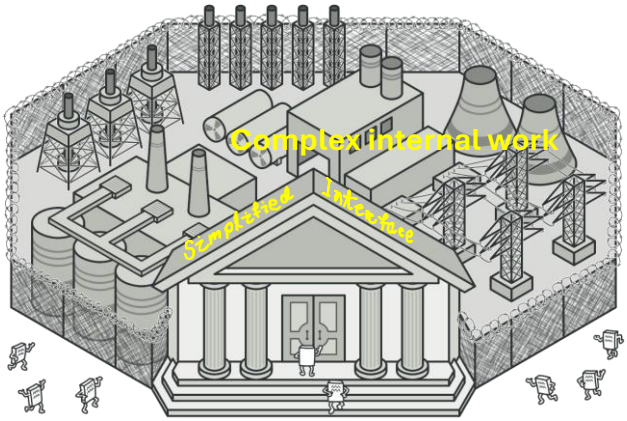
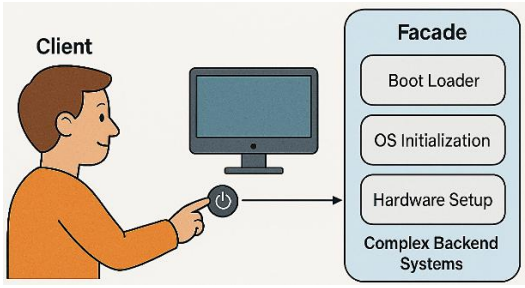
Facade Design Pattern

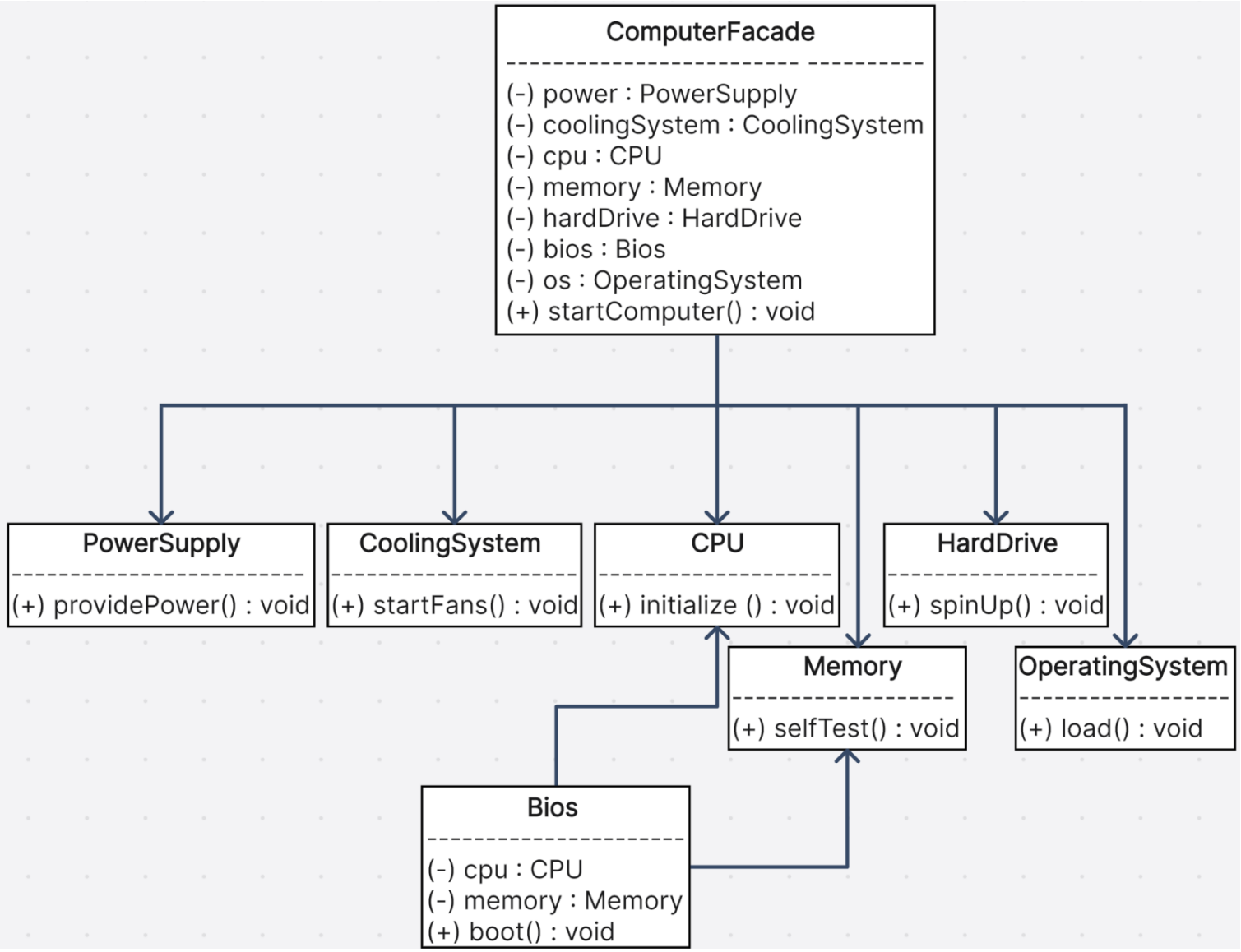


Defⁿ: The *Facade Design Pattern* is a *structural design pattern* that provides a *unified* and *simplified* interface to a complex subsystem or set of classes.



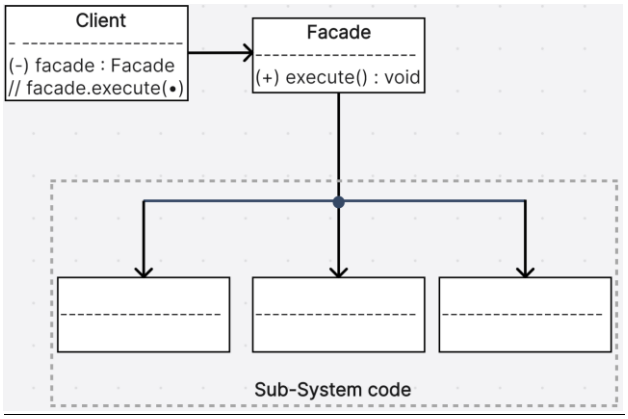
Example: Consider the complex set of *operations* that occur behind the scenes when a *computer starts*. From the user's perspective, it's as simple as clicking the *power button*— without knowing about the *complex* and *detailed internal steps* happening inside the system. This is the essence of the *Facade Design Pattern:* it provides a *simplified interface* that encapsulates and manages *complex subsystems*, making the system easier and more user-friendly to interact with.

UML:

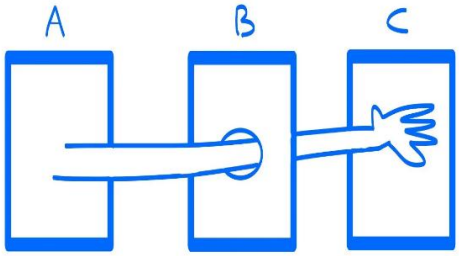


Code Link:
https://github.com/sibasundarj8/-System-Design-/tree/main/Codes/17___Facade%20Design%20Pattern%20code

Standard UML of Facade design pattern:



Law of Demeter or Principle of Least Knowledge:



- A *class* should only *talk* to its *immediate friends* and *not* to *strangers*.
- In simple terms, a class or method should only *interact* with *objects* it *directly owns*, *creates*, or *receives* — not with objects returned by those objects.
- It basically says to *avoid chained calls*.

More specifically:

When working with any *object*, the *Principle of Least Knowledge* states that you should only invoke methods that belong to:

- The *object itself*
- Objects *passed as parameters* to the method
- Objects *created* within the method
- Objects that are *directly held* by the object (i.e., have a "*has-a*" relationship)

Why it matters:

- *Reduces* coupling between components.
- *Increases* maintainability and readability.
- Makes the code more robust and less sensitive to changes.

Difference between Adapter design pattern and Facade design pattern:

Both *Facade* and *Adapter* are *structural design patterns*, and while they may look similar (they both "wrap" other classes), they serve different purposes.

Aspect	*Facade Pattern*	*Adapter Pattern*
Purpose	*Simplify* a *complex subsystem* by providing a unified, *high-level interface*.	*Convert* one interface into another that a client expects.
Intent	Hide internal complexity and make usage easier.	Make *incompatible* interfaces *compatible*.
Used When	You want to *simplify* and *group complex subsystem* logic for clients.	You want to use an *existing class* but its interface *doesn't match* your needs.
Client Awareness	The client knows only the *Facade*, not the underlying subsystem.	The client knows the *target interface*, and the *Adapter* makes the adaptee *fit*.
Example Analogy	*TV Remote*: Simplifies turning on TV, setting input, volume, etc.	*Power Adapter*: Converts a *two-pin plug* to a *three-pin socket*.
Typical Scenario	You design it from *scratch* to *wrap subsystems*.	You apply it when *integrating* with *existing* or *legacy* systems.
Changes Interface?	*No* – it just provides a simpler one.	*Yes* – it changes one interface into another.