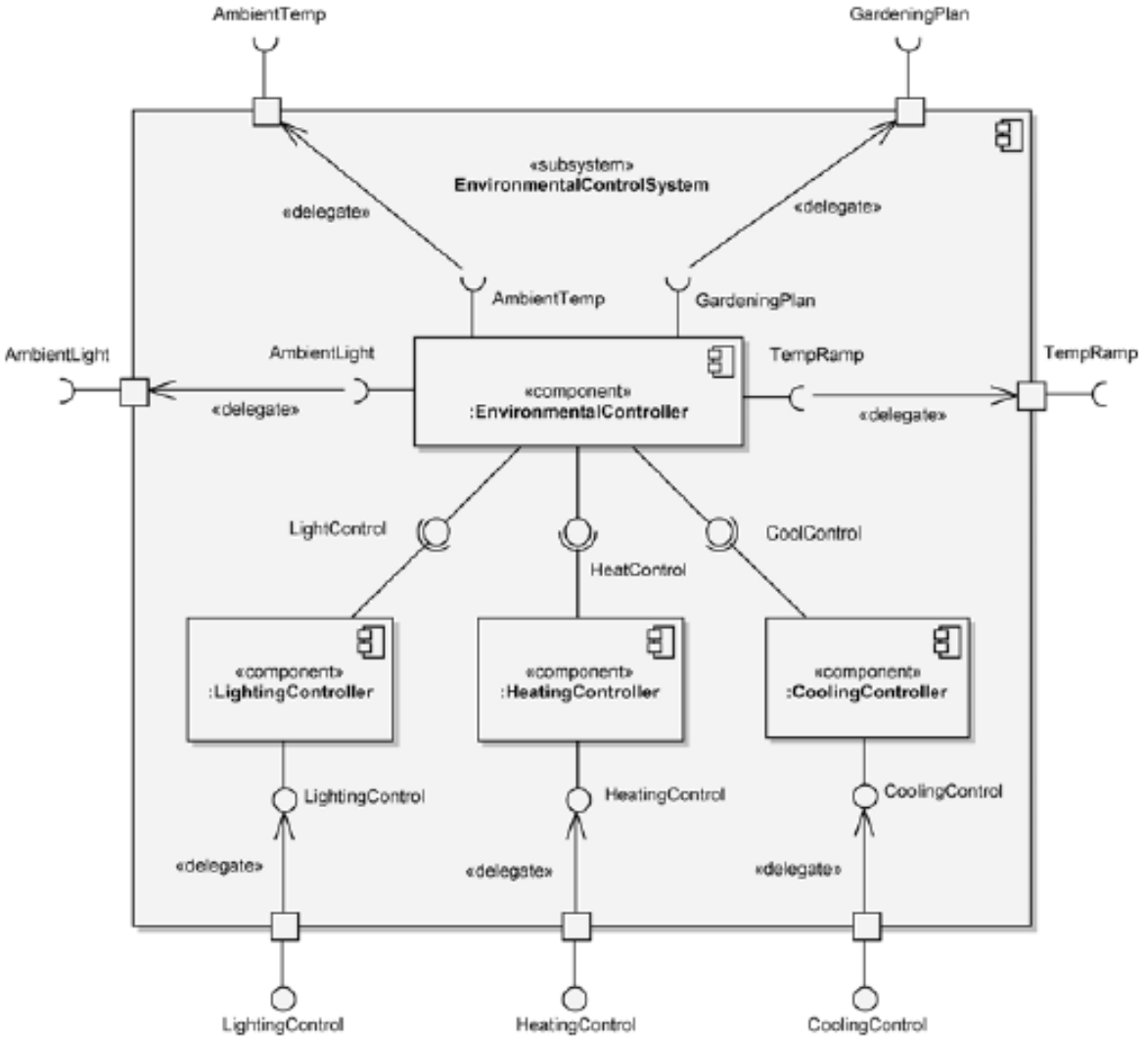


Figure 5–16 The Internal Structure of `EnvironmentalControlSystem`

Component Diagrams

- In practice, a large system has one top-level component diagram, consisting of the subsystems at the highest level of abstraction.

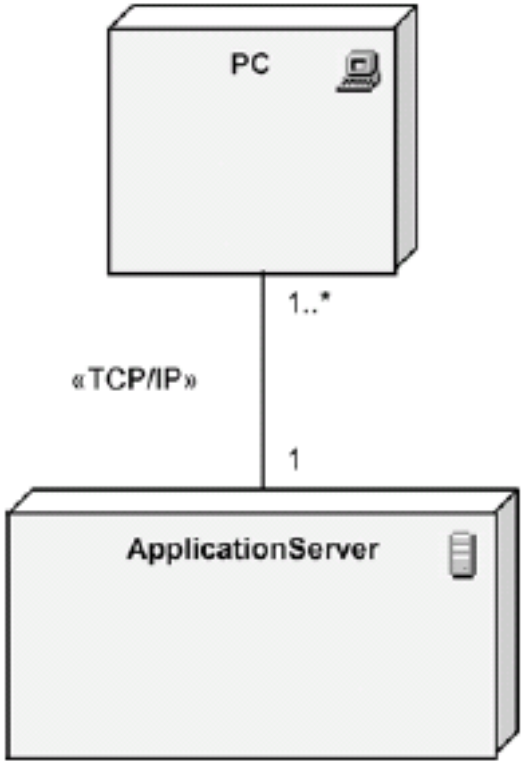
Deployment Diagrams

- 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.

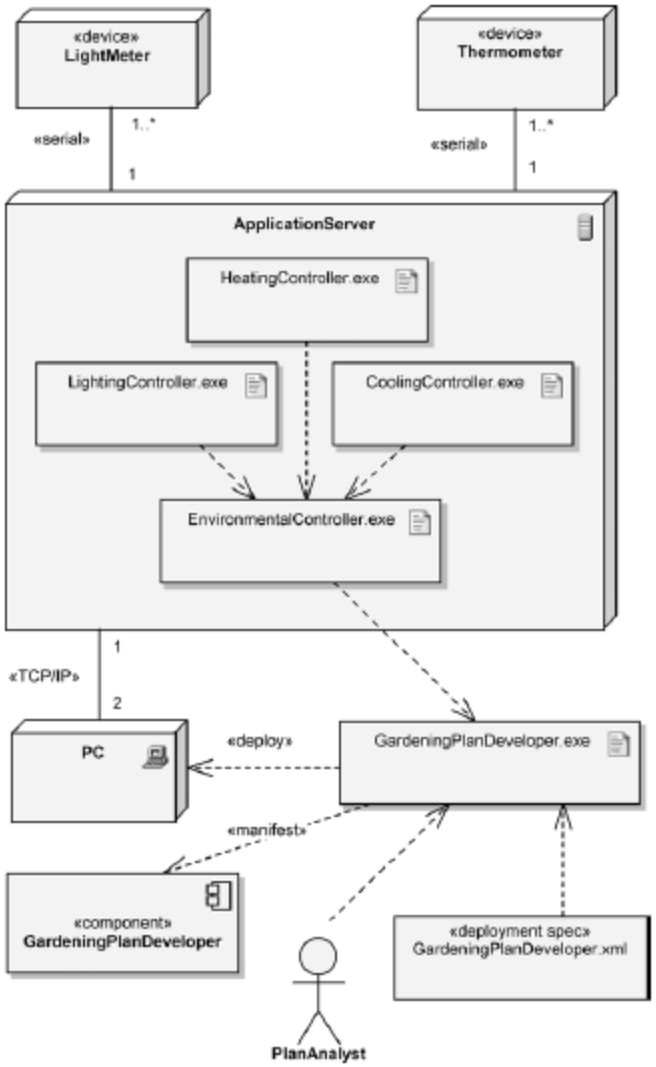
Deployment Diagrams - Node

- Computational resource, typically containing memory and processing, on which artifacts are deployed for execution.
- Two types of nodes:
  - Devices – Piece of hardware providing computational capabilities Ex. Computer, Modem
  - Execution environments – Software providing deployment of specific types of executing artifacts Ex. <<database>>, <<JEE Server>>
- Communication path between nodes
  - May contain keyword like <<http>>, <<TCP/IP>>

Deployment Diagrams - Node



Deployment Diagrams



Use Case Diagrams

- Depicts the context of the system to be built and the functionality provided by that system.
- Depict who (or what) interacts with the system.
- Shows what the outside world wants the system to do.

Actors

- Actors are entities that interface with the system.
- They can be people or other systems.
- Actors, which are external to the system they are using, are depicted as stylized stick figures.
- Consider the roles the actors play.

Actors

- Actors are entities that interface with the system.
- They can be people or other systems.
- Actors, which are external to the system they are using, are depicted as stylized stick figures.
- Consider the roles the actors play.

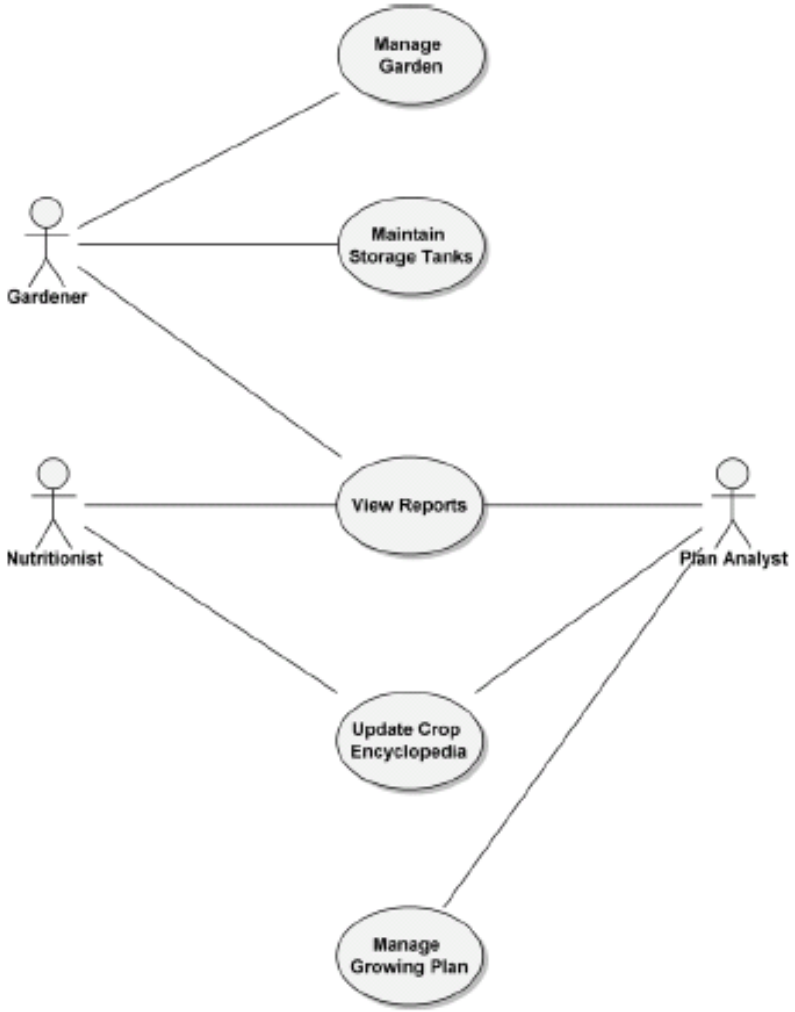
Use Case Diagrams - Actor

Use Cases

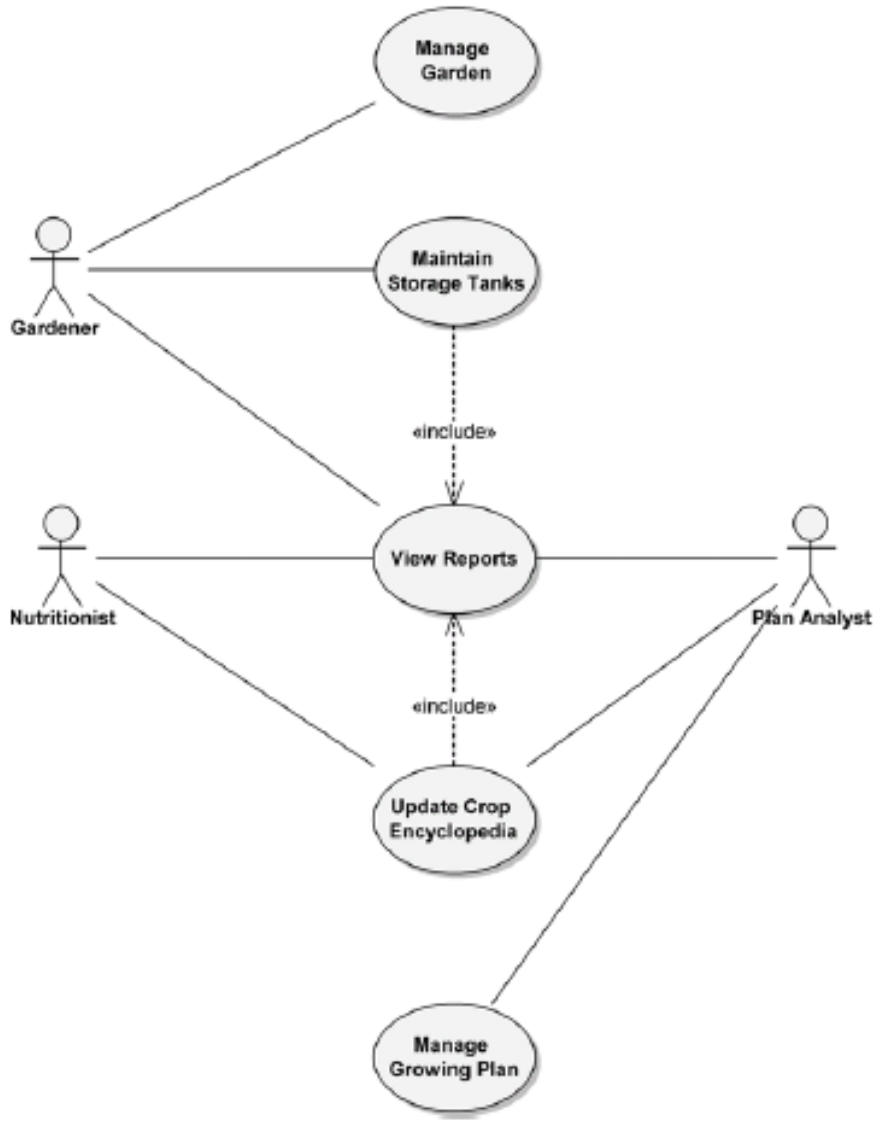
- A use case is a specific way of using the system by using some part of the functionality.
- A use case is thus a special sequence of related transactions performed by an actor and the system in a dialogue
- Each use case is a complete course of events in the system from a actor's perspective that provides value to the actor.

Use Case Diagram – Use Cases

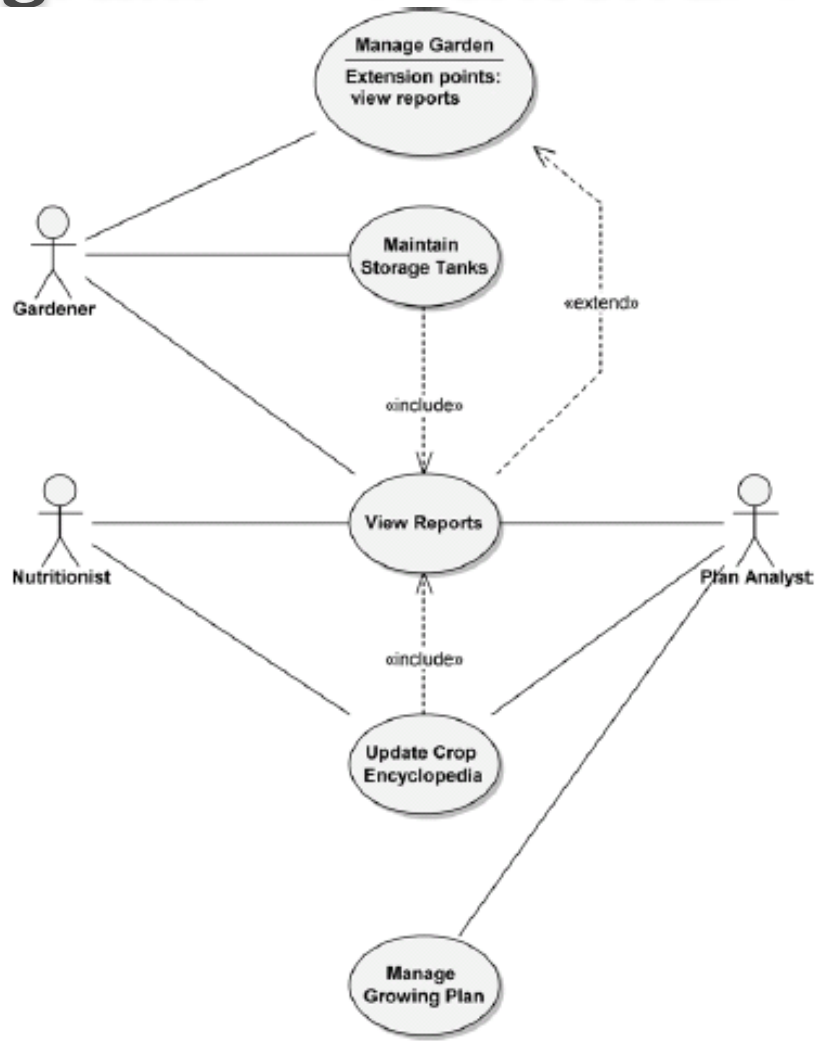
Use Case Diagram - Basic



Use Case Diagram - <<include>> (mandatory)



Use Case Diagram - <<extend>> (optional)



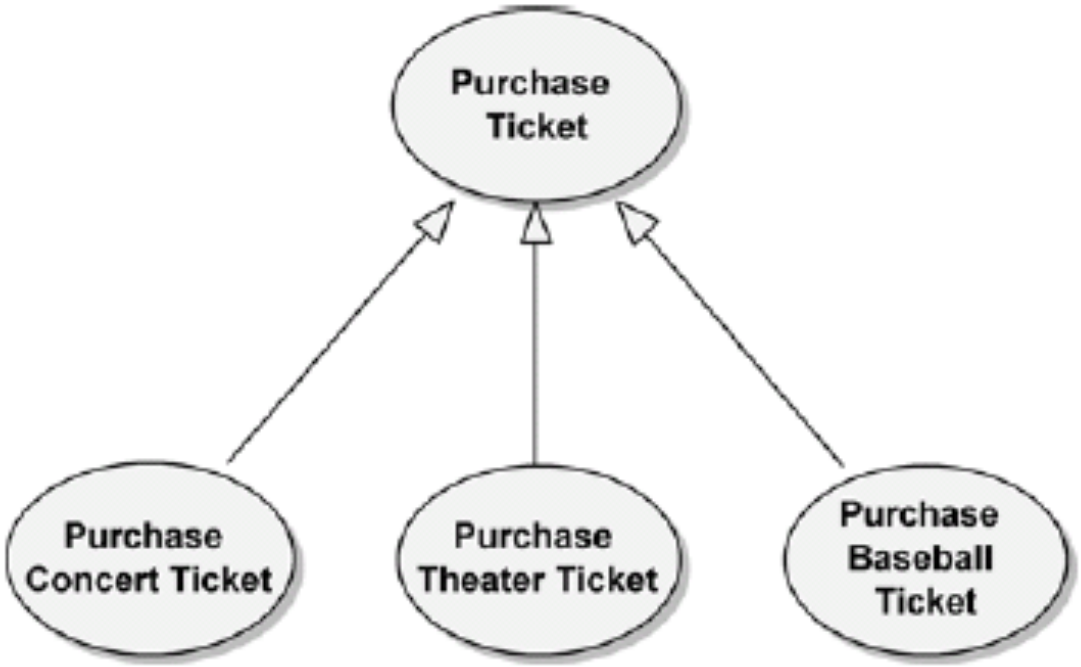
Dangers of <<include>> and <<extend>>

- Included and Extended Basic use case principles should not be violated
  - a use case represents a *complete flow* of activity of *what* the actor wants your system to do *from the actor's point of view that provides value to the actor*
- Should not use to "functionally decompose" use cases

Differences between <<include>> and <<extend>>

	Included Use Case	Extending Use Case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

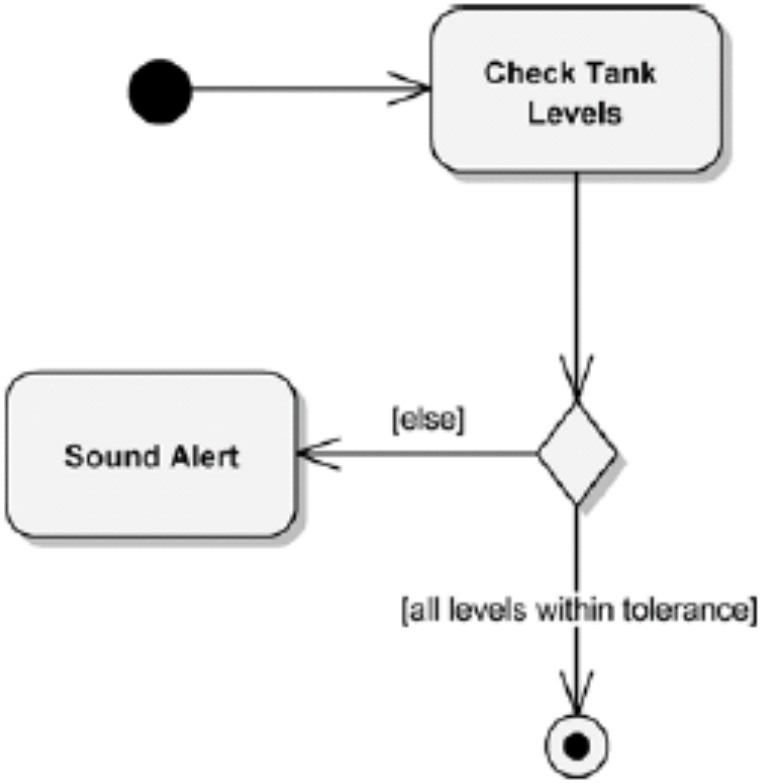
Use Case Generalization



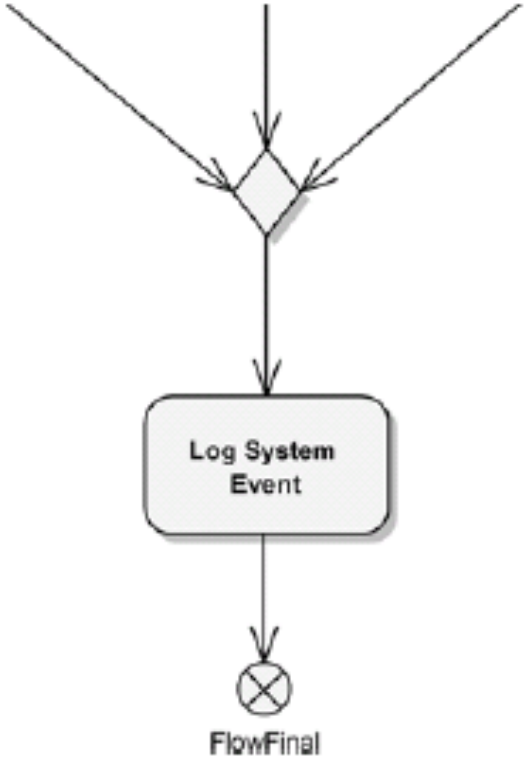
Activity Diagrams

- Visual depictions of the flow of activities, whether in a system, business, workflow, or other process.
- These diagrams focus on the activities that are performed and who (or what) is responsible for the performance of those activities.
- Elements: Action, Control (Initial/Final (activity final and flow final), Decision/Merge, Fork/Join)

Activity Diagrams



Activity Diagrams – Merge, Flow final

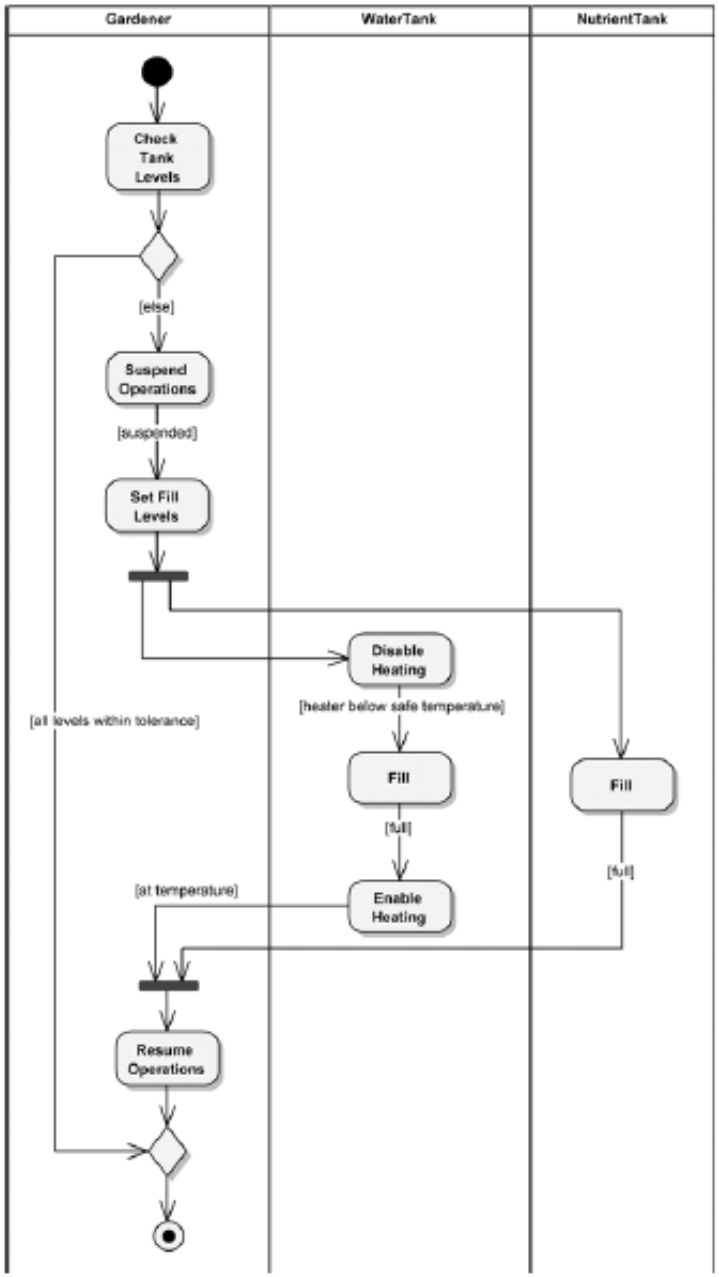


Activity Diagrams - Partitions

- Grouping of elements in activity diagram
- Indicate where the responsibility lies for performing specific activities.
  - Business model - partitions may be business units, divisions, or organizations
  - Systems model - partitions may be other systems or subsystems
  - Application model - partitions may be objects in the application

Activity Diagrams

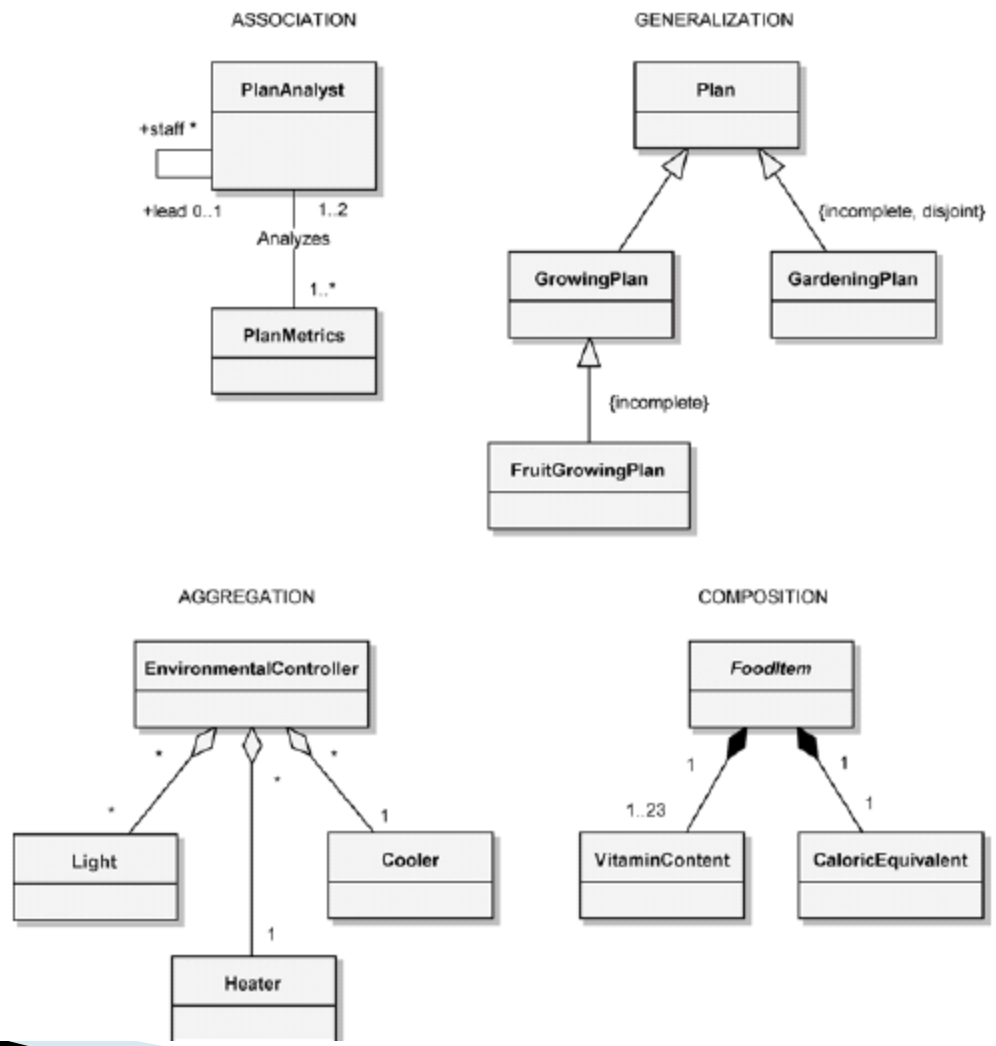
- Fork – Single flow forked into multiple concurrent flows
- Join – All incoming flows to be completed to proceed with the single outgoing flow



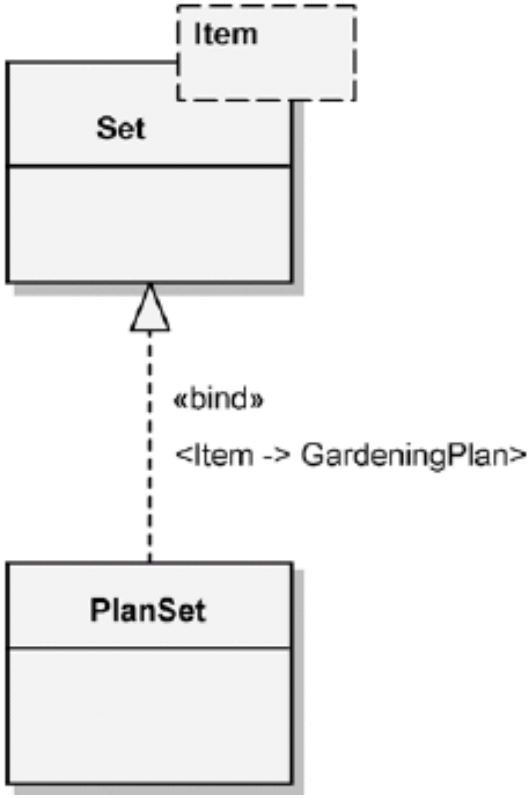
Class Diagrams

- Shows the existence of classes and their relationships in the logical view of a system.
- A single class diagram represents a view of the class structure of a system.
- During analysis, we use class diagrams to indicate the common roles and responsibilities of the entities that provide the system's behavior.
- During design, we use class diagrams to capture the structure of the classes that form the system's architecture.
- The two essential elements of a class diagram are classes and their basic relationships.

Class Diagrams - Relationships



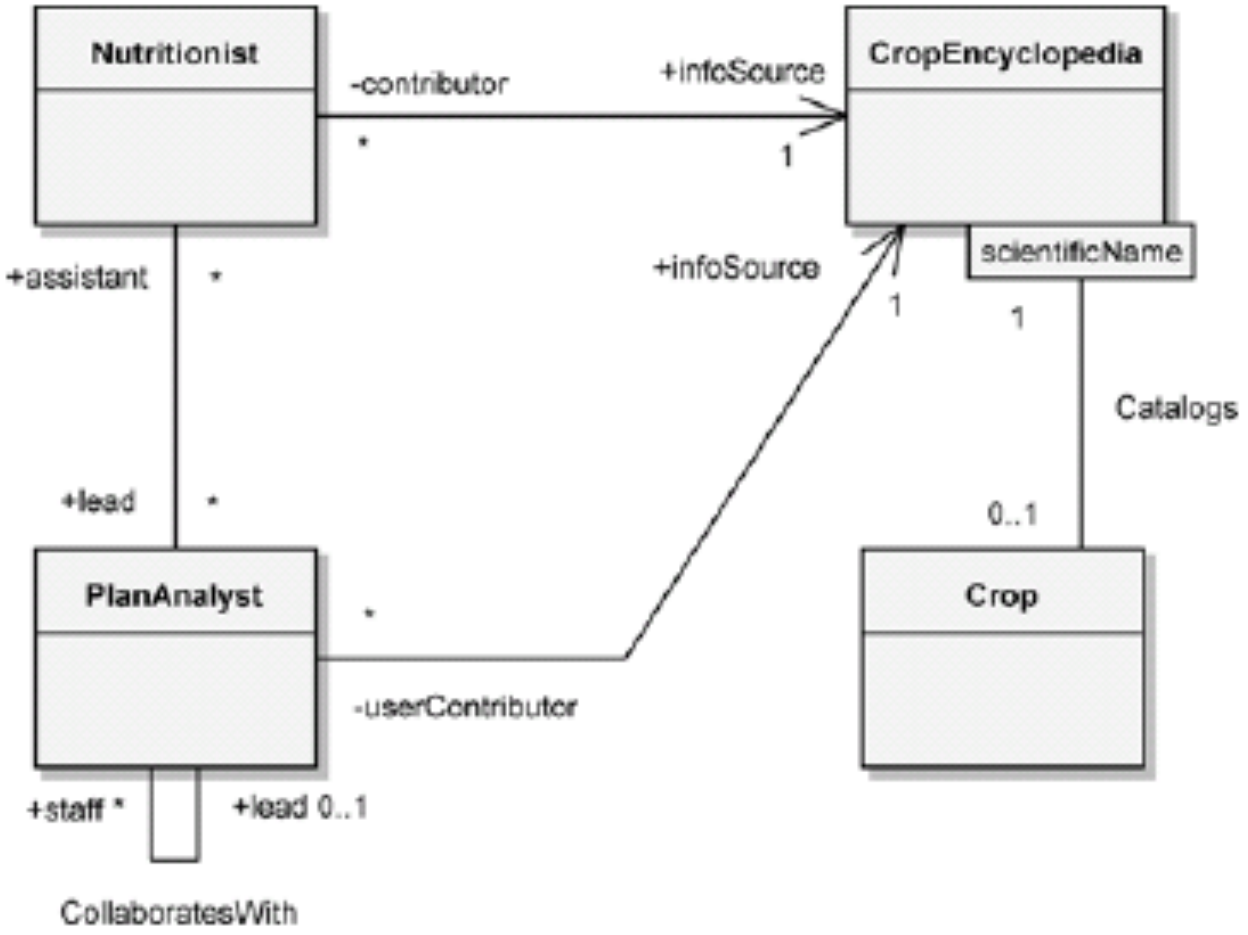
Class Diagram – Parameterized Class



Visibility

- Public (+) Visible to any element that can see the class
- Protected (#) Visible to other elements within the class and to subclasses
- Private (-) Visible to other elements within the class
- Package (~) Visible to elements within the same package

Association End Names and Qualifiers

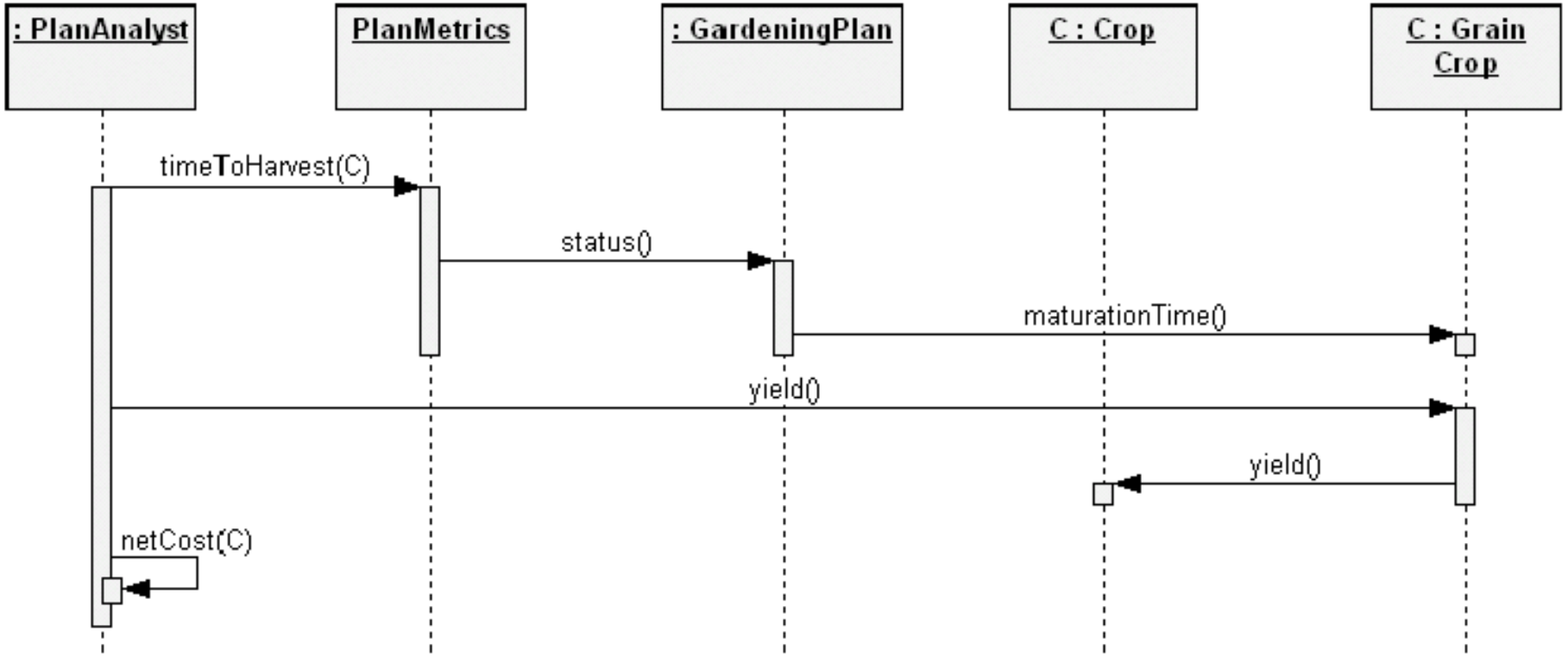


Sequence Diagrams

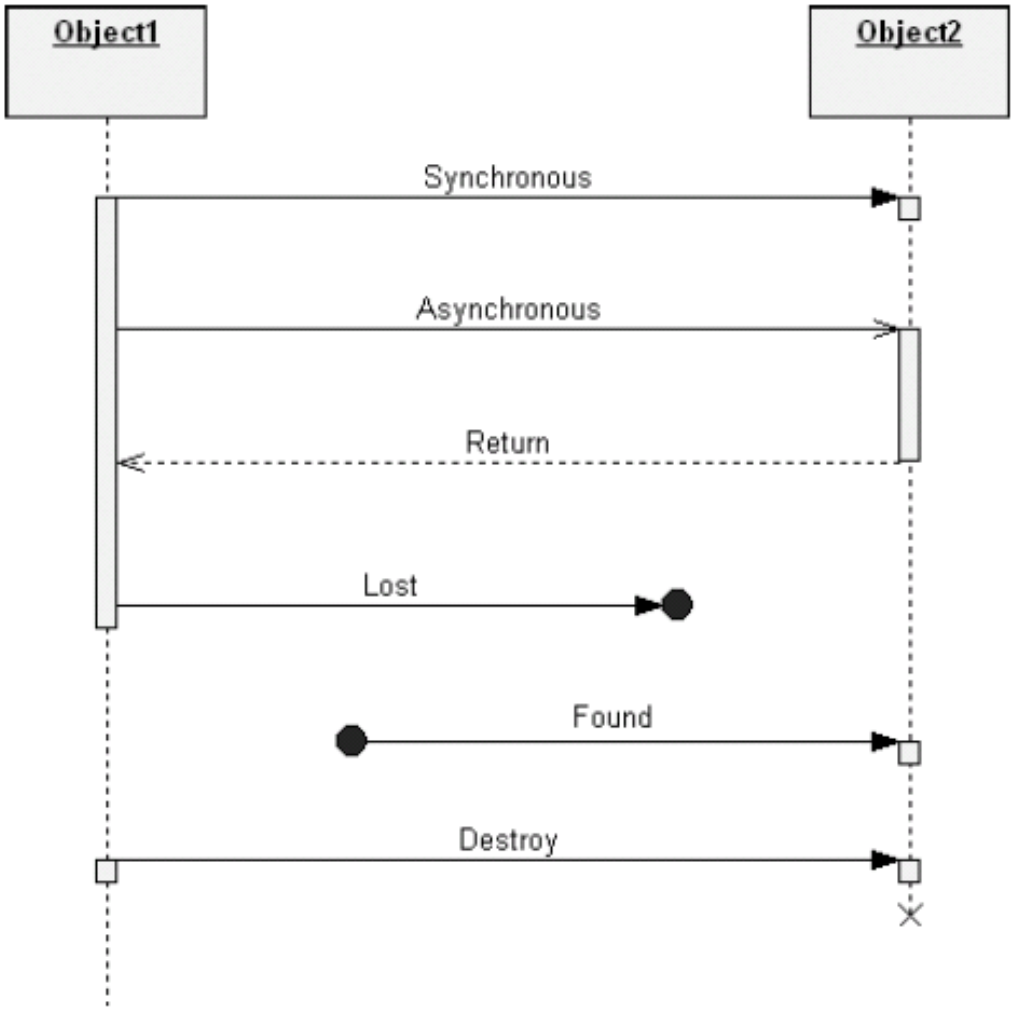
- A sequence diagram is used to trace the execution of a scenario in the same context as a communication diagram.

- Elements – Objects, Interactions, Lifelines, Messages

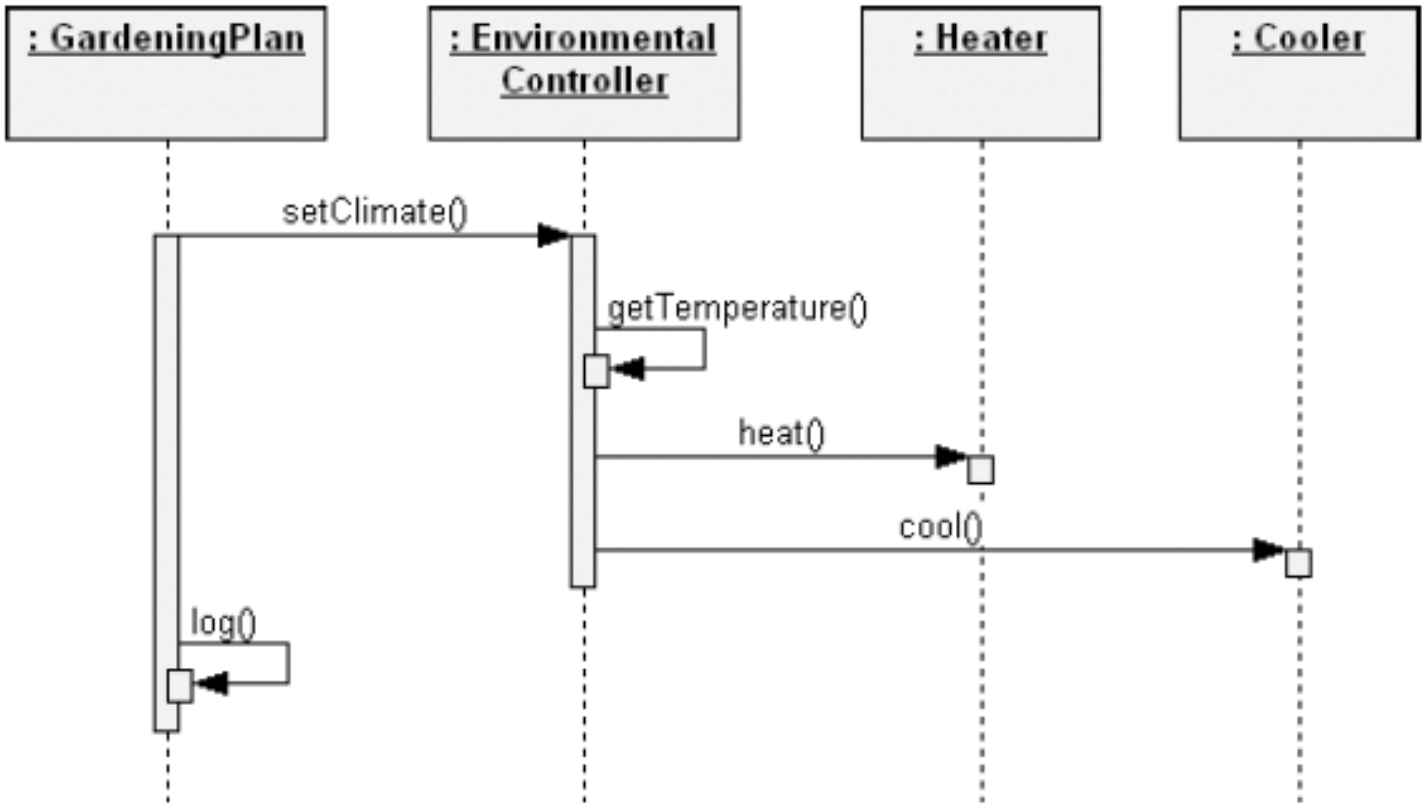
Sequence Diagram



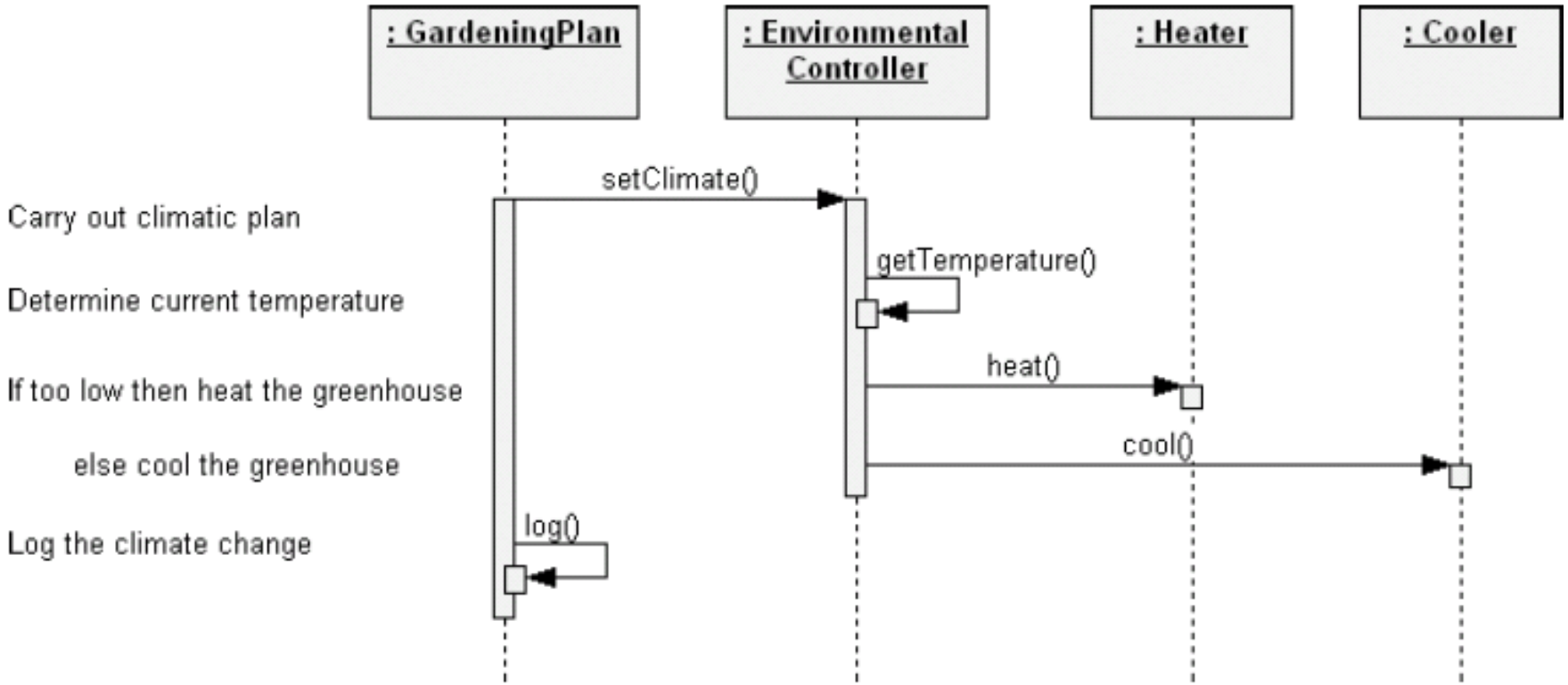
Notations for Type of Messages



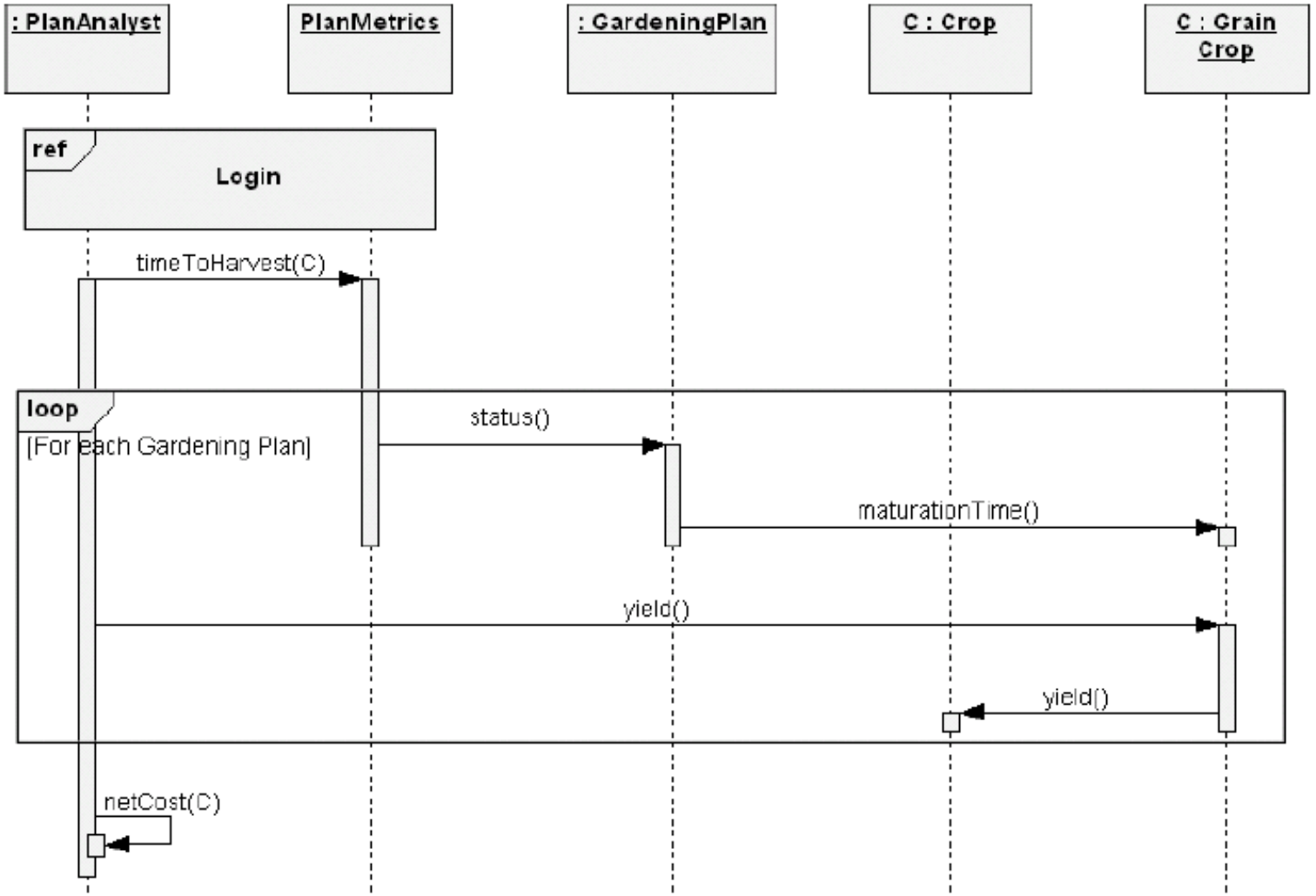
Execution Specification



Scripts in Sequence Diagram



Interaction Use

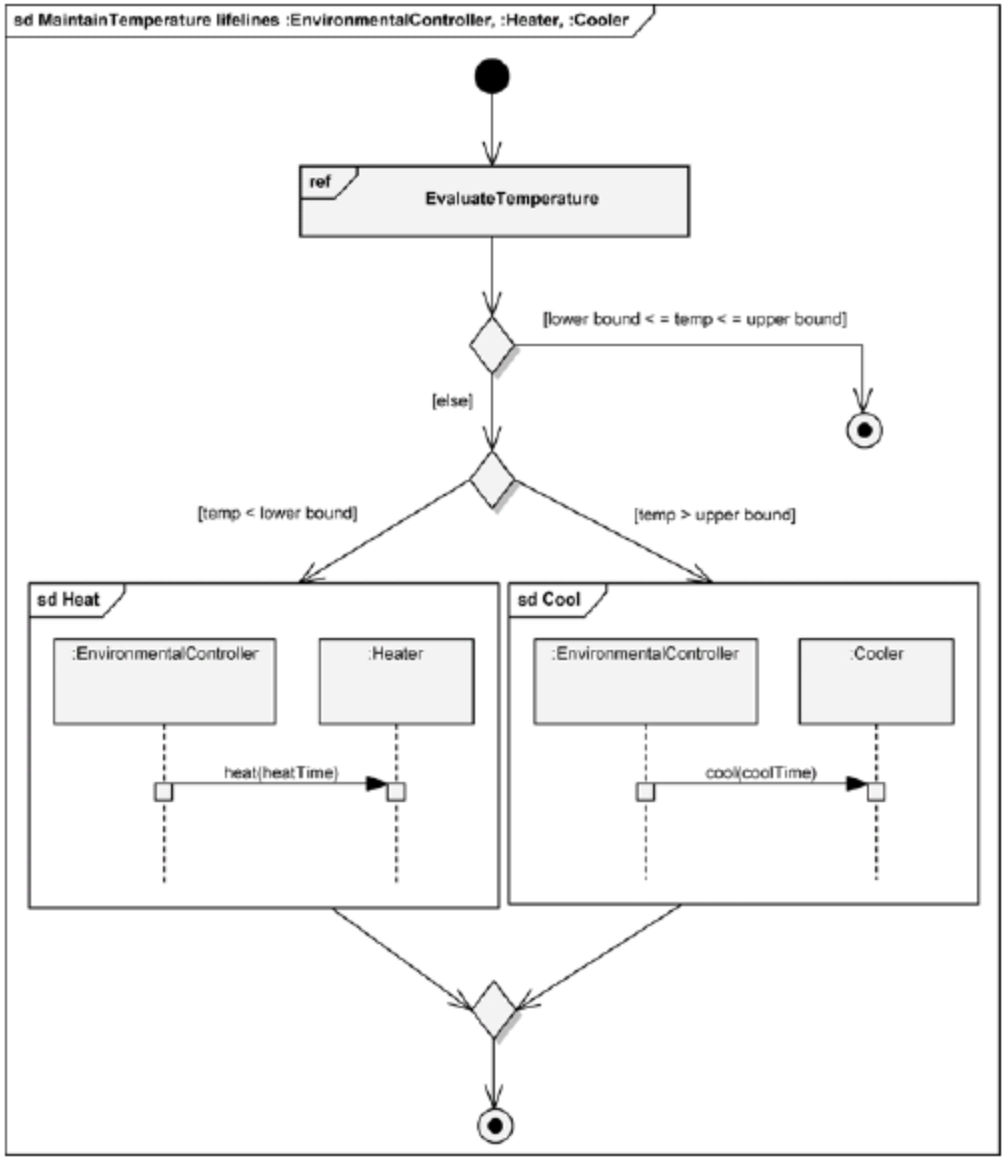


Interaction Overview Diagrams

- Combination of activity diagrams and interaction diagrams that are intended to provide an overview of the flow of control between interaction diagram elements

- Elements – Frames, Flow of control elements, interaction diagram elements

Interaction Overview Diagrams



State Machine Diagrams

- State machines are well known in industries that use real-time processing.
- State machine diagrams are used to design and understand time-critical systems, in which the consequences of improper timing are severe
  - Medical devices
  - financial trading systems
  - satellite command and control systems
  - weapon systems
- how systems behave in reaction to key events.

State Machine Diagrams

- Expresses behavior as a progression through a series of states, triggered by events, and the related actions that may occur.
  - Also known as behavioral state machines.
- Used to describe the behavior of individual objects.
- Also used to describe the behavior of larger elements of any system.
- State machine diagrams are cousins to activity diagrams.
  - Focus is on the states and transitions between those states versus the flow of activities

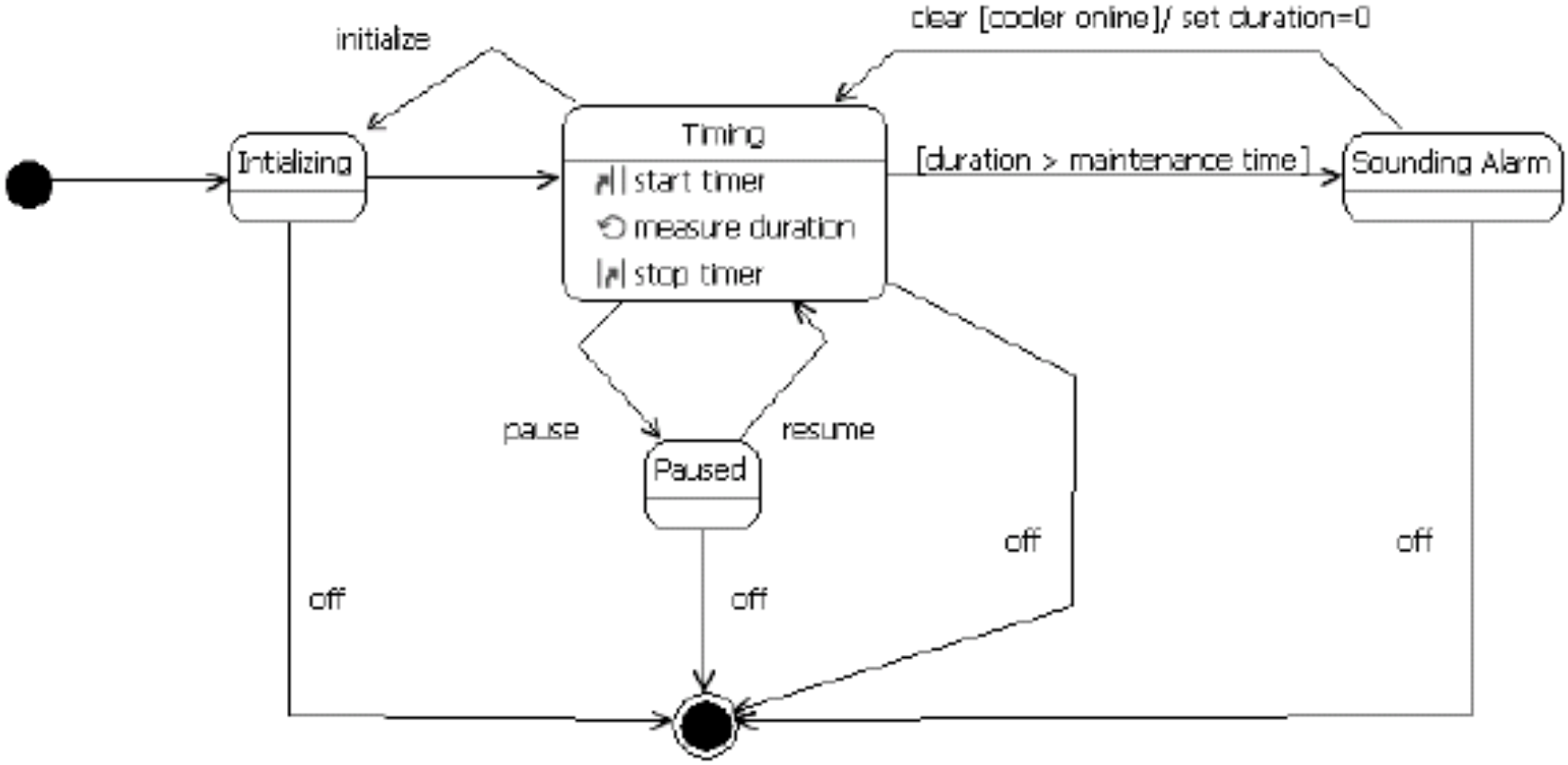
State Machine Diagrams

- We supply state machine diagrams only for those classes that exhibit such behavior.
- To show the event-ordered behavior of the system as a whole.
- During analysis, we may use state machine diagrams to indicate the dynamic behavior of the system.
- During design, we use state machine diagrams to capture the dynamic behavior of individual classes or of collaborations of classes.
- Elements - states and state transitions.

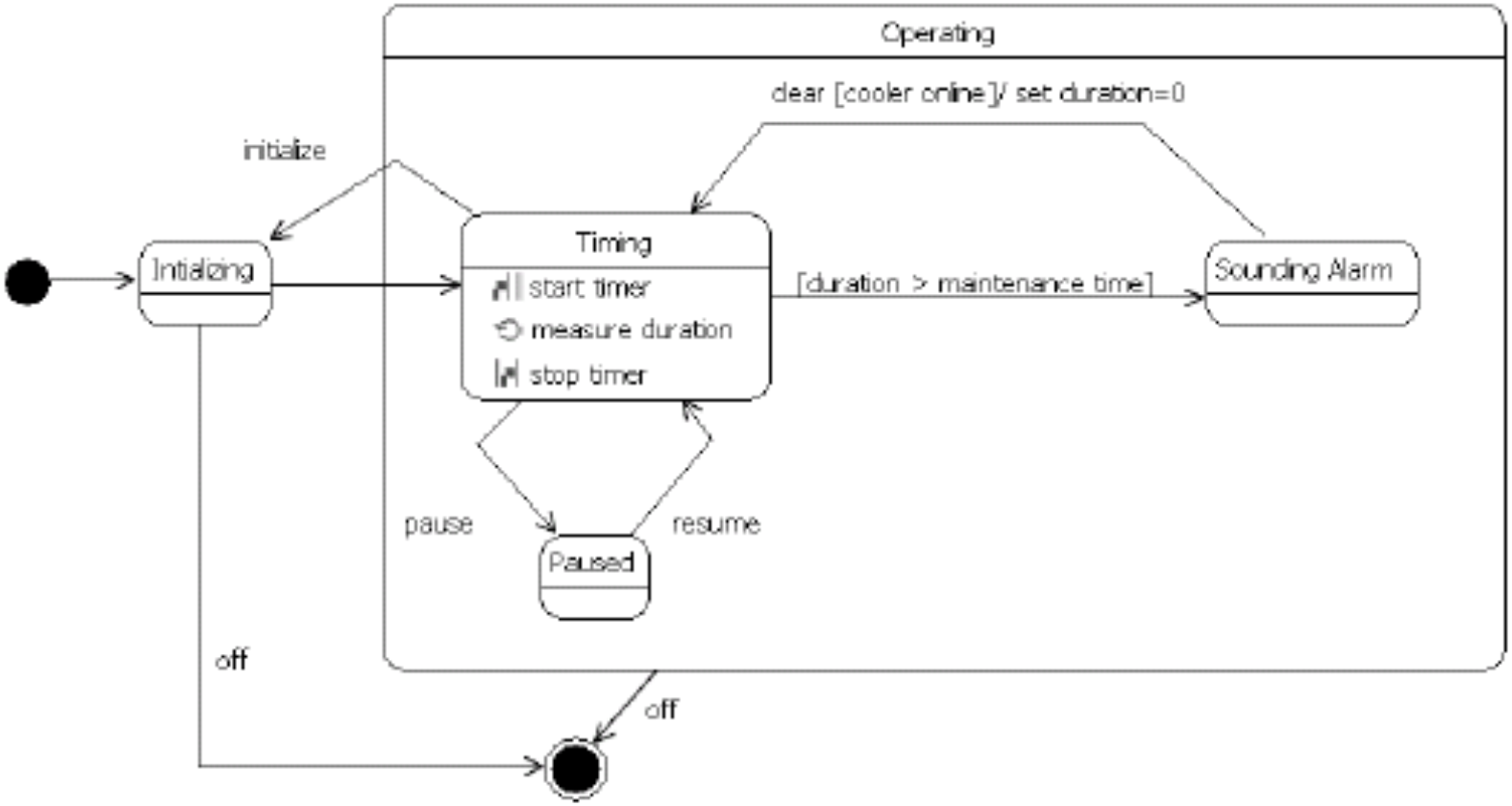
State Machine Diagrams

- We supply state machine diagrams only for those classes that exhibit such behavior.
- To show the event-ordered behavior of the system as a whole.
- During analysis, we may use state machine diagrams to indicate the dynamic behavior of the system.
- During design, we use state machine diagrams to capture the dynamic behavior of individual classes or of collaborations of classes.
- Elements - states and state transitions.

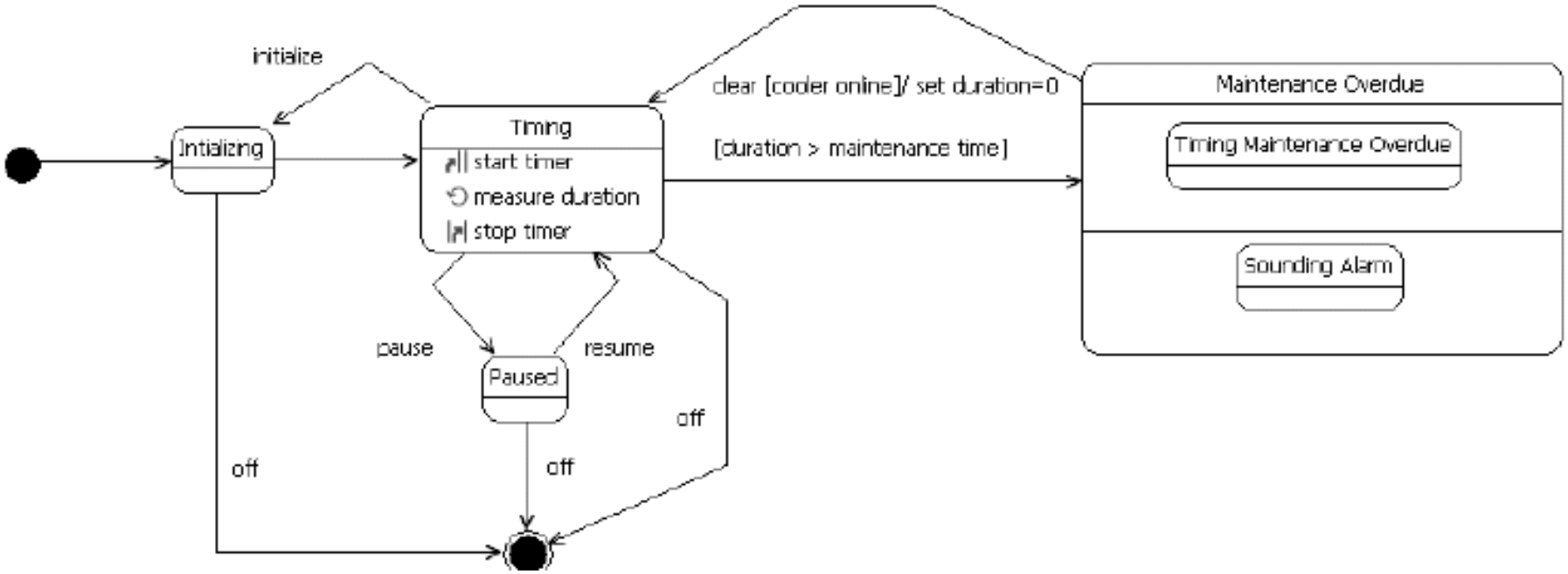
State Machine Diagram – State, transitions, events, condition, effect



State Machine Diagram – Composite (region) and Nested States



State Machine Diagram – Composite with Concurrent

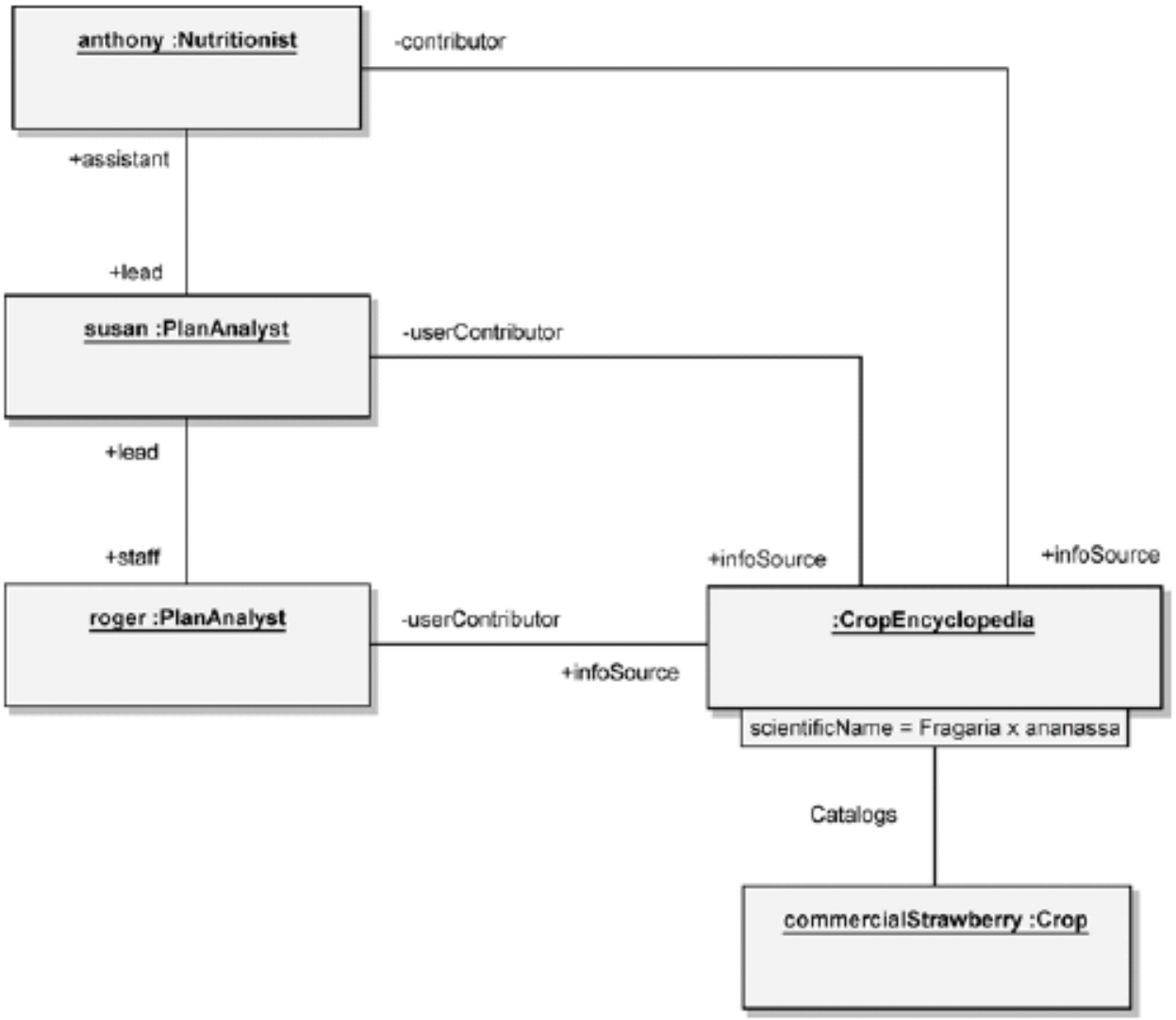


Object Diagrams

- An object diagram shows the existence of objects and their relationships in the logical design of a system.

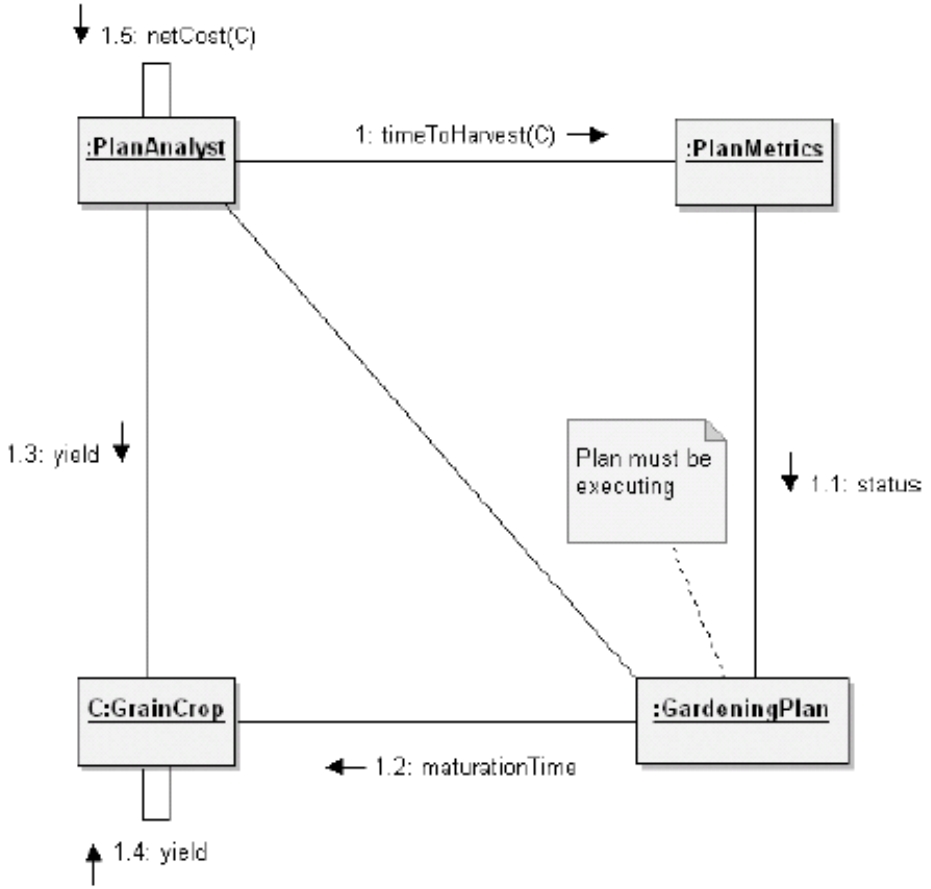
- A single object diagram represents a view of the object structure of a system and is typically used to represent a scenario

Object Diagrams



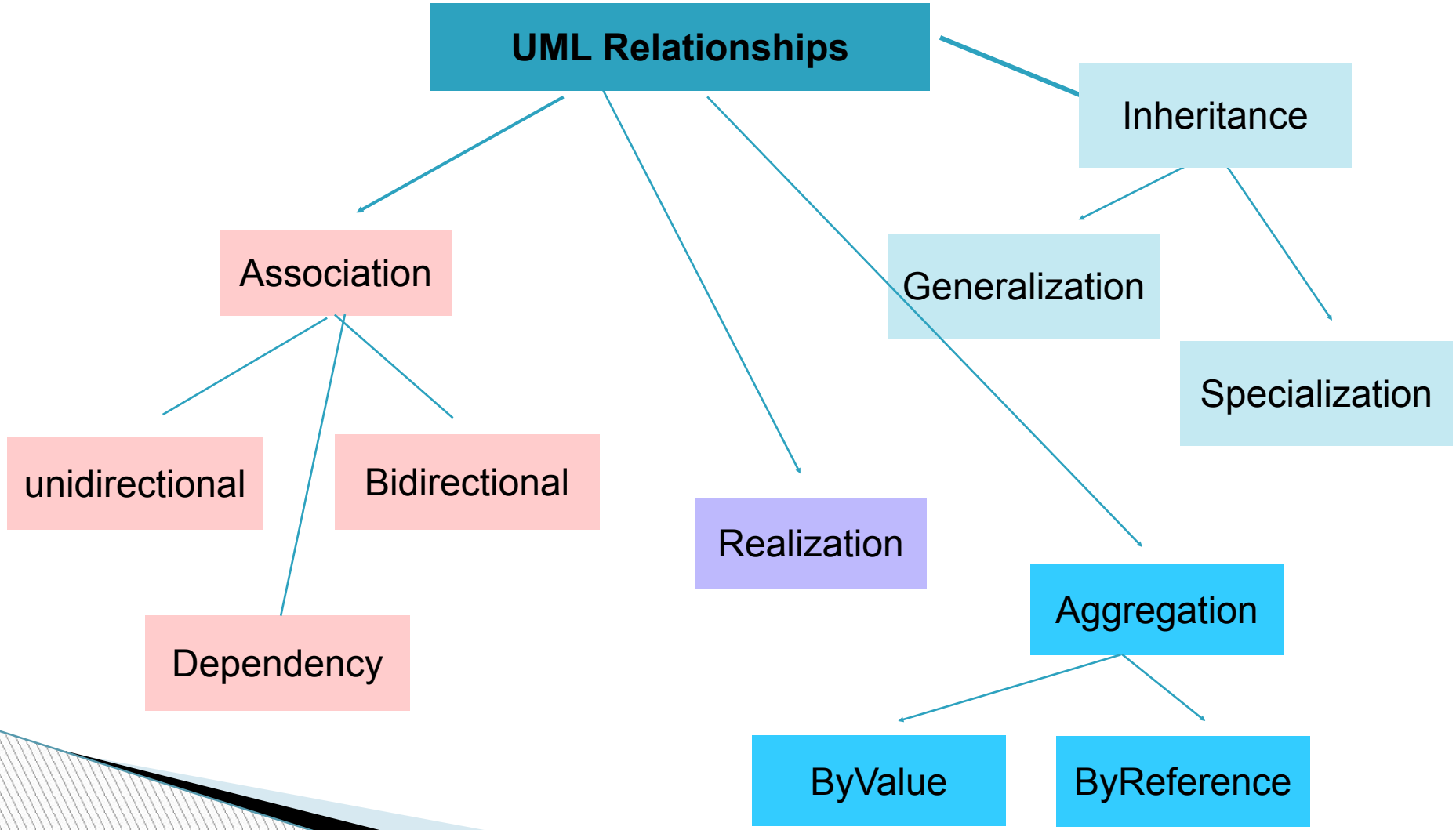
Communication / Collaboration Diagrams

A communication diagram is a type of interaction diagram that focuses on how objects are linked and the messages they pass, as they participate in a specific interaction.



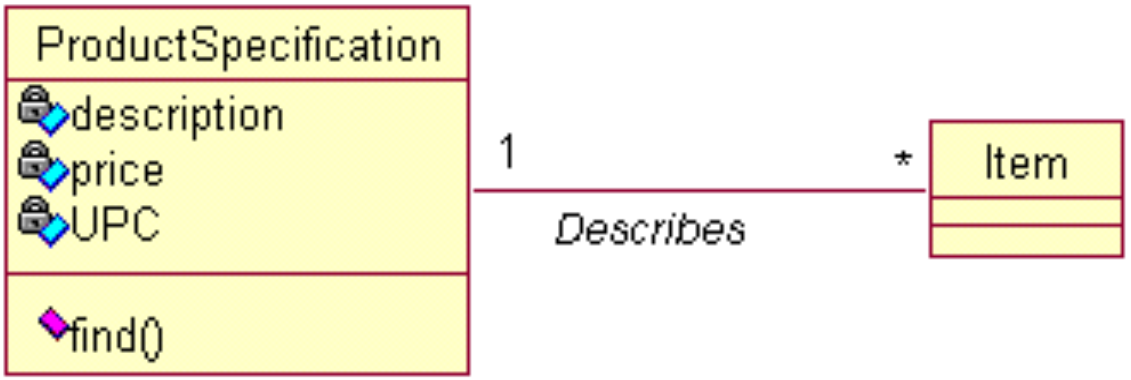
Modeling relationships using UML

Types of Relationships

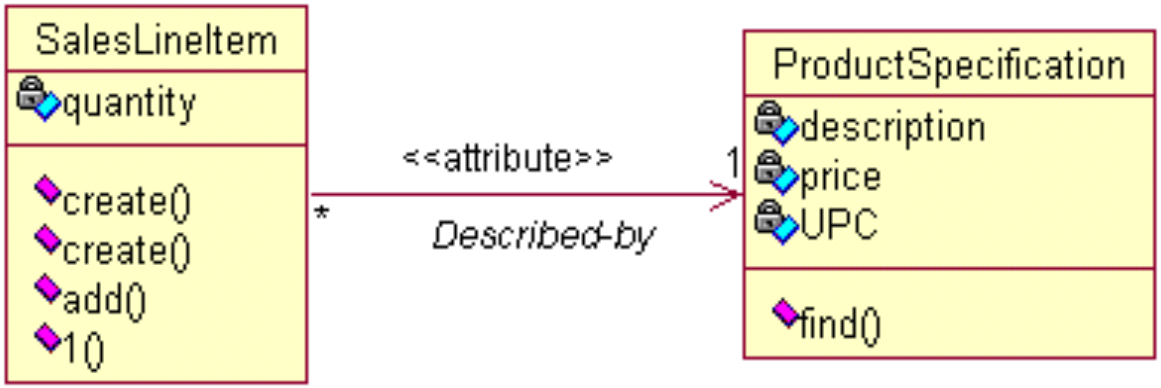


Association:

Bidirectiohal

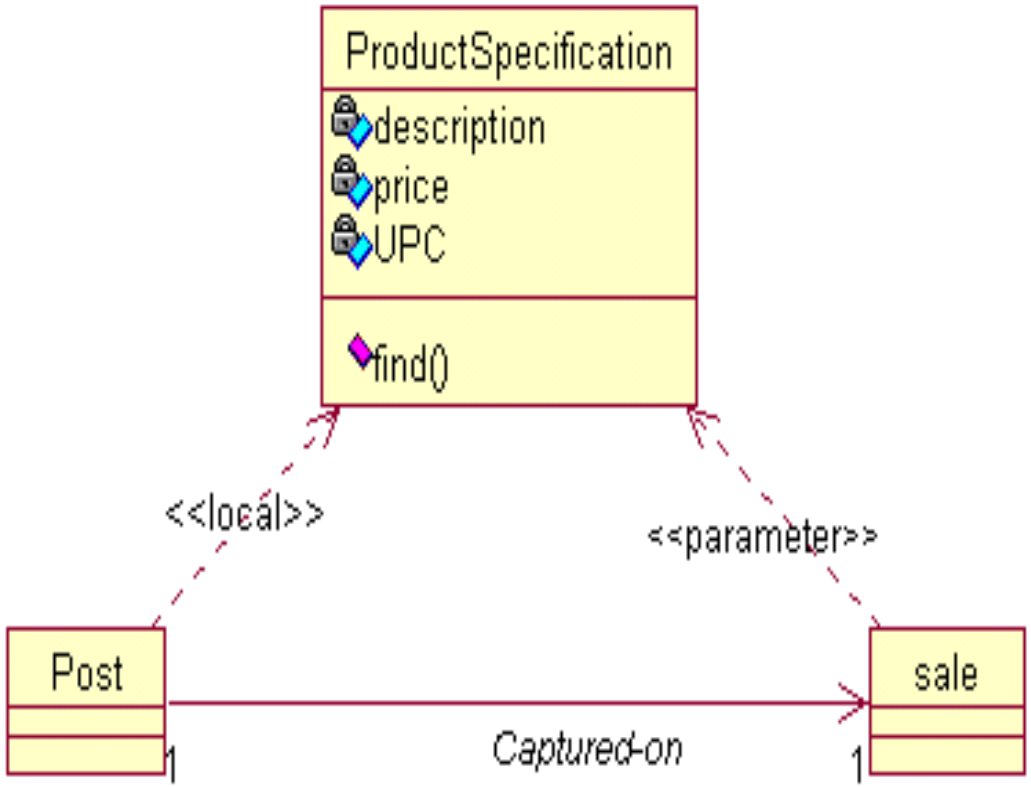


unidirectional



Attribute Visibility

Dependency: Non Attribute Level Visibility

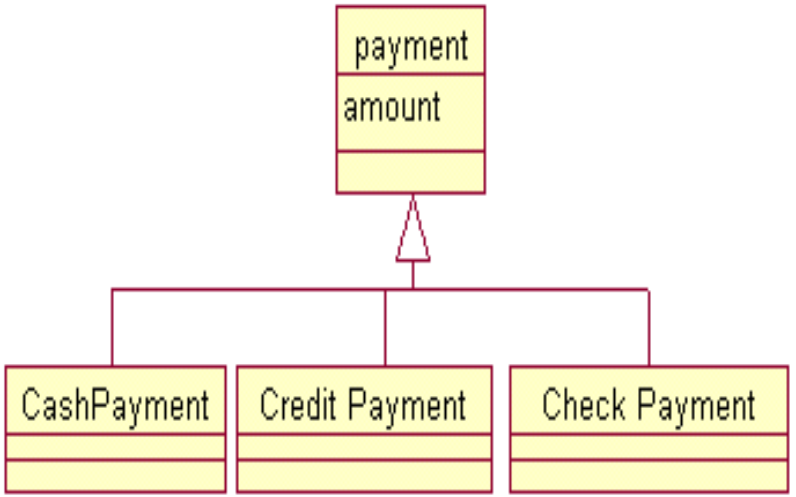


Generalization - Specialization Relationship

What is a Correct Subtype?

1.> 100% Rule (definition conformance)
100 % of the super-type's definition should be applicable to the subtype.
The subtype must conform to 100% of the super-type's attribute and association.

2. > Is-a Rule (set membership conformance)
All the members of a subtype must be members of their super-type set
i.e. subtype **is a** super-type.



Generalization - Specialization Relationship

Motivations to Partition a Type into Subtypes:

1> Create a subtype has additional attributes of interest.
2> The subtype has additional associations of interest.
3> The subtype concept is operated upon, handled, reacted to or manipulated differently than the supertype or other subtypes, in ways that are of interest.
4> The subtype concept represents an animate thing (for example, animal, robot) that behaves differently than the supertype or other subtypes, in ways that are of interest.

Generalization - Specialization Relationship

Motivations to Partition a Type into super-type:

Create a super-type in a generalization relationship to subtypes when:

> The potential subtypes represent variations on a similar concept

> All subtypes have the same attribute which can be factored out and expressed in the super-type.

>All subtypes have the same association which can be factored out and related to the super-type.

COMPOSITION:

Consider showing composition when:

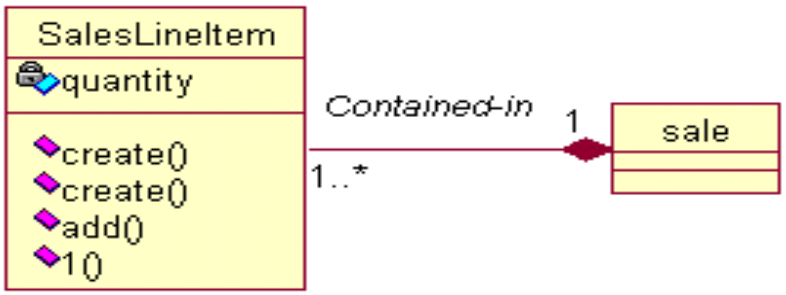
> The Lifetime of the part is bound within the lifetime of the composite

there is a create-delete dependency of the part on the whole.

> There is an obvious whole-part physical or logical assembly.

> Some properties of the composite propagate to the parts, such as its location.

> Operations applied to the composite propagate to the parts, such as destruction, movement, recording.



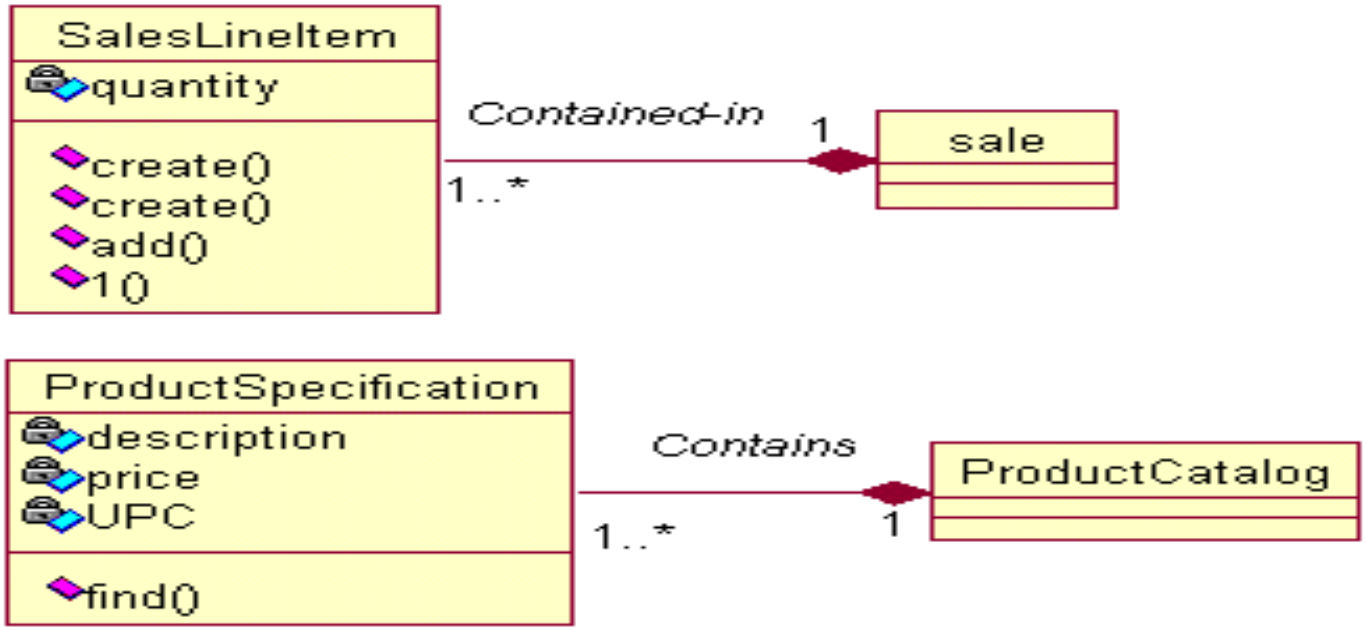
Benefit of Showing Aggregation:

NOTE: Identifying and illustrating aggregation is not profoundly important; it is quite feasible to exclude it from a conceptual model.

Include aggregation because it has the following benefits:

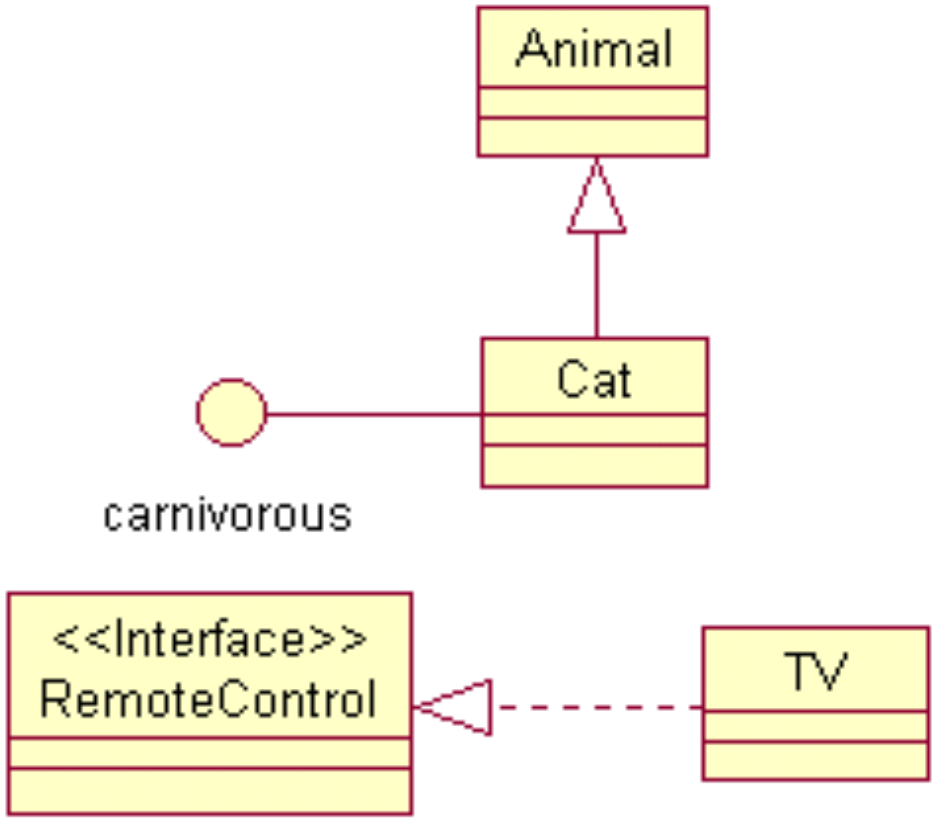
- It clarifies the domain constraints regarding the eligible existence of the part independent of the whole. In aggregation-by val, the part may not exist outside of the lifetime of the whole.
    - During the code phase, this has an impact on the create-delete dependencies between the whole and part software classes.
- It assists in the identification of a creator (the composite) using the GRASP Creator patterns.
- Operations - such as copy or delete - applied to the whole should often propagate to the parts.
- Identifying a whole in relation to a part supports encapsulation. The GRASP Don't Talk to Strangers pattern is used to hide the parts within the whole.

Aggregation in the Point-of-sale application

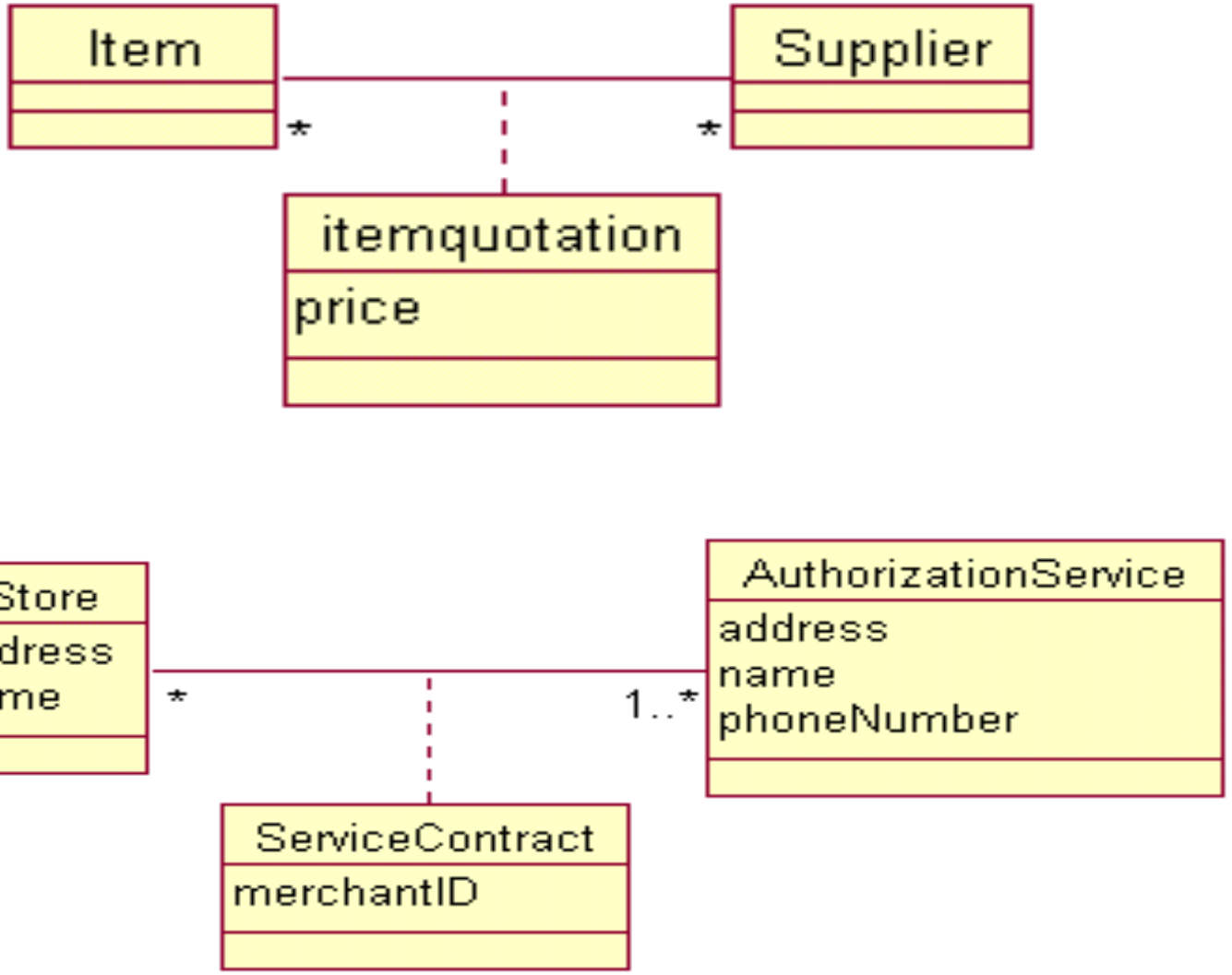


Realization:

Interface is realized in the class.



An associative Type/Class:



Qualified Associations:

A qualifier may be used in an association; it distinguishes the set of objects at the far end of the association based upon the qualifier value. An association with a qualifier is a **qualified association**.