Design Patterns

SE3030 – Software Architecture | Vishan Jayasinghearachchi

Topics

- What are design patterns?
- Gang (GoF) of Four design patterns
- Scenario Discussion – A Smart Home Solution
  - Problem #1: Device Compatibility Across Brands (Abstract Factory)
  - Problem #2 - Protocol-Specific Device Creation (Factory Method)
  - Problem #3 - Centralized Automation Controller (Singleton)
  - Problem #4 - Flexible Automation Rules (Strategy)
  - Problem #5 - Undoable User Actions (Command)
  - Problem #6 - Standardized Device Initialization (Template Method)
  - Problem #7 - Real-Time Device Status Updates (Observer)
  - Problem #8 - Decoupling UI from Device Logic (Bridge)

What are design patterns?

- Algorithms such as searching algorithms, and sorting algorithms.
- Templates for data structures
- Standard guidelines when designing software systems in a certain domain.

Are these design patterns?

Key Points about Design Patterns

- **Not algorithms or data structures:** Unlike algorithms, which are step-by-step procedures for solving a specific computational problem (like sorting or searching), design patterns are higher-level strategies for organizing code and structuring systems [1] [6] [7].
- **Not templates for data structures:** Design patterns are not concrete templates for data structures like arrays or linked lists. Instead, they provide guidance on how to solve design problems that recur across different projects [7].
- **Standard guidelines:** They act as standard guidelines or best practices for solving design challenges in software engineering, especially in object-oriented systems [5] [7].
- **Reusable and adaptable:** Patterns are intended to be reused and adapted, not copied verbatim. The implementation details will differ based on the specific context and requirements of your project [1] [7].
- **Common vocabulary:** Design patterns provide a shared vocabulary for developers, making it easier to communicate complex design ideas efficiently [4] [5].

What are design patterns?

reliable

- Basically, a Design Pattern is a tried-and-true solution to a common problem encountered when designing software systems.
- The term “Design Patterns” has roots in (building) architecture.

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Christopher Alexander, A Pattern Language: Towns, Buildings, Construction (1977)

What are design patterns?

- *"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It **describes the problem, the solution, when to apply the solution, and its consequences**. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.“*

  Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides a.k.a Gang of Four (GoF) (1995)

GoF Design Pattern Categories

- Creational Patterns
  - Deals with object creation mechanisms, trying to create objects in a manner suitable to the situation with loose coupling.
- Behavioural Patterns
  - Deals with interactions, communication, and sharing of responsibilities among objects.
- Structural Patterns
  - Deals with techniques to compose objects to form larger structures.

GoF Design Patterns Discussed

- Abstract Factory
- Factory Method
- Singleton

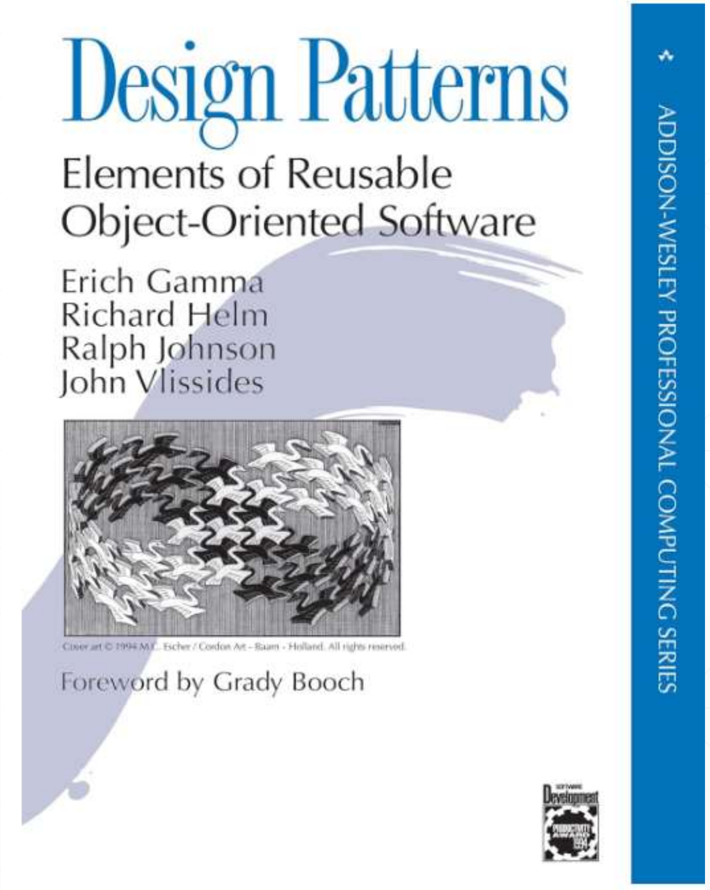
Creational Patterns

- Strategy
- Command
- Template Method
- Observer

Behavioural Patterns

- Bridge

Structural Patterns



Scenario: Smart Home Automation System

- A company is developing a Smart Home Automation System that allows users to control devices (lights, thermostats, security cameras, etc.) from a mobile or web application.
- The system needs to support multiple device brands, and different communication protocols (Wi-Fi, Zigbee, Bluetooth), and offer automation features.
- We will examine some common problems the developers will encounter when developing this software system, and how design patterns can help overcome them.

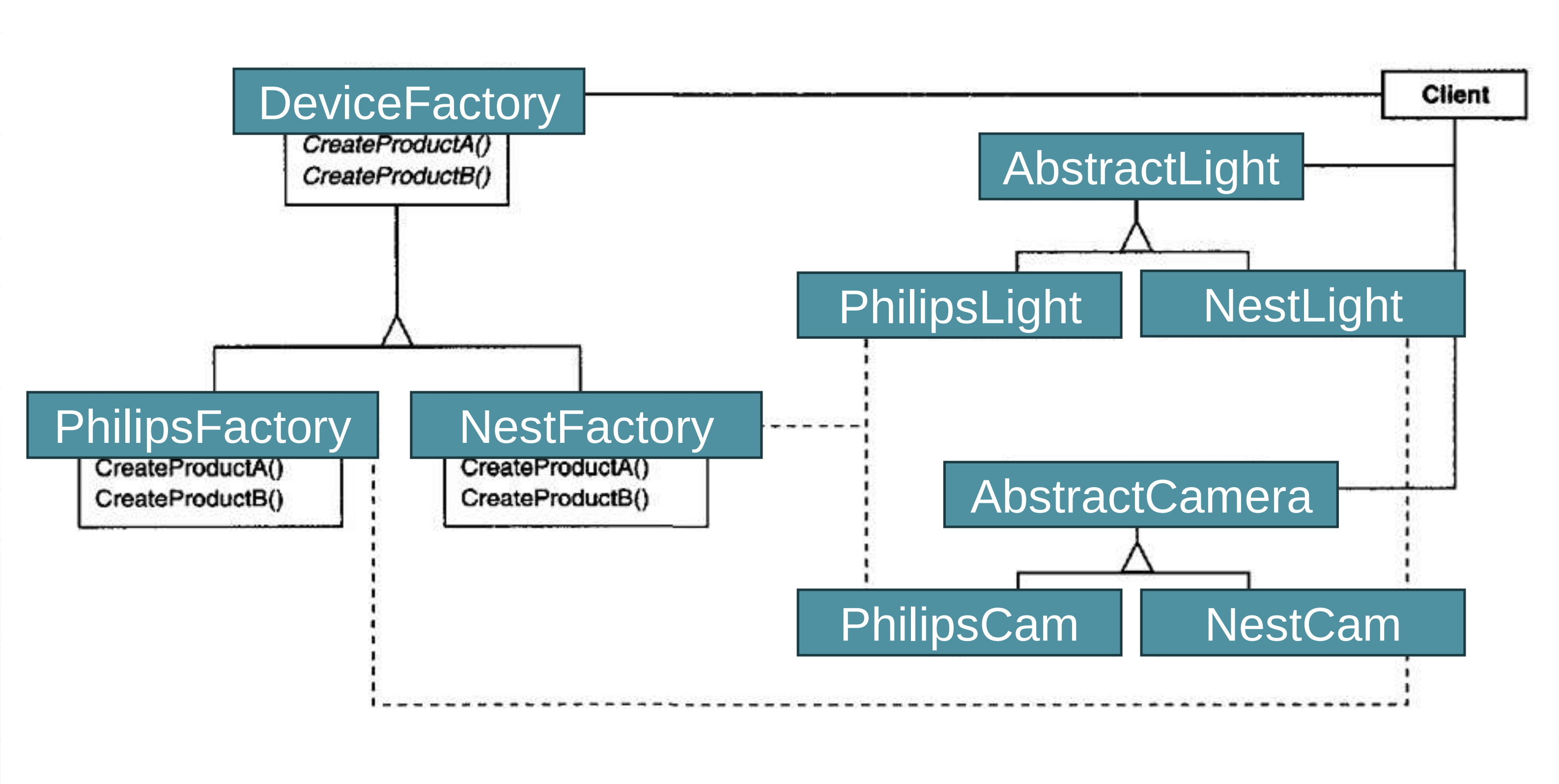
Problem #1: Device Compatibility Across Brands

- Different manufacturers like Philips, Google Nest etc. provide smart home devices (lights, thermostats, cameras) with unique features and configurations.
- A smart home system needs to support multiple device brands, each with unique features and configurations.
- Without a structured approach, adding new brands would require modifying large portions of the codebase, leading to a rigid and error-prone design.

Problem #1: Device Compatibility Across Brands

- The Abstract Factory Pattern can be used here.
- Instead of hardcoding each device type, the **Abstract Factory** pattern can create families of related objects without specifying their concrete classes.
- A DeviceFactory creates device-specific factories (PhilipsFactory/ NestFactory), each producing compatible devices (PhilipsLight, PhilipsThermostat, NestCamera etc.).
- Benefit: Easily supports adding new brands without modifying existing code.

Abstract Factory



Why Abstract Factory

- The Abstract Factory pattern allows the system to create families of related objects (lights, thermostats, cameras) without specifying their concrete classes.
- Instead of tightly coupling the system to specific brands (if (brand == "Philips") { createPhilipsLight(); }), the system relies on a factory interface, which produces brand-specific implementations.
- This decouples device creation from device usage, allowing seamless integration of new brands without modifying existing logic.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

Example for Abstract Method

Problem #2 - Protocol-Specific Device Creation

- Different smart home devices use Wi-Fi, Bluetooth, or Zigbee for communication.
- If we hardcode protocol selection, maintaining and extending the system becomes difficult (e.g., adding support for a new protocol requires modifying existing code).

Issue: Diverse Communication Protocols

Smart home devices use different protocols with unique connection requirements:

- **Wi-Fi:** Requires IP-based configuration and authentication.
- **Bluetooth:** Needs device pairing and short-range management.
- **Zigbee:** Relies on mesh network setup.

Hardcoding protocol-specific logic would lead to:

- **Brittle code** tightly coupled to specific protocols.
- **Difficulty scaling** when adding new protocols (e.g., Thread, Matter).
- **Code duplication** for common connection tasks (e.g., retry mechanisms).

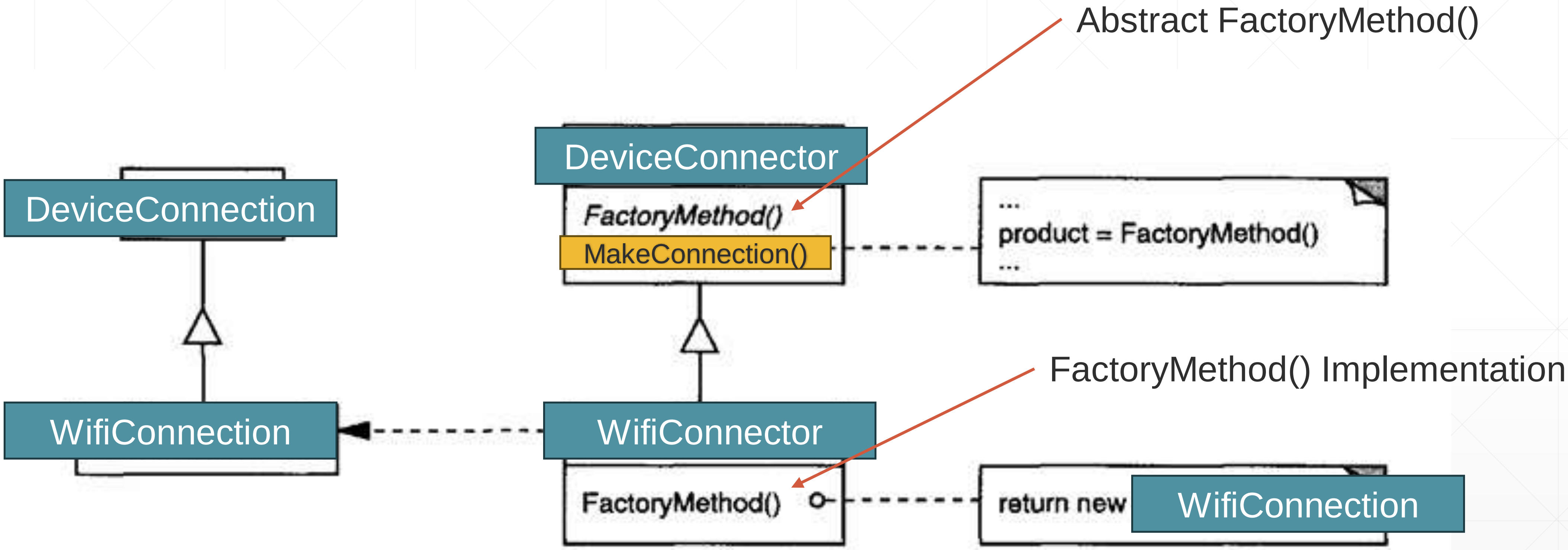
Solution: Factory Method Pattern

The Factory Method pattern delegates protocol-specific connection creation to subclasses while maintaining a unified interface for device communication.

Problem #2 - Protocol-Specific Device Creation

- The Factory Method pattern can be used here.
- A DeviceConnector (Creator) abstract class has a **Factory Method** createConnection(), implemented in WiFiConnector, BluetoothConnector, and ZigbeeConnector subclasses (ConcreteCreators)
- DeviceConnection (Product) Interface is implemented by WifiConnection BluetoothConnection, and ZigbeeConnection classes (ConcreteProducts).
- Each ConcreteCreator’s factory method implementation returns a matching ConcreteProduct instance.
- Benefit: Decouples the object creation logic, enabling seamless integration of new communication methods.

Factory Method



Why Factory Method

- The Factory Method pattern encapsulates the logic for selecting the correct communication protocol within a factory method (createConnection() in the DeviceConnector class).
- Subclasses (WiFiConnector, BluetoothConnector, ZigbeeConnector) of the DeviceConnector class override the factory method to instantiate the appropriate protocol.
- This decouples protocol selection from the main logic, making it easier to integrate new protocols without altering existing code.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

Problem #3 - Centralized Automation Controller

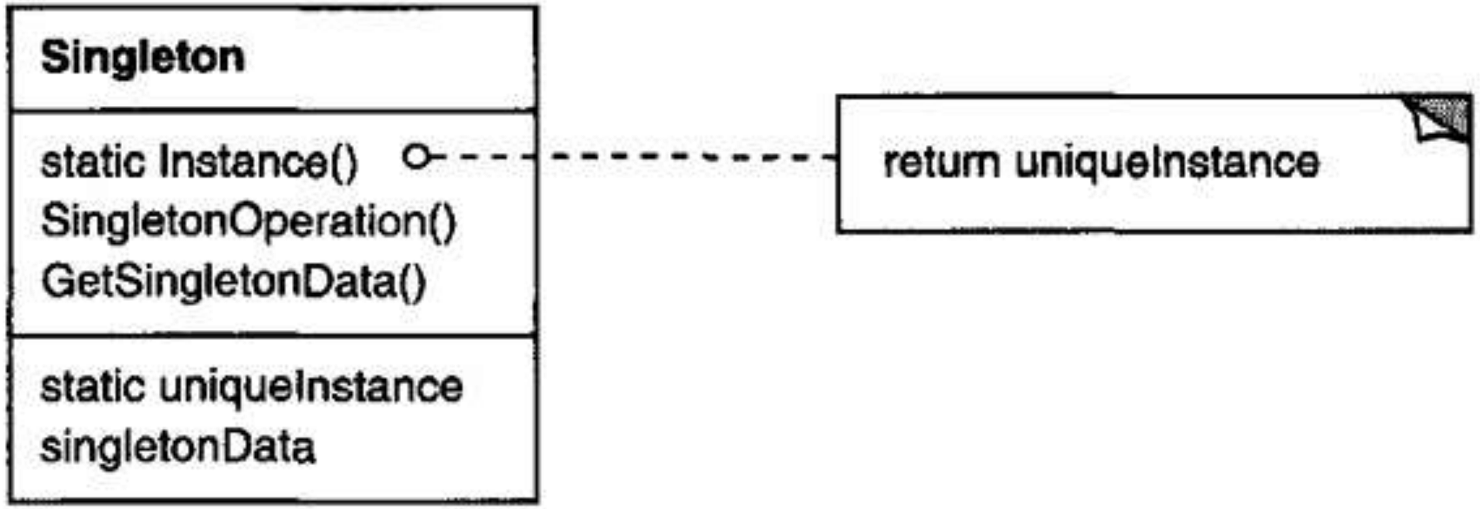
- The system requires a single control point to manage all devices and automation rules.
- If multiple instances of the controller exist, they may execute conflicting automation rules (e.g., one instance turns on a light while another turns it off).

Singleton pattern restricts a class to a single instance and provides global access to it. This ensures all automation rules and device commands are managed through one control point.

Problem #3 - Centralized Automation Controller

- The Singleton Pattern can be used here.
- The system requires a single Automation Controller to manage all devices, ensuring that only one instance of this controller exists at any given time.
- The HomeAutomationController class follows the Singleton pattern to ensure all requests (turning on lights, adjusting the thermostat) are managed centrally.
- Benefit: Prevents multiple conflicting automation routines from running simultaneously.

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Why Singleton?

- The Singleton pattern ensures that only one instance of HomeAutomationController exists at any time.
- This centralizes automation logic, preventing inconsistencies and ensuring all devices respond to a unified set of commands.
- It also saves resources by avoiding redundant controllers managing the same set of devices.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

Problem #4 - Flexible Automation Rules

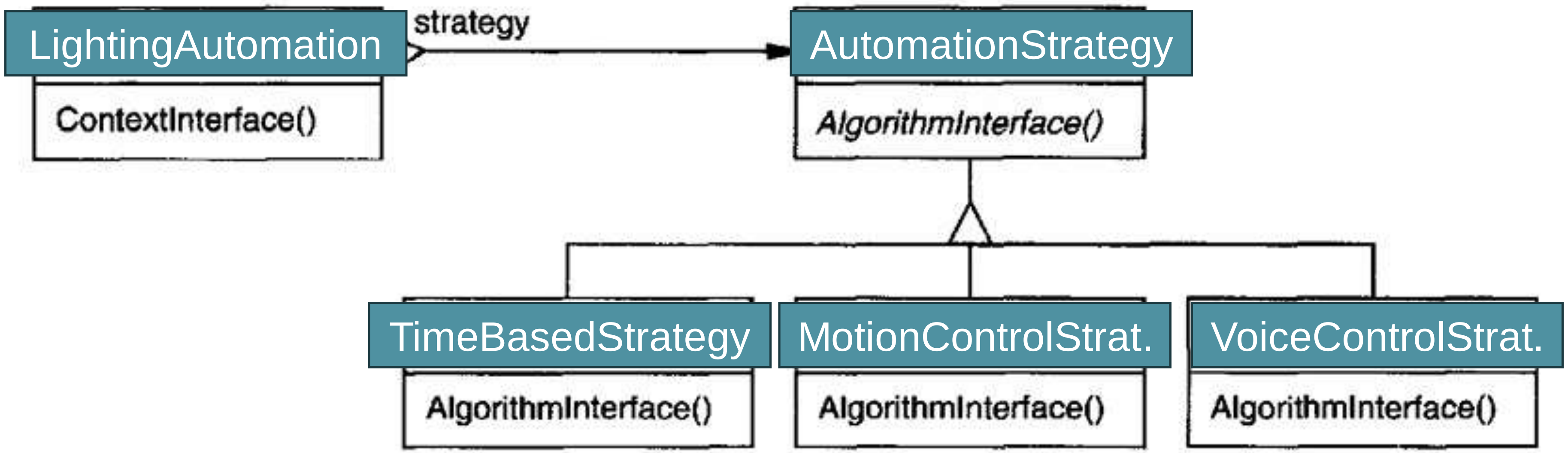
- Users want different automation behaviors (e.g., time-based scheduling, motion-activated lighting, voice control).
- If we implement these behaviors in a single class, it would become rigid and hard to modify (e.g., adding a new automation rule requires modifying a huge if-else block).

The Strategy pattern encapsulates each automation behavior into interchangeable classes, enabling runtime selection of rules.

Problem #4 - Flexible Automation Rules

- The Strategy Pattern can be used here.
- Users should be able to choose different strategies for automating their homes, such as time-based scheduling, motion-triggered activation, or voice control.
- Use Case: A LightingAutomation class allows different automation strategies (TimeBasedStrategy, MotionSensorStrategy, VoiceControlStrategy) to be selected dynamically.
- Benefit: Users can switch between automation rules without modifying core system logic.

Strategy



LightingAutomation
ContextInterface()
strategy
AutomationStrategy
AlgorithmInterface()
TimeBasedStrategy
AlgorithmInterface()
MotionControlStrat.
AlgorithmInterface()
VoiceControlStrat.
AlgorithmInterface()

Why Strategy?

- The Strategy pattern defines a family of automation algorithms (TimeBasedStrategy, MotionSensorStrategy, VoiceControlStrategy), encapsulating them into separate classes.
- The LightingAutomation class delegates the automation logic to a strategy object, allowing users to dynamically switch automation rules at runtime.
- This promotes open/closed principle (open for extension, closed for modification), making it easy to introduce new automation strategies without altering existing code.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

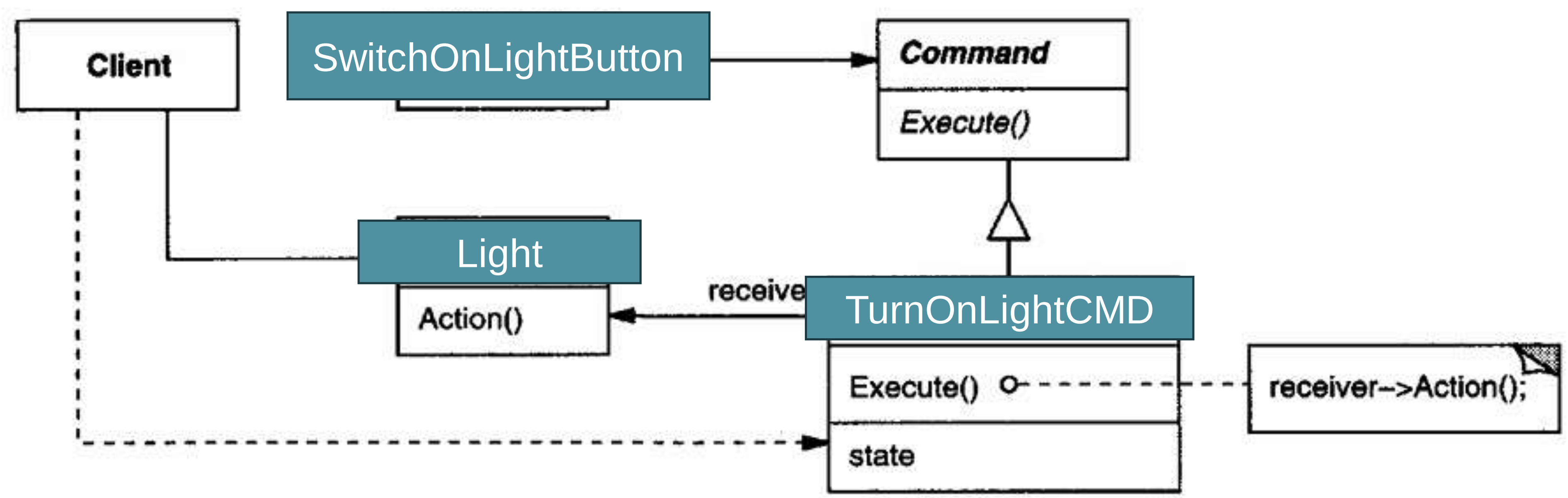
Problem #5 - Undoable User Actions

- Users need to undo or redo actions like turning on a light, adjusting the thermostat, or locking a door.
- If commands are executed directly (e.g., light.turnOn()), there is no way to store a history or revert actions.

Problem #5 - Undoable User Actions

- The Command Pattern can be used here.
- Users may want to undo actions like turning off the lights or adjusting the thermostat.
- The TurnOnLightCommand, AdjustThermostatCommand, and LockDoorCommand classes implementing the Command interface encapsulate user actions, which makes it possible to maintain a history for undo/redo functionality.
- Benefit: Enhances user experience by enabling a reversible command history. Also, the Invoker (App menu item/ button) can issue requests to objects (concrete Commands) without knowing anything about the operation being requested or the receiver of the request.

Command



Client
SwitchOnLightButton
Command
Execute()
Light
Action()
receive
TurnOnLightCMD
Execute()
state
receiver->Action();

Why Command?

- The Command pattern encapsulates actions as objects (TurnOnLightCommand, AdjustThermostatCommand).
- These commands can be stored in a history list, allowing users to undo or redo previous actions.
- This also decouples the UI from the device logic, making it easier to add new commands without modifying existing device classes.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

Problem #6 - Standardized Device Initialization

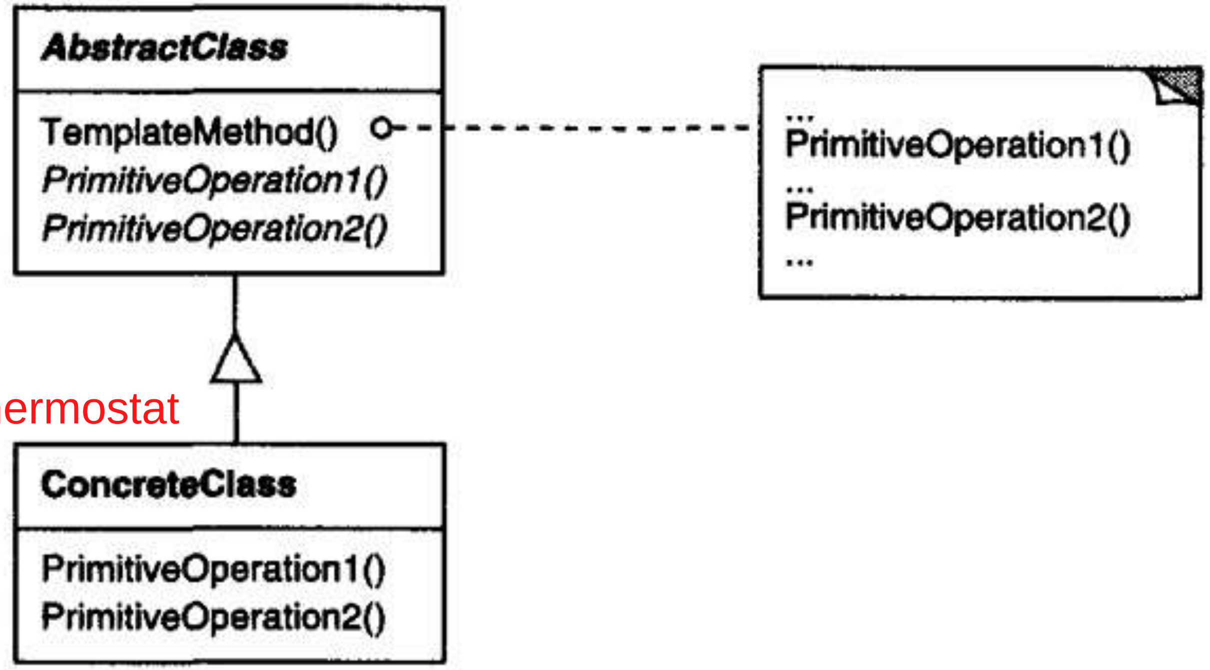
- All smart devices require a common setup process (e.g., authentication, network configuration, state synchronization), but specifics vary between devices.
- If each device implements initialization independently, there will be code duplication and inconsistencies.

Problem #6 - Standardized Device Initialization

The Template Method pattern defines a skeleton algorithm in a base class, allowing subclasses to override specific steps while preserving the overall structure.

- The Template Method Pattern can be used here.
- The SmartDevice abstract class defines a template method initialize(), with PhilipsLight and NestThermostat subclasses providing custom implementations for device-specific setup.
- Benefit: Ensures consistency while allowing customization where needed.

Template Method



Why Template Method?

- The Template Method pattern defines a skeleton (initialize()) in an abstract SmartDevice class, enforcing a standard initialization sequence.
- Concrete classes (PhilipsLight, NestThermostat) implement only the device-specific steps, ensuring consistency while allowing customization.
- This prevents duplicate setup code and ensures all devices follow a uniform onboarding process.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

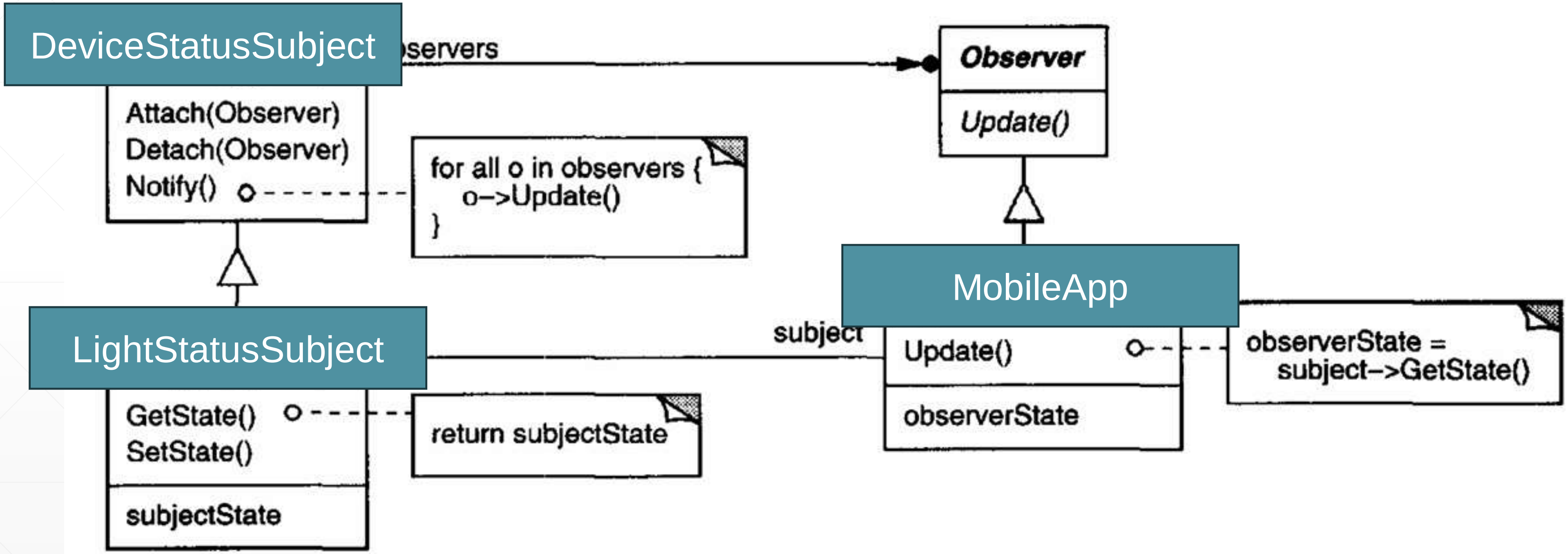
Problem #7 - Real-Time Device Status Updates

- Users expect real-time updates when a device's status changes (e.g., the thermostat updates its temperature, a camera detects motion).
- If the application continuously polls devices for updates, it wastes device processing power, and network bandwidth and introduces delays.

Problem #7 - Real-Time Device Status Updates

- The Observer Pattern can be used here.
- A DeviceStatus object acts as a subject, notifying observers (mobile app, web dashboard) whenever a device state changes.
- Benefit: Enables event-driven programming, reducing the need for constant polling.

Observer



DeviceStatusSubject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
LightStatusSubject
GetState()
SetState()
subjectState
return subjectState
subject
MobileApp
Update()
observerState
observerState =
subject->GetState()

Why Observer?

- The Observer pattern lets DeviceStatus act as a subject, notifying all registered observers (mobile app, web dashboard) whenever a change occurs.
- This enables event-driven programming, where updates are pushed to users instead of constantly polling for status changes.
- It makes the system scalable and more responsive to real-time events.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

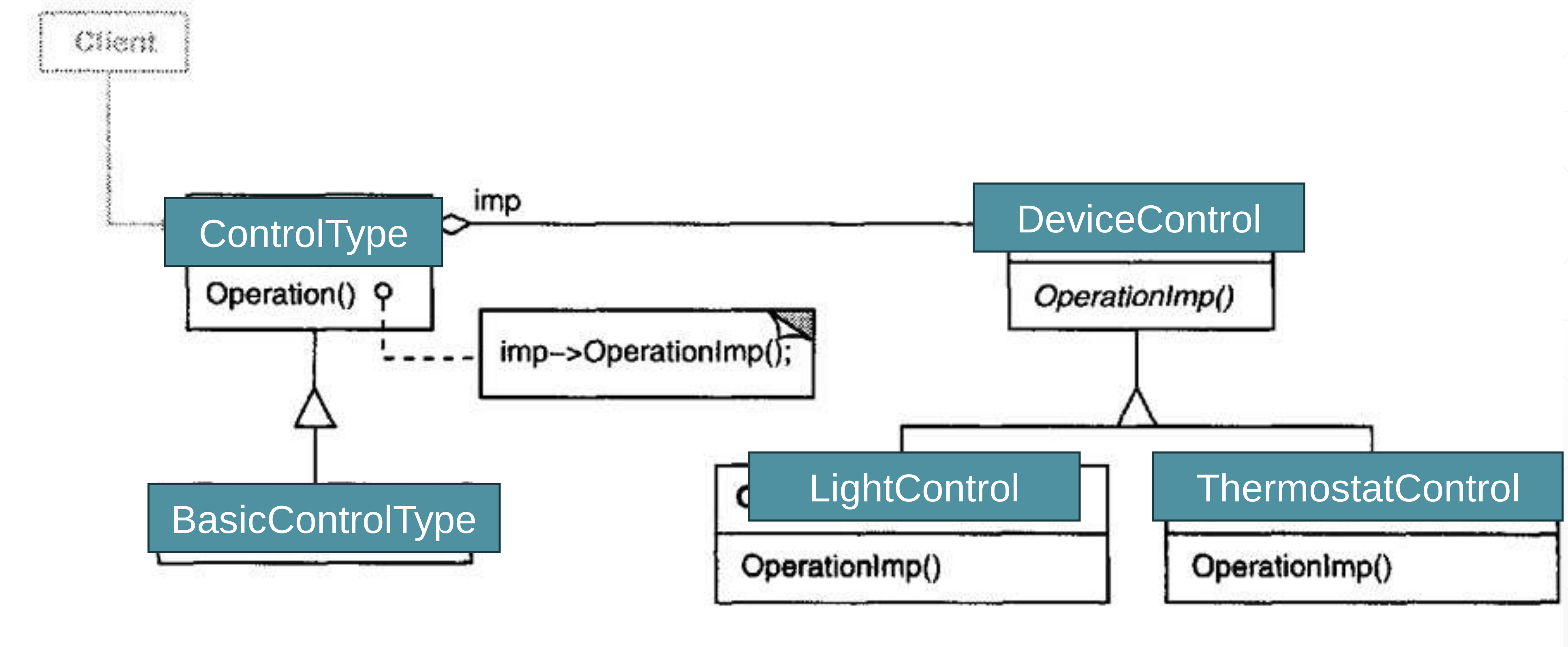
Problem #8 - Decoupling UI from Device Logic

- The system supports multiple levels of control (ControlType) depending on the service offering the customer has (i.e. BasicControl, Basic+Control, PremiumControl etc.)
- If the UI is tightly coupled with device logic for Control, then adding a new ControlType requires modifying the UI. Adding a new UI type (e.g., a smartwatch app) requires modifying all devices.
- Without separation, UI code will have to handle device logic directly, leading to duplicated and tightly coupled code.

Problem #8 - Decoupling UI from Device Logic

- The Bridge Pattern can be used here.
- A ControlType (Abstraction) class is bridged with DeviceControl Interface (Implementor) which is implemented by LightControl, ThermostatControl, and CameraControl concrete implementations, ensuring UI components remain independent of device-specific logic.
- BasicControl, Basic+Control, PremiumControl classes are the RefinedAbstractions.
- Benefit: Facilitates cross-platform compatibility without redundant code.

Bridge


Client
ControlType
Operation()
imp
imp->OperationImp();
DeviceControl
OperationImp()
BasicControlType
LightControl
OperationImp()
ThermostatControl
OperationImp()

Why Bridge?

- The Bridge pattern separates the UI from device-specific logic (LightControl, ThermostatControl, CameraControl).
- This ensures that UI components remain independent of backend logic, making it easier to support cross-platform compatibility.
- If new device types are introduced, UI changes are minimal, since the bridge provides a consistent interface.
- Refer the relevant chapter on the GoF Design Patterns Book for more details on this pattern.

Summary

Problem	Pattern	Solution
Hardcoding device brands makes it difficult to add new brands	Abstract Factory	Creates a factory interface to instantiate brand-specific devices without modifying existing code
Hardcoding protocol selection makes the system rigid	Factory Method	Encapsulates protocol selection logic in a factory method, allowing seamless integration of new protocols
Multiple automation controllers may cause conflicts	Singleton	Ensures a single instance manages all automation requests, preventing inconsistencies
Automation rules are hardcoded, making changes difficult	Strategy	Encapsulates different automation behaviors as interchangeable strategies
No way to undo or redo actions	Comman d	Encapsulates actions as objects, enabling command history and undo functionality
Device initialization is inconsistent and duplicated	Template Method	Defines a standardized setup process, allowing device-specific overrides
Devices need to send real-time updates, but polling is inefficient	Observer	Enables event-driven updates by notifying observers when device states change
UI is tightly coupled to device specific logic, making cross-platform support difficult	Bridge	Decouples UI from device specific logic, ensuring flexibility and maintainability

The End