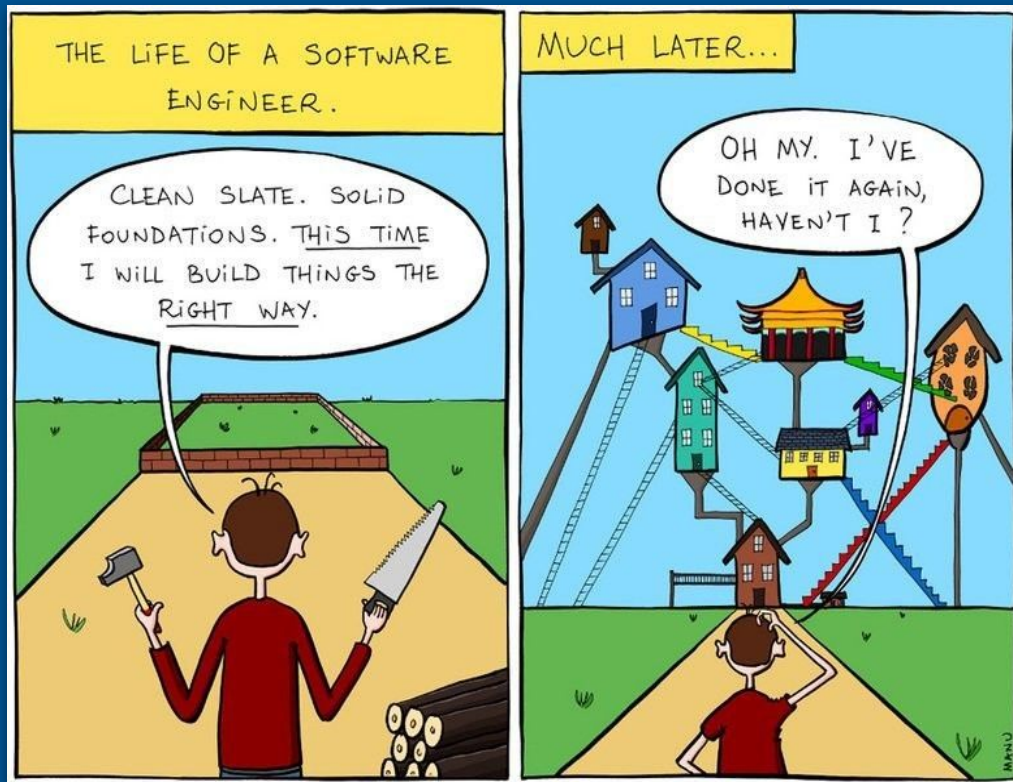


For personal use only,
please do not
distribute on
non-UBC domains.

06a-Patterns III

CPSC310

Nick Bradley & Kyle Chin

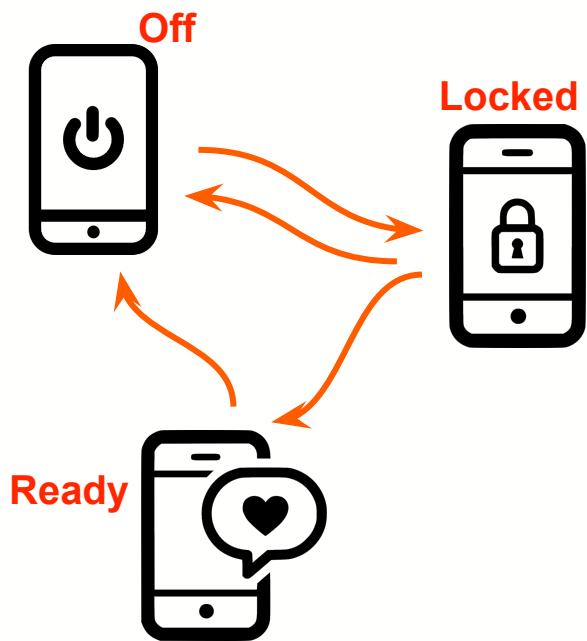


Learning goals

By the end of this lecture, you should be able to:

- Identify a decision in existing code that is likely to change, and where knowledge of it is scattered or tangled.
- Explain a design pattern in terms of the change, the pain, the mechanism and the cost.
- Refactor unfamiliar code to a pattern in small steps that keep its behaviour the same (today: State).
- Evaluate whether a pattern is worth its cost in a particular situation.

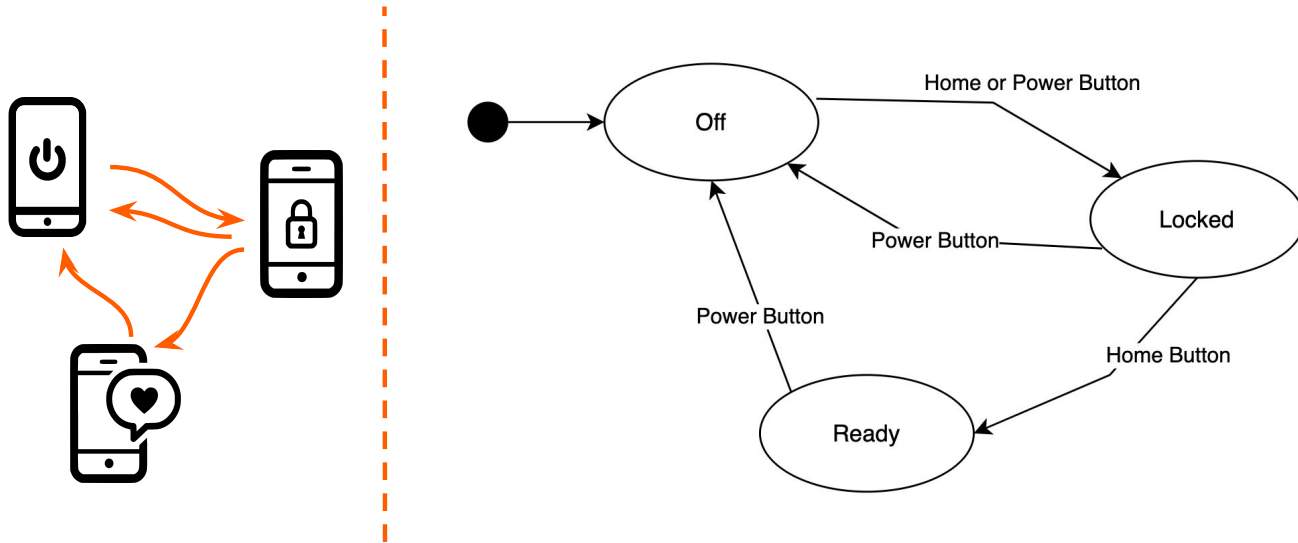
Scenario



You're working on the operating system for a new phone. The phone can be **off**, **locked**, or **ready**. Pressing the power or home button, or receiving a notification each work differently depending on state.

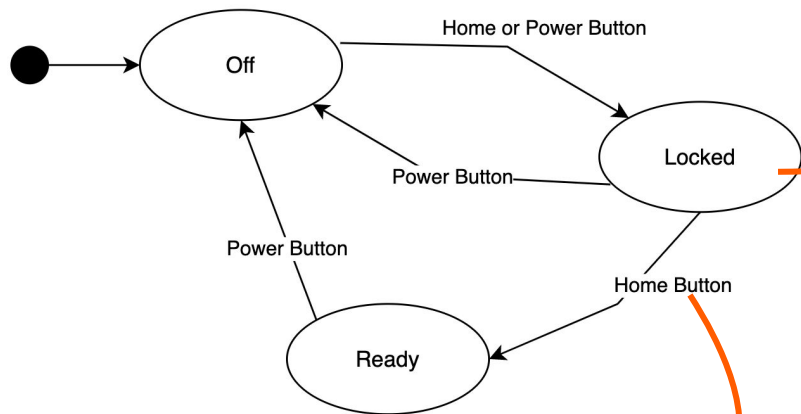
State Design Pattern

Intent: An object behaves differently depending on what state it's in, and its state changes while the program runs. Without the pattern, every method ends up with a switch on the current state to decide what to do.



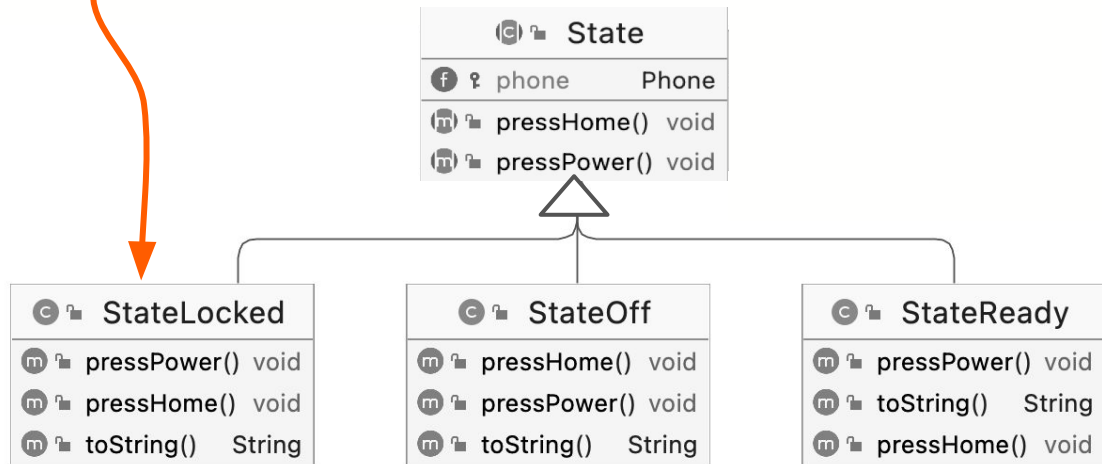
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. The result is an object-oriented state machine.

Refactor: Encapsulate States in Objects



States become objects.

Transitions become methods.



State: Canonical Depiction

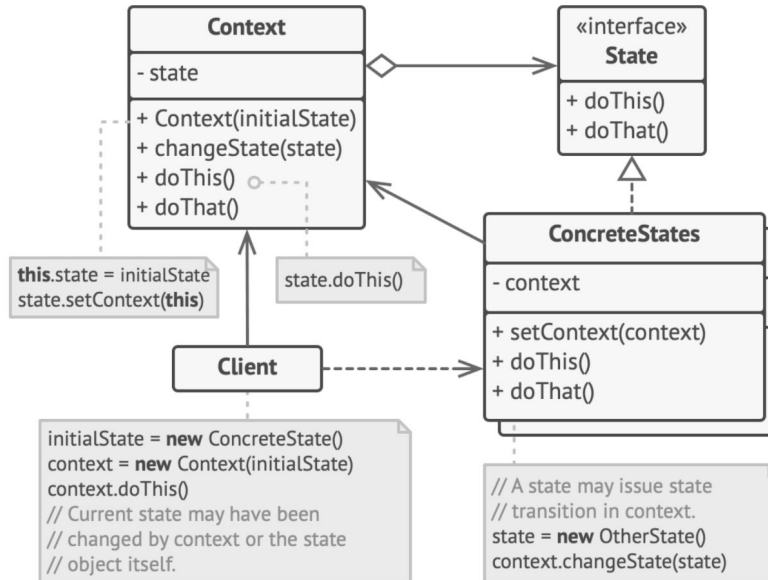
1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.

3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.



State: Analysis

1. **Change:** new states the phone can be in, each with its own rules
2. **Pain:** one case per state in every event's switch (scattered); each event mixes every state's rules and transitions (tangled)
3. **Mechanism**
 - a. Abstraction: PhoneState interface → `pressPower()`, `pressHome()`, `receiveNotification()`
 - b. Oblivious caller: Phone only knows its current PhoneState; each state handles every event and decides what comes next
 - c. Trigger: each Phone event is handed to the current state, which calls `phone.setState()` to move on
4. **Cost:** one class per state; transitions spread across the state classes; a new event touches every state; `setState()` is public

Synthesis: Design Patterns

Each pattern puts an **interface** between the client and the thing that varies. A new variant is a new class; the client **doesn't change**.

Pattern	Interface	New Variant	Unchanged
Adapter	MusicSource	YouTube Music	Reports
Composite	Employee	DataScientist	Manager, org totals
Decorator	Drink / AddOn	Mocha add-on	Menu drinks, cart
State	PhoneState	Lockout state	Phone

Open/Closed Principle (OCP)

A unit of code (function, class, module) should be:

- Open for extension: new behaviour can be added
- Closed for modification: adding the behaviour doesn't change code that already works.

How: depend on an **abstraction**, then extend by writing a new implementation of it. The four patterns are four ways of putting the abstraction in place.

Axis of Change: Phone States

No design is open to every change.

	New State (Lockout)	New button (Volume)
switch (before/)	Edit every method	Add two methods to Phone
State (after/)	Add one class	Edit PhoneState and every class

Every Pattern has a Closed Axis

Open to new implementations, closed to new operations.



Pattern	Cheap	Expensive (touches every class)
Adapter	new music service	new query on MusicSource
Composite	new role	new operation (e.g., avg salary)
Decorator	new add-on	new property (e.g., calories)
State	new state	new button

OCP: Choosing the Axis

Every extension point costs **indirection**: an interface, a dispatch, more classes. Spend it only where variation will actually arrive.

Before adding an extension point, ask:

- What will vary? (new services, roles, add-ons, states...)
- What must stay stable for the code that depends on it?
- Is that variation *really* likely?