Strategy

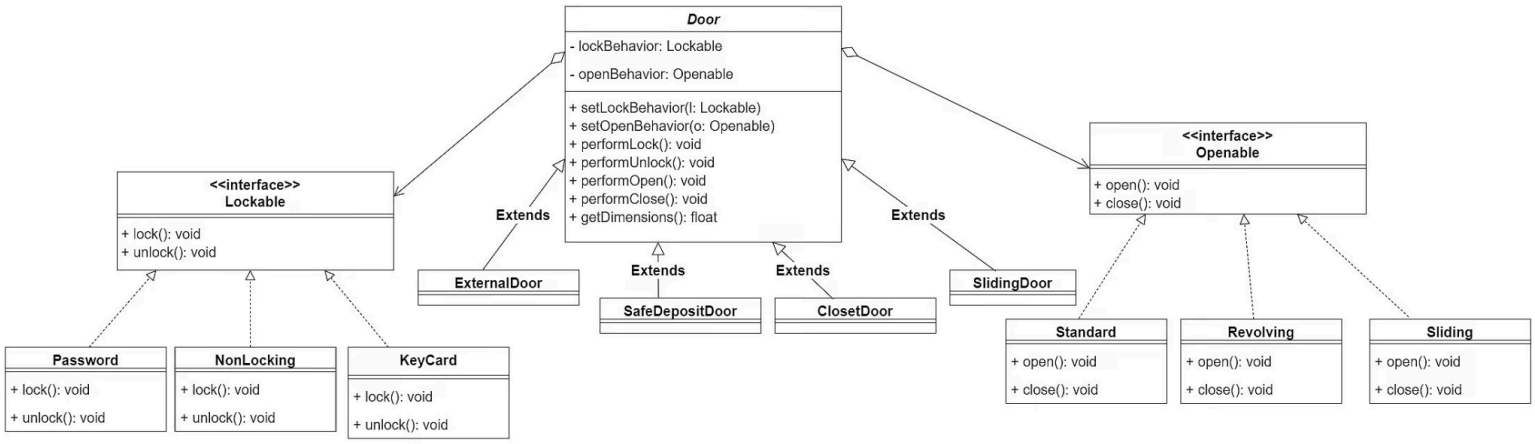
Define a family of algorithms that can be chosen at runtime, client code is flexible!
Motivation: multiple concrete objects may share the same methods, better to store the method and reuse it than implement it for each concrete object internally

Solution:

1. Leave all implementation for the subclasses - use interfaces that declare methods instead of implementing methods at the base class
2. Encapsulate what varies - identify parts of the app that vary (interfaces) and separate them from constants (base class)

Design + Implementation:

1. Context (navigator class) - delegate responsibility to the strategy provided
2. Strategy interface - blueprint ensuring consistency across all strategies
3. Concrete Strategies - define the specific behavior or algorithm
4. Client - interacts with the context and decides which concrete strategy to use



Limitation + Pitfalls:

1. Increased Number of Classes - each new strategy requires a new interface
2. Risk of Over-Engineering - might be simpler to use conditional code to specify strategy

Closing Notes:

1. Separation of Concerns
2. Interchangeable Strategies
3. Composition over Inheritance

Observer

To send notifications through our code which are triggered by certain state changes.

Motivation:

Polling - system maintains info and clients initiate requests to retrieve info (inefficient)

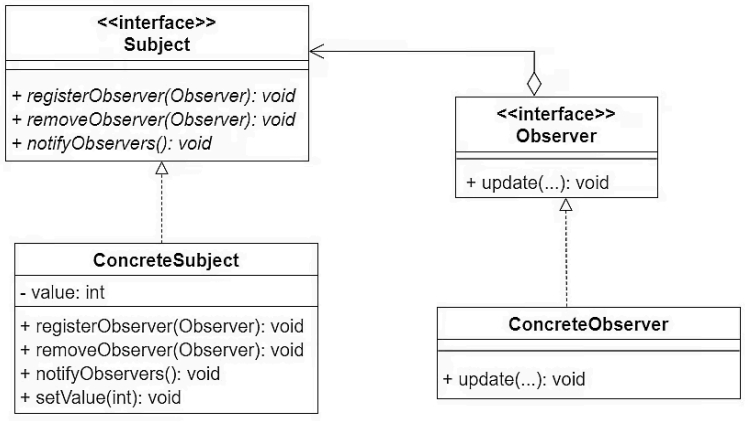
Solution:

Loose Coupling - the components or the classes within a system are designed to have as little dependency on each other as possible.

1. **Subscription** - observers subscribe to receive notifications
2. **Change** - when change occurs, system automatically sends notification to subscribers
3. **Notification** - observers receive notification from the system

Design + Implementation: (Inversion of Control)

1. **Subject (Publisher)** - This interface/abstract class specifies the methods that our concrete subject will implement.
    - **registerObserver** - allows the observers to subscribe to event changes
    - **removeObserver** - allows the observers to unsubscribe to event changes
    - **notifyObservers** - makes sure all observers are notified once the data in the subject changes
2. **Concrete Subject** - implements Subject interface, Concrete Subject manages a list of observers and will manage the data that the observers are interested in observing the changes for.
3. **Observer (Interface/Abstract class)** - This interface/abstract class declares the update method which is called to inform the observer of changes in the subject.
4. **ConcreteObserver** - concrete class implementation which implements the Observer and contains a reference to the Subject via composition.



Technically there's the Push Model (shown above) and a Pull model.

Limitation + Pitfalls

1. Processing Overhead - too many observers or computationally expensive notification operations may be costly
2. Unintended effects - order which observers are notified isn't explicit, debug flow is hard

Closing Notes:

1. Loose Coupling - Subjects and Observers interact but have little knowledge of the other
2. Inversion of Control - observer patter flips the control mechanism between objects

State

allows an object to alter its behavior based on its internal state. Each state behavior is encapsulated inside of a class, which adheres to a State interface.

Motivation:

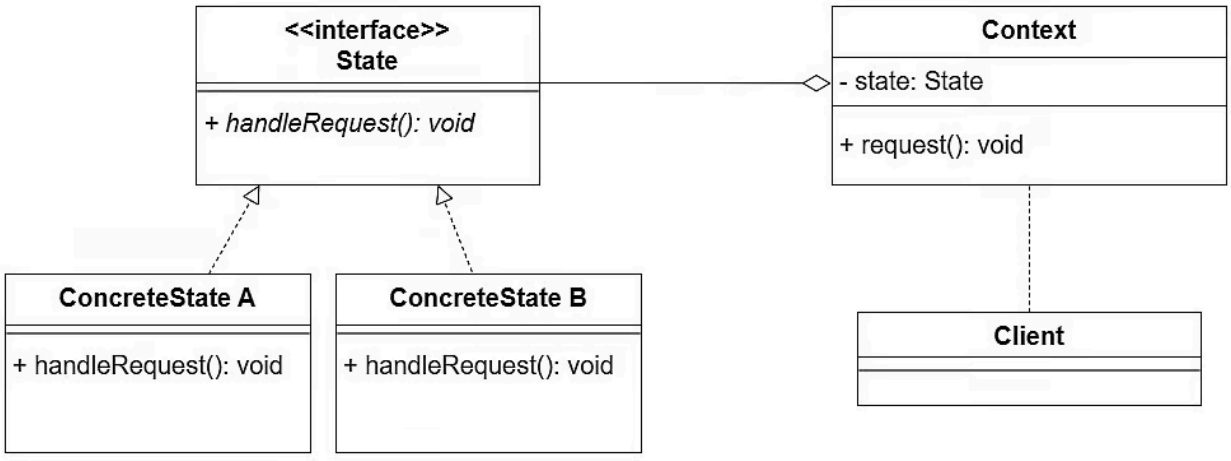
When adding new states introduces impractical complexity towards conditional code logic, its better to use a state pattern

Solution:

1. Object can change its state dynamically in response to a change in its internal state
2. Encapsulate state-specific behaviors into a state class which implements a state interface
3. Single Responsibility Principle - each state class is responsible only for the behavior associated with its specific state, code is easier to manage and extend

Design + Implementation:

1. Context - represents the system, references the current state object, provides and interface for triggering state transitions and delegates behavior associated with each state to the corresponding state object.
2. State Interface - defines common set of methods that all concrete state classes must implement, methods represent actions or events relevant to the system
3. Concrete State Classes - implements state interface and encapsulates behavior specific to a particular state of the traffic light.
4. Client - interacts with the context to initiate state changes, state transitions and internal logic are abstracted away from client and handled by the state themselves to implement



Limitation + Pitfalls

1. Potential of Many Classes - complex systems may have many states
2. Risk of Tight Coupling - if state pattern allows state to know about the context and its other states,

Closing Notes: