Architecture Design

Instructor: Mehroze Khan

Design Methodology

- We have an
  - abstract description of a solution to our customer’s problem
  - a software architectural design
  - a plan for decomposing the design into software units and allocating the system’s functional requirements to them
- No distinct boundary between the end of the **architecture-design** phase and the start of the **module-design** phase
- No comparable design recipes for progressing from a **software unit’s specification** to its **modular design**
- The process taken towards a final solution is not as important as the **documentation** produced
- Design decisions are periodically **revisited** and **revised**
- **Refactoring**
  - to simplify complicated solutions or to optimize the design

Design Documentation

- The details of the system architecture is documented in *Software Architecture Document* (SAD)
- SAD serves as a bridge between the requirements and the design
- Program (or module) design acts as a bridge from architecture design to code
- Many ways to document the design
- **Design by contract**: a particular approach that uses the documentation not only to capture the design but also to encourage interaction among developers

Design by Contract

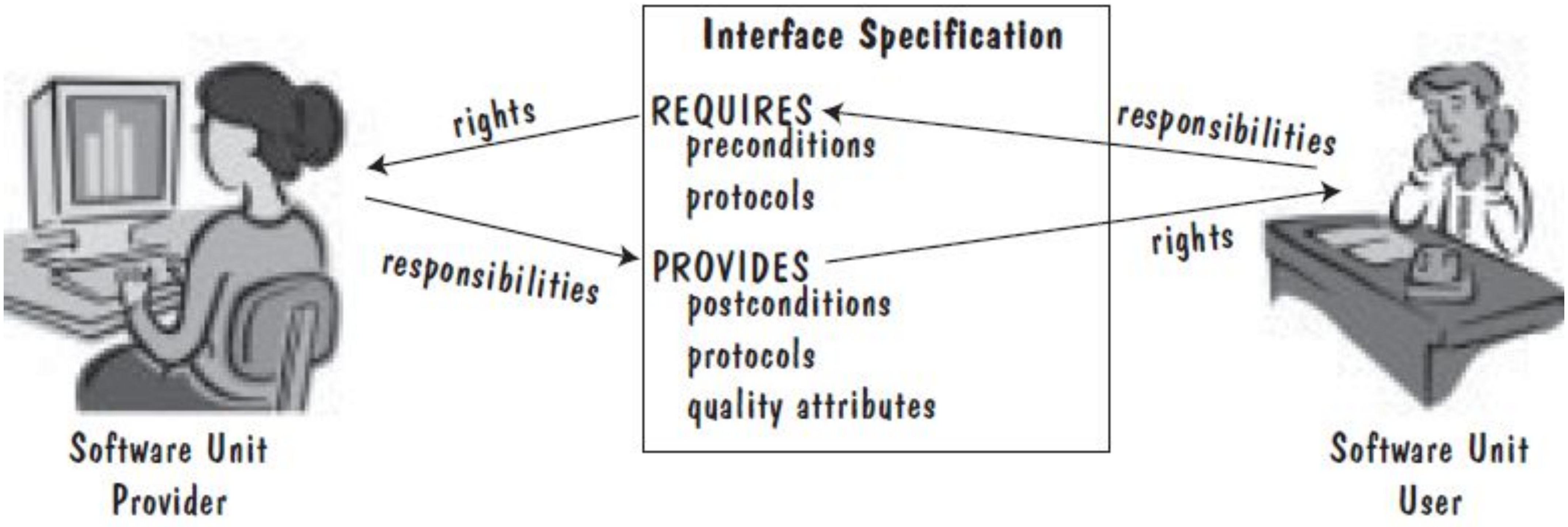
- In design by contract, each module has an interface specification that precisely describes what the module is supposed to do
  - Meyer (1997) suggests that design by contract helps ensure that modules interoperate correctly
  - This specification, called a **contract**, governs how the module is to interact with other modules and systems
  - Such specification cannot guarantee a module’s correctness, but it forms a clear and consistent basis for testing and verification
  - The contract covers mutual obligations (the preconditions), benefits (the postconditions), and consistency constraints (called **invariants**)
  - Together, these contract properties are called **assertions**

Design by Contract

- As the module provider, we uphold our end of the contract as long as
  - Our module provides (at the least) all of the **postconditions**, **protocols**, and **quality attributes** that are advertised in the interface specification
  - Our code requires from its environment no more than what is stated in the **interface's preconditions** and **protocols**
- As a software-unit user, we uphold the contract as long as
  - Our code uses the unit only when the unit's specified preconditions and protocols are satisfied
  - Our code assumes no more about the unit's behavior than is stated in its interface's postconditions, protocols, and invariants

Design by Contract

- Design Contract between software provider and user



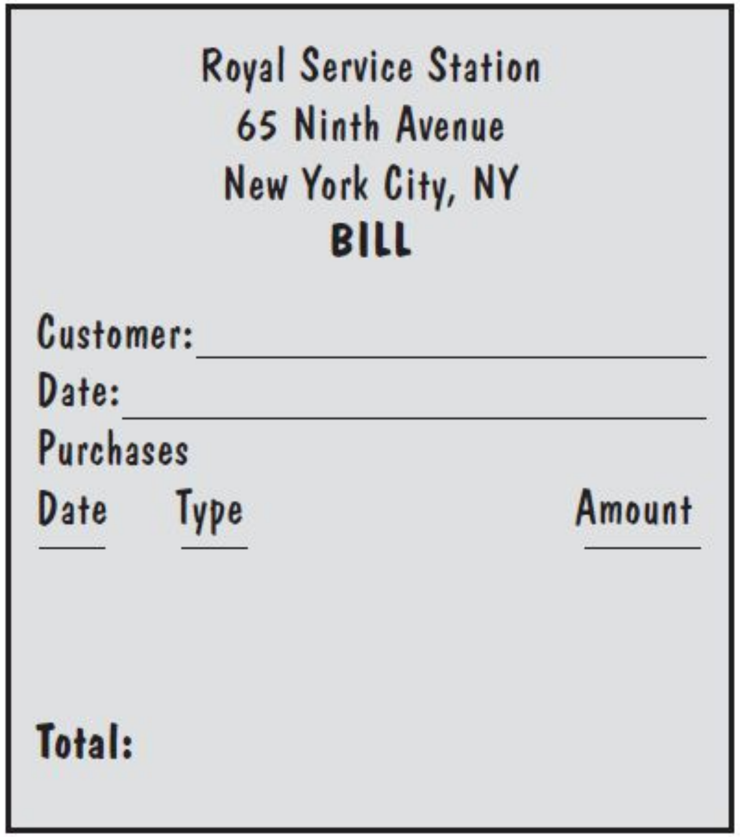
Other Design Considerations

Designing User Interfaces

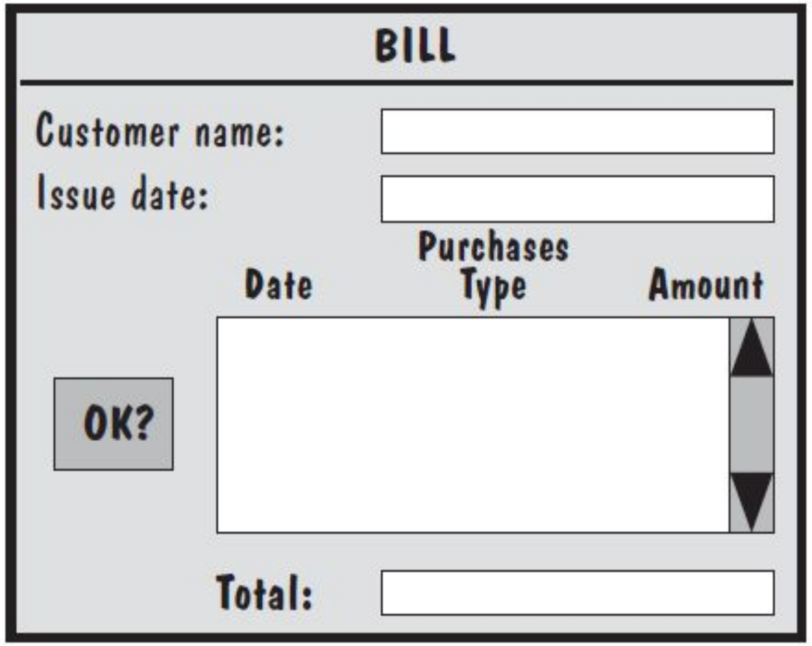
- Must consider several issues:
  - identifying the humans who will interact with the system
  - defining scenarios for each way that the system can perform a task
  - designing a hierarchy of user commands
  - refining the sequence of user interactions with the system
  - designing relevant classes in the hierarchy to implement the user-interface design decisions
  - integrating the user-interface classes into the overall system class hierarchy

Other Design Considerations
Designing User Interfaces

Before



After



Component based Software Engineering

- Components are **higher-level abstractions** than objects and are defined by their **interfaces**.
- They are usually larger than individual objects, and all **implementation details are hidden** from other components.
- Component-based software engineering is the process of defining, implementing, and integrating or composing these **loosely coupled**, **independent** components into systems.

Component based Software Engineering

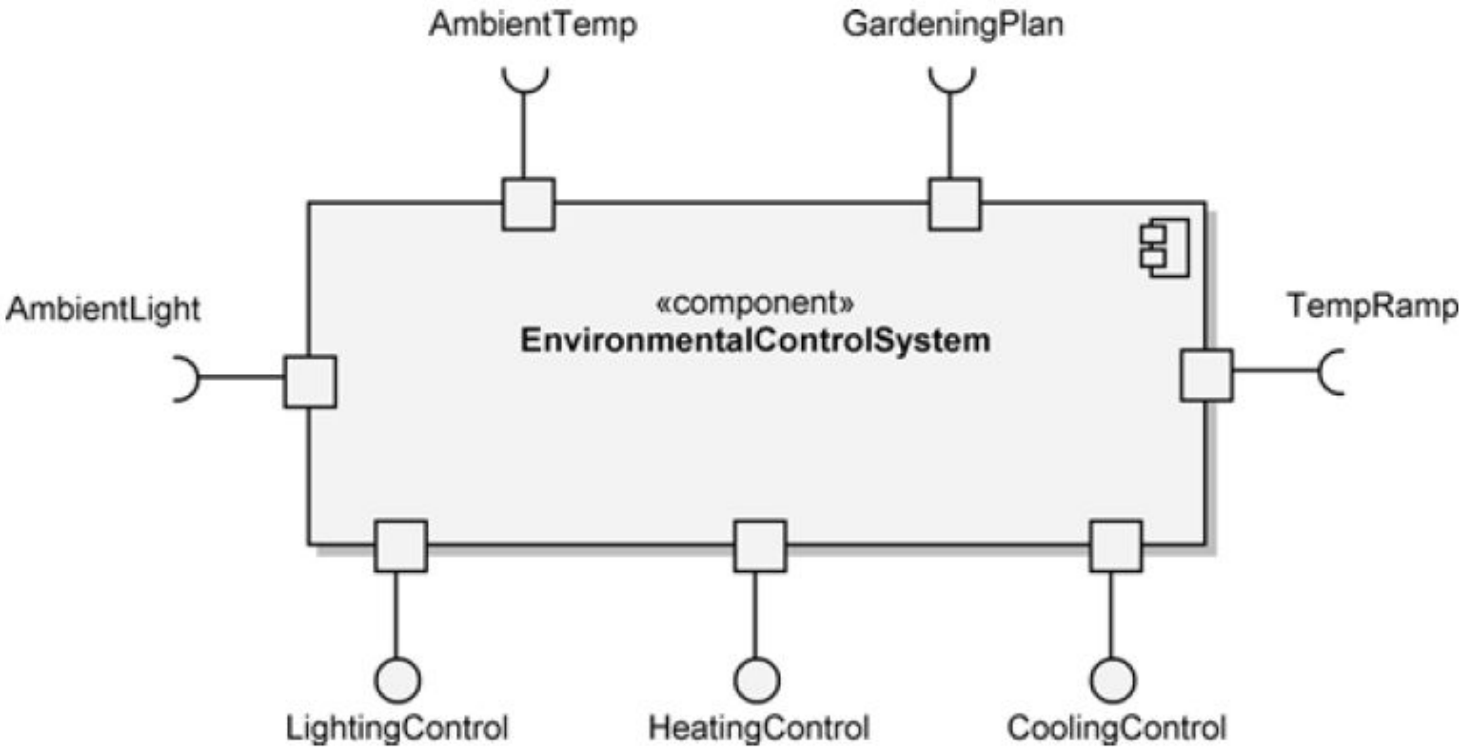
- Components are **independent**, so they do not interfere with each other's operation.
- Components communicate through **well-defined interfaces**.
- Component infrastructures offer a range of **standard services** that can be used in application systems. This reduces the amount of new code that must be developed.

Component

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

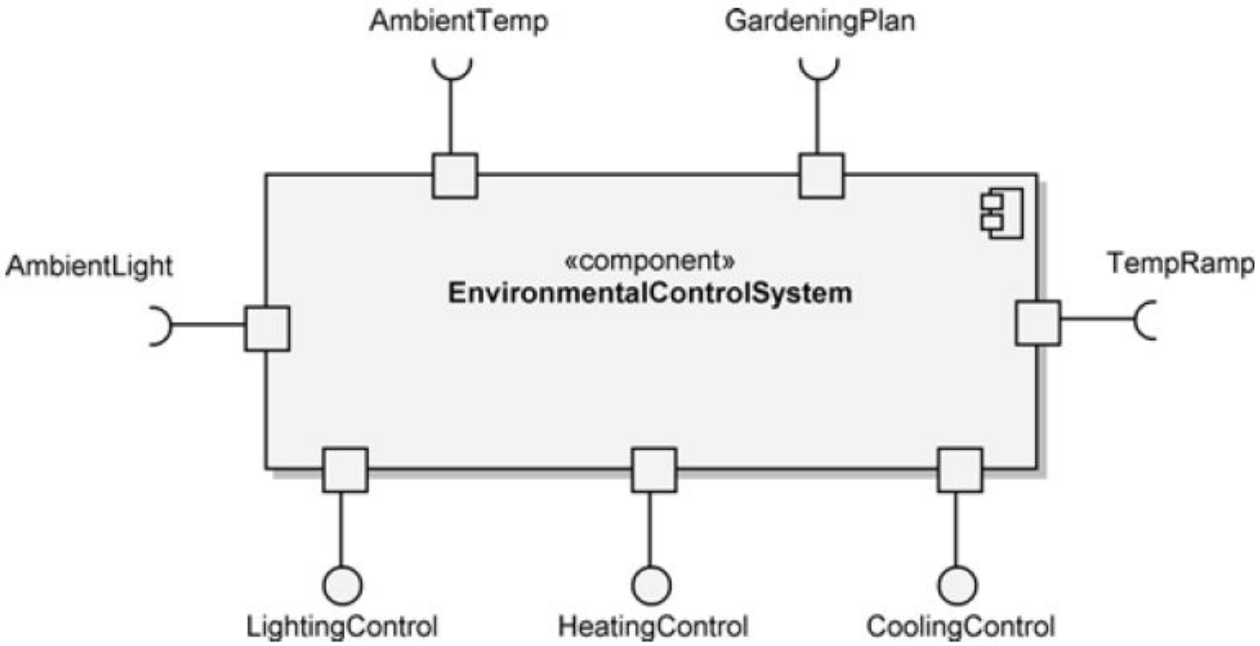
Component Notation

- Its name, EnvironmentalControlSystem, 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.



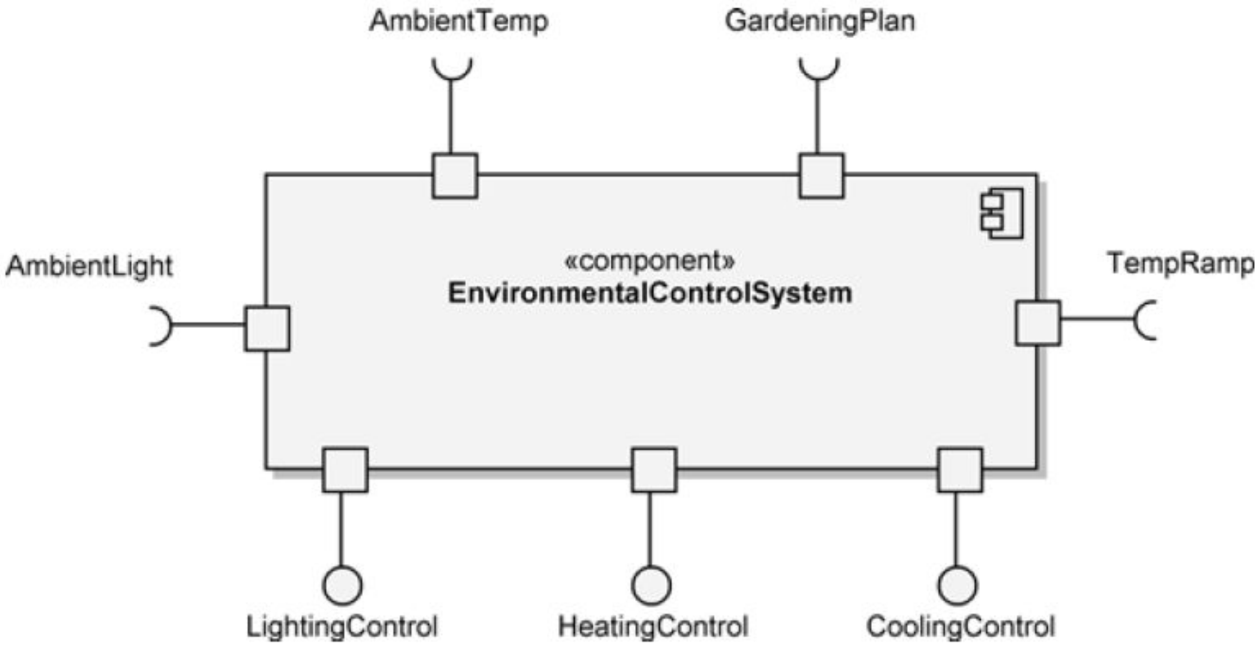
Component Notation

- On the boundary of the classifier rectangle, we have seven **ports**, which are denoted by small **squares**. Ports are used by the component for its **interactions** with its environment
- 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.



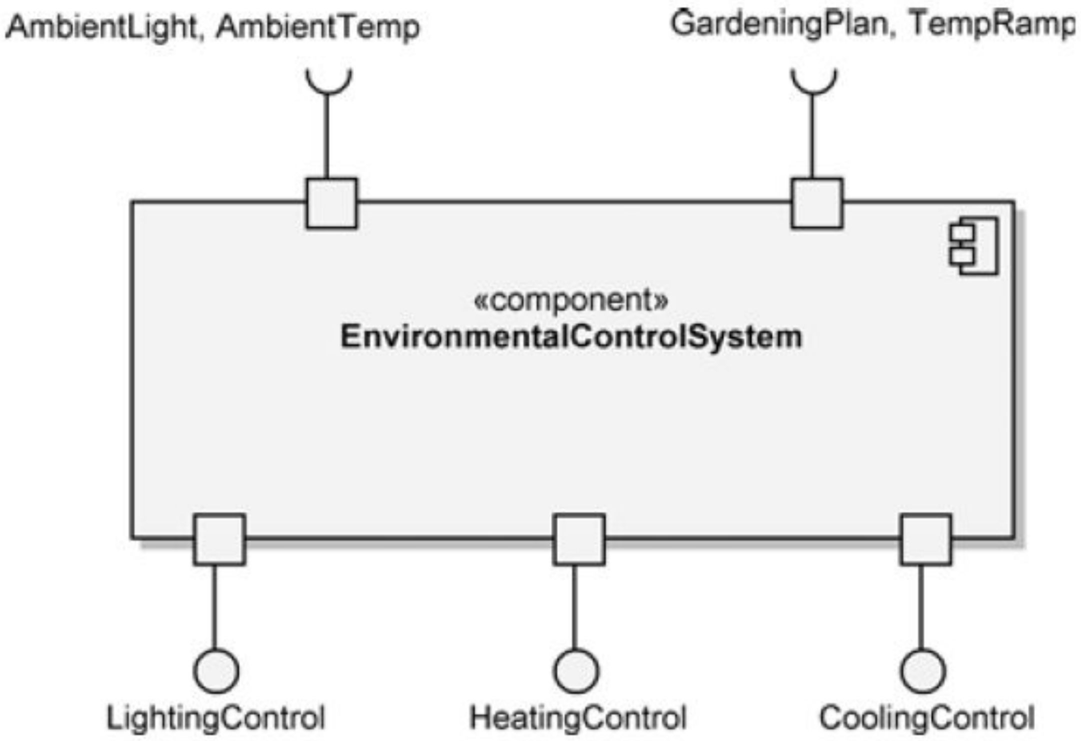
Component Notation

- Ports 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 i.e. LightingControl.
- **Required** interfaces use the **socket** notation to specify the services that the component requires from its environment i.e. AmbientTemp



Component Notation

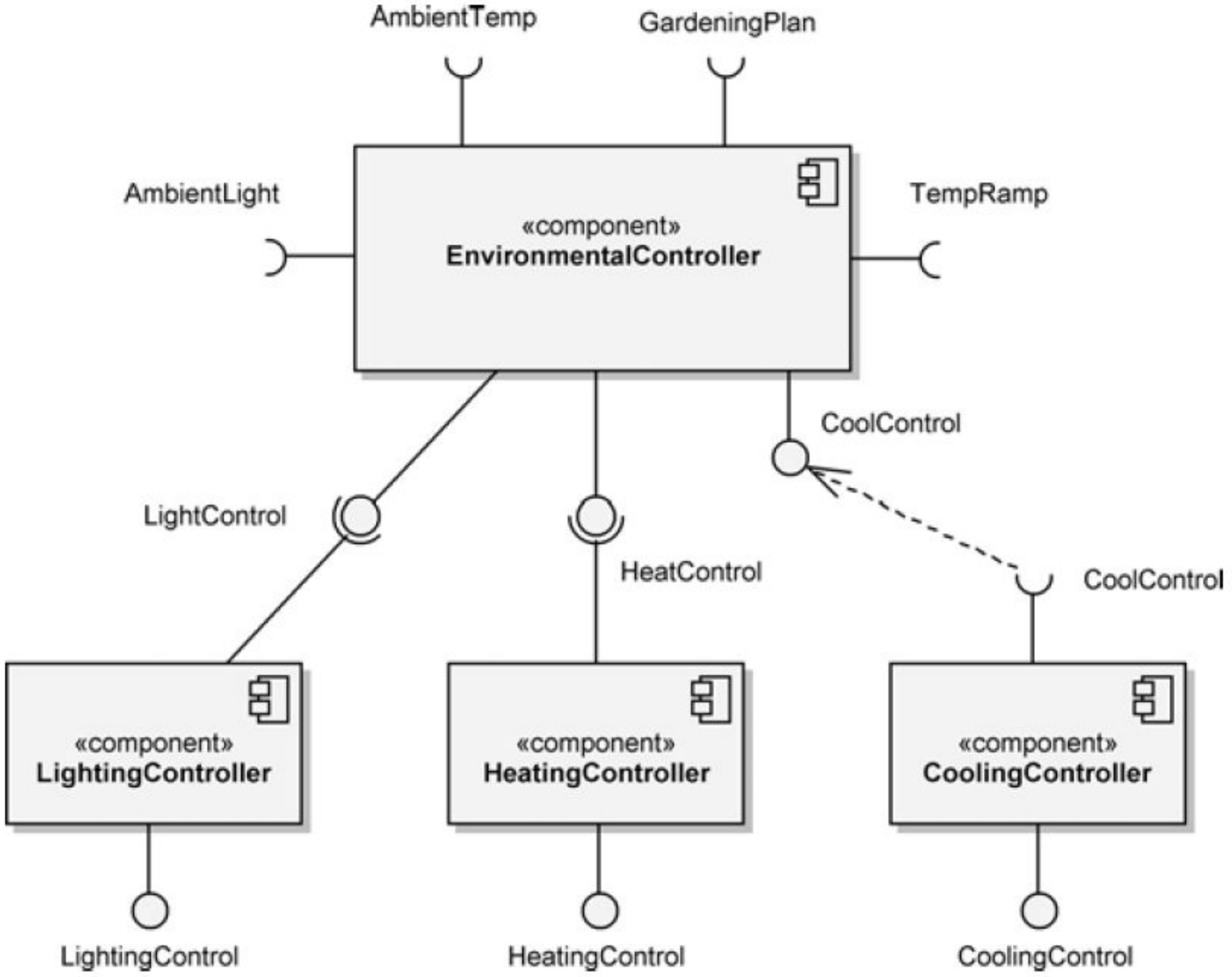
- Representation of EnvironmentalControlSystem is considered a **black-box** perspective since we see only the functionality required or provided by the component at its boundary.
- We are not able to peer inside and see the encapsulated components or classes that provide the functionality.
- A one-to-one relationship between ports and interfaces is not required; ports can be used to **group** interfaces



Component Diagram

- During development, we use component diagrams to indicate the logical layering and partitioning of our architecture.
- We represent the interdependencies of components, that is, their collaborations through well-defined interfaces to provide a system’s functionality

Component Diagram(EnvironmentalControlSystem)

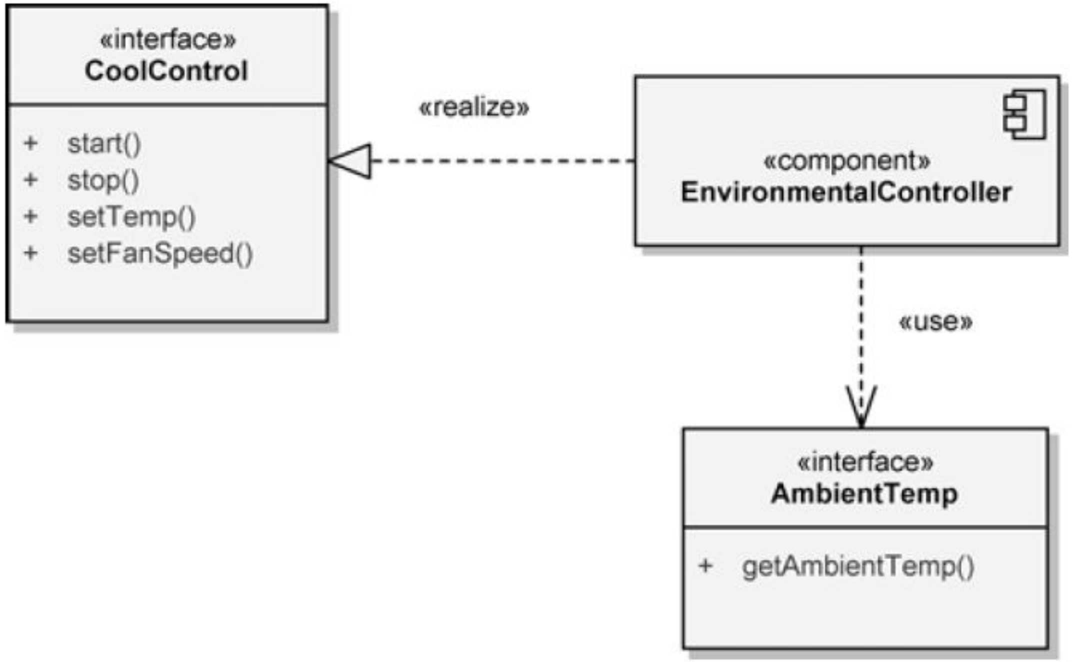


Component Diagram

- 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***.
- The interface between EnvironmentalController 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

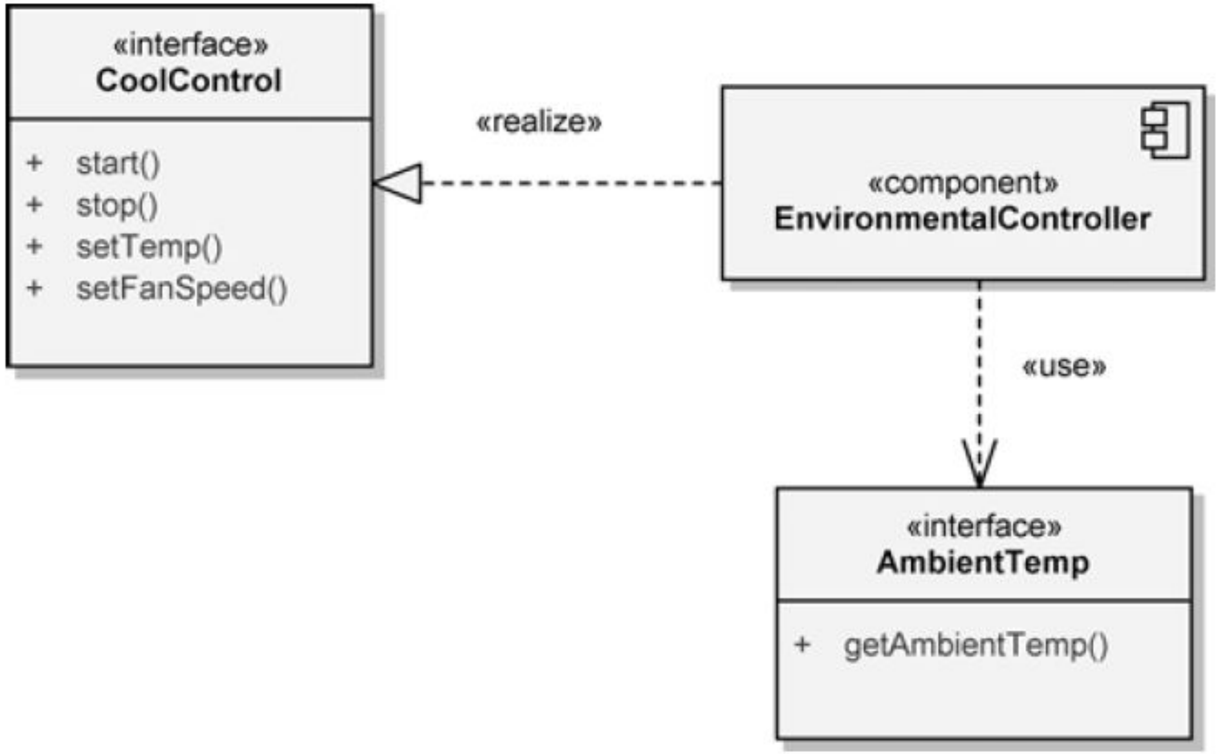
Component Diagram

- If we need to show more details about a component's interfaces, we may provide an **interface specification.**
- 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. These operations may be further detailed with **parameters** and **return types**, if needed.
- The CoolingController component requires the functionality of this interface.

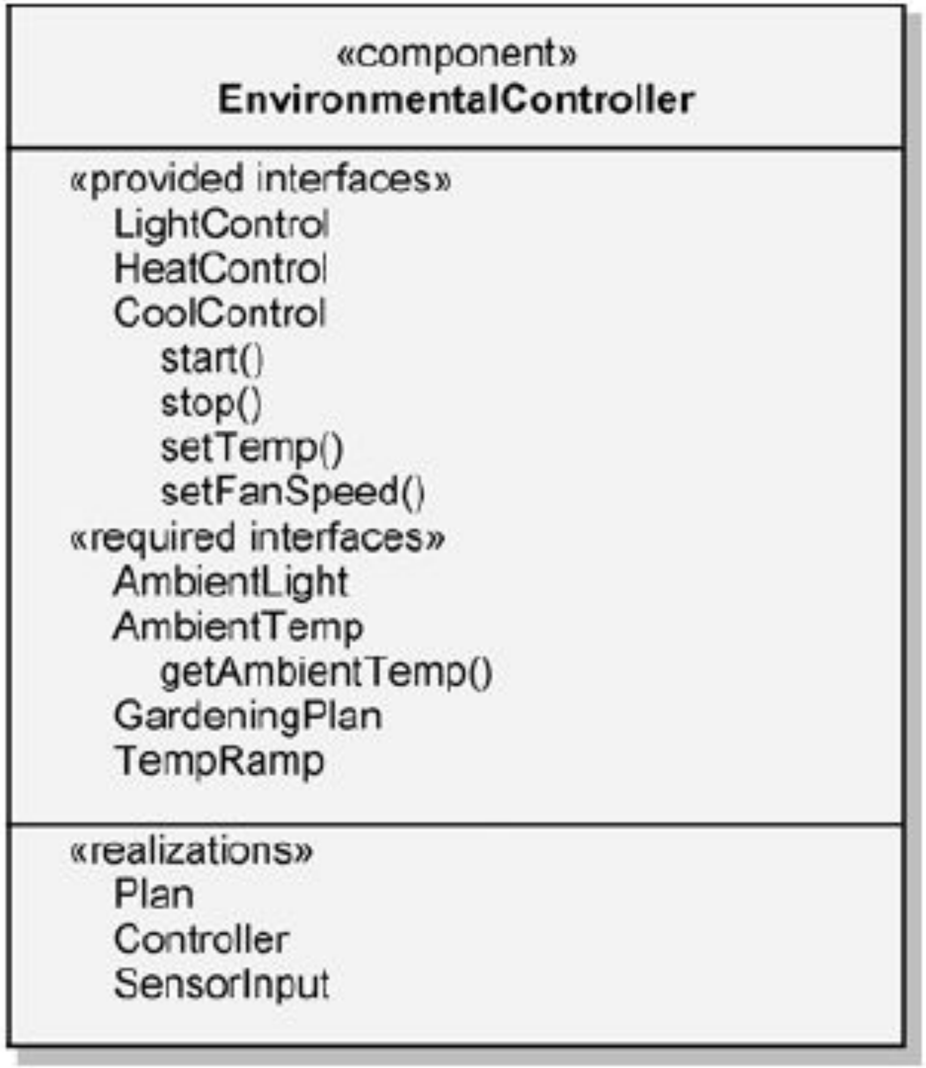


Component Diagram

- It 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.

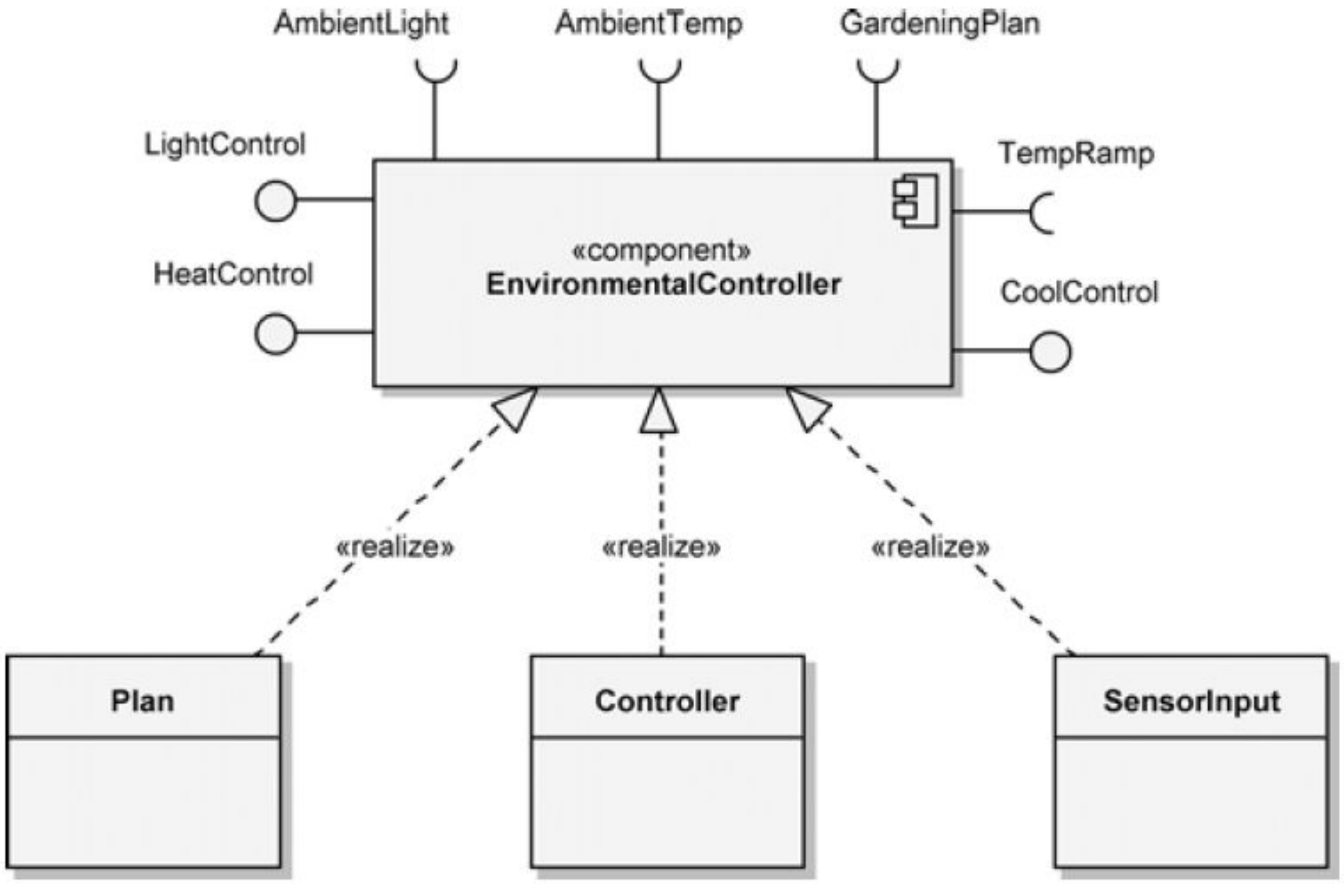


Component Diagram



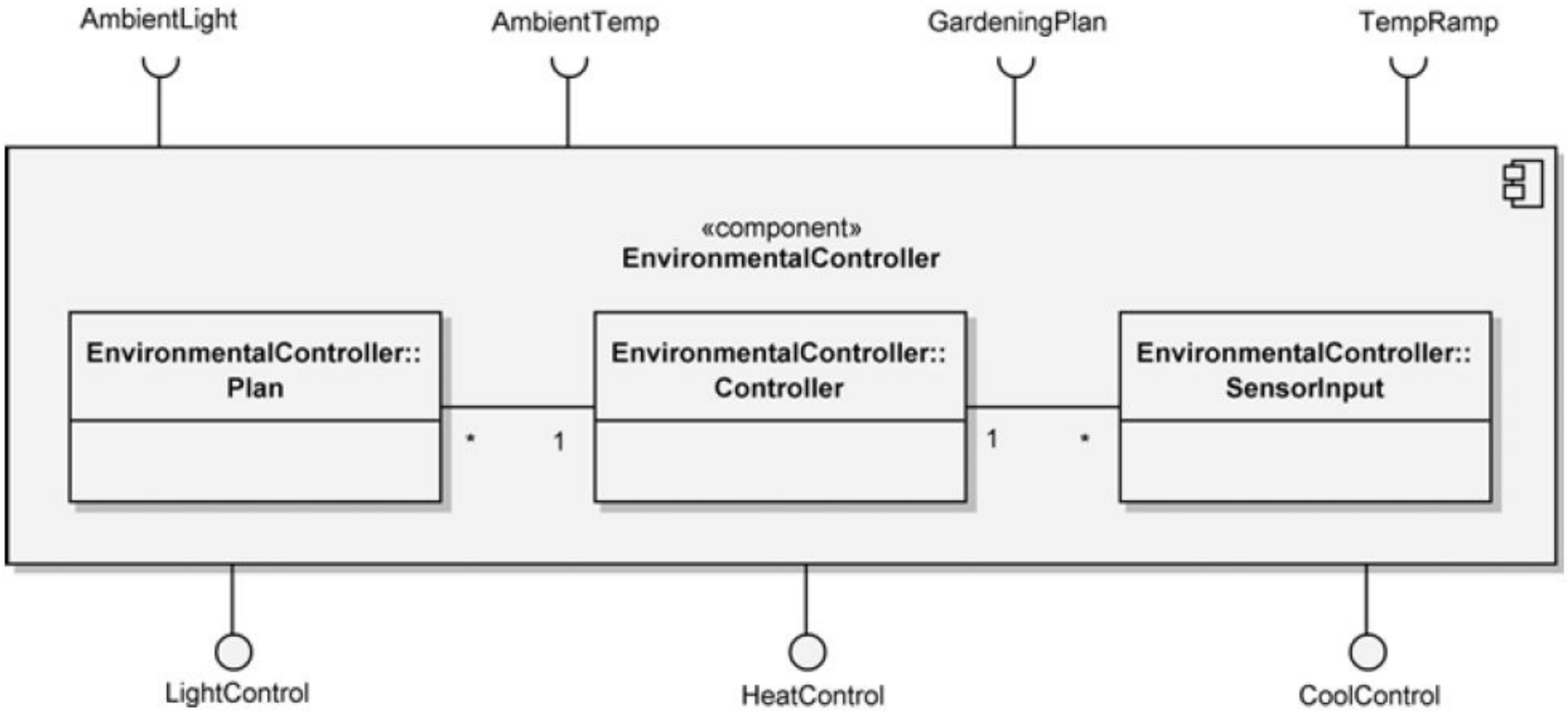
Component Realization

- EnvironmentalController component is realized by the classes Plan, Controller, and SensorInput.
- We need a realization dependency from each of the classes to EnvironmentalController

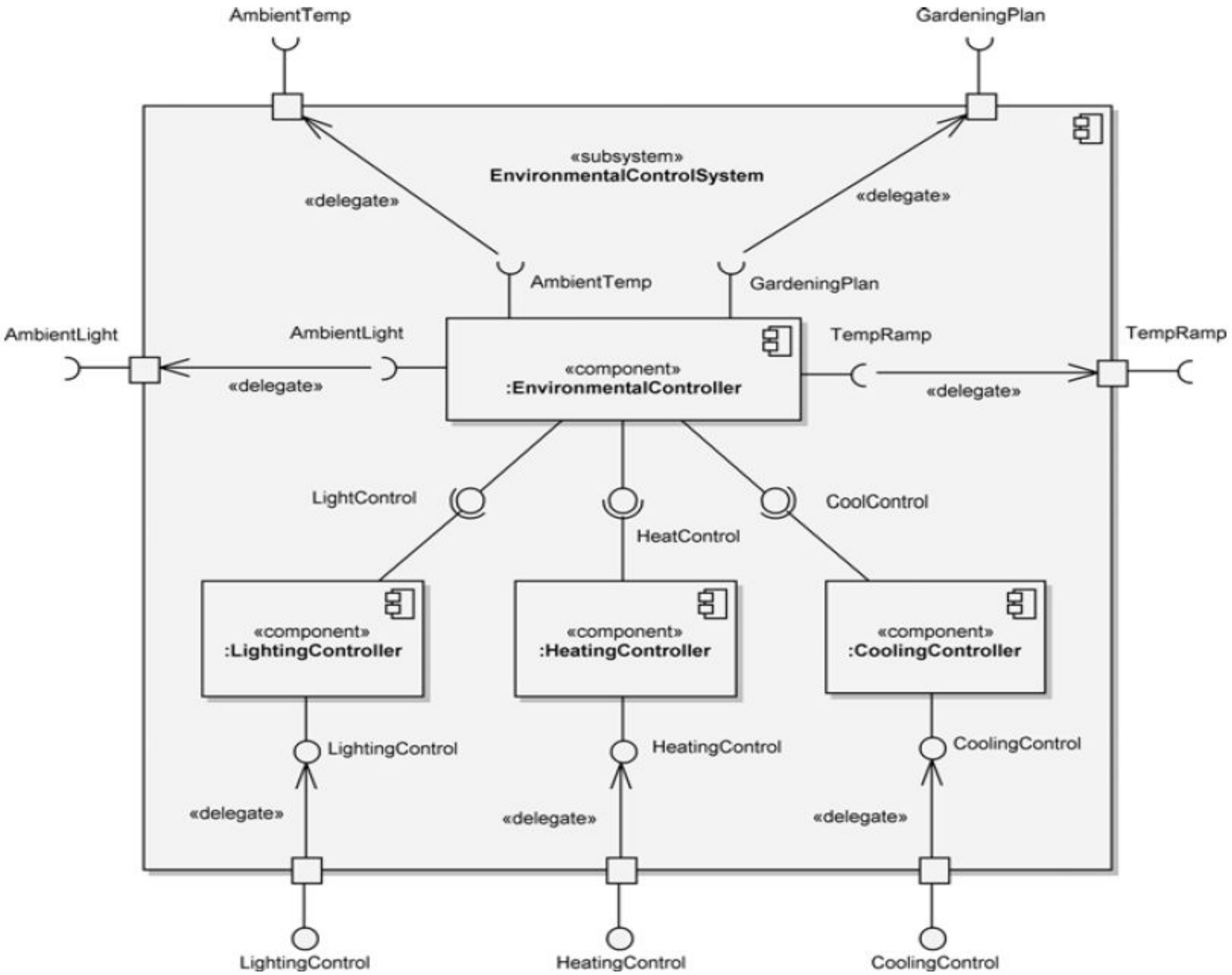


Containment Representation of EnvironmentalController's Realization

- EnvironmentalController component is realized by the classes Plan, Controller, and SensorInput.



Component's Internal Structure



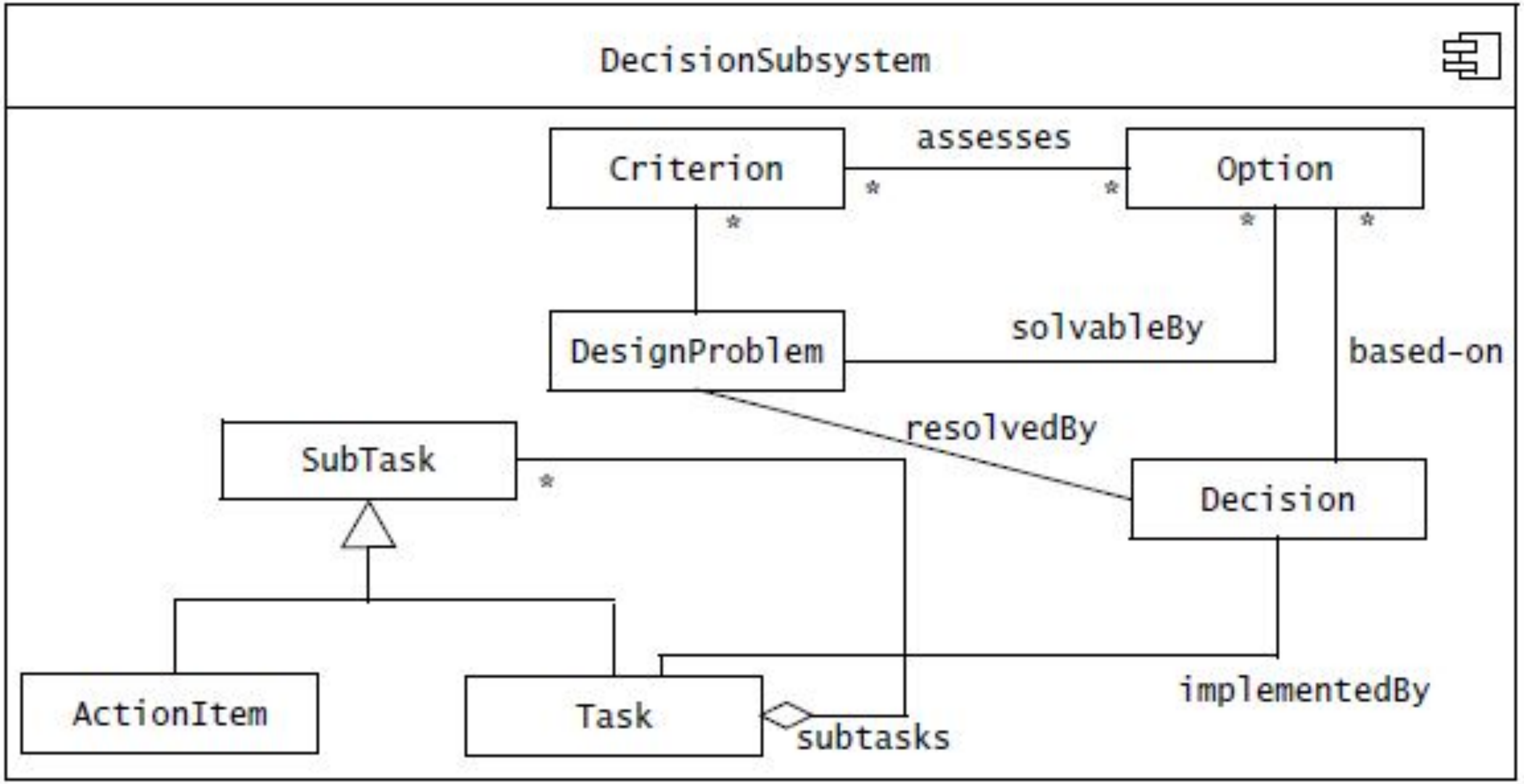
Component Diagram

- 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.

Decision Tracking System (DTS)

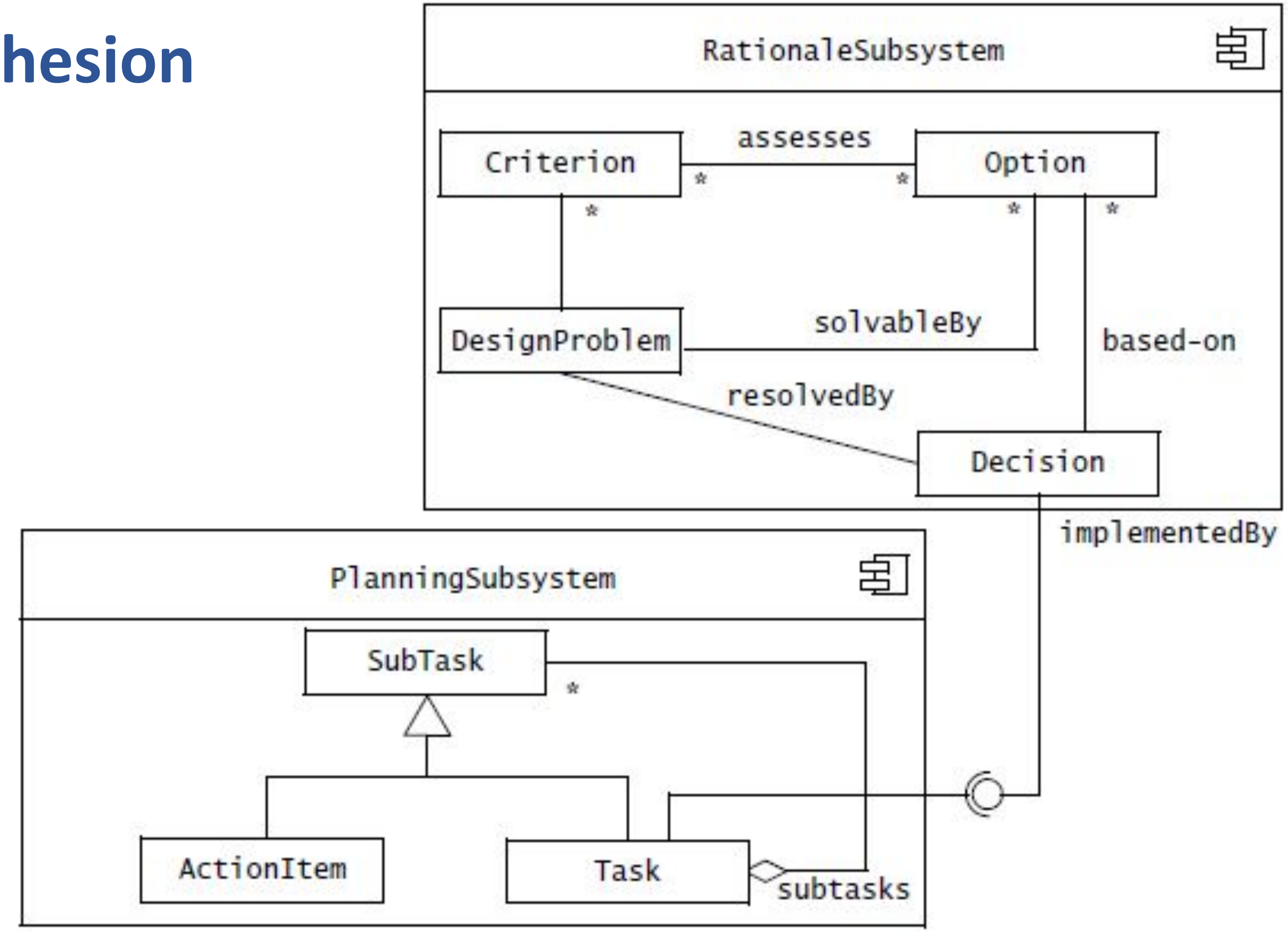
- Consider a decision tracking system for recording design problems, discussions, alternative evaluations, decisions, and their implementation in terms of tasks. `DesignProblem` and `Option` represent the exploration of the design space: we formulate the system in terms of a number of `DesignProblem` and document each `Option` they explore. The `Criterion` class represents the qualities in which we are interested. Once we assessed the explored `Option`s against desirable `Criteria`, we implement `Decision`s in terms of `Task`s. `Task`s are recursively decomposed into `Subtask`s small enough to be assigned to individual developers. We call atomic tasks `ActionItem`s.

DTS Subsystem



Component Diagram (Internal View)

Better Cohesion



References

- Roger S. Pressman, Software Engineering A Practitioner's Approach, 9th Edition. McGrawHill
- Shari PFleeger, Joanne Atlee, Software Engineering: Theory and Practice, 4th Edition
- Grady Booch et al., Object-Oriented Analysis and Design with Applications (3rd Edition), Pearson 2007.