



SWE 621

FALL 2018

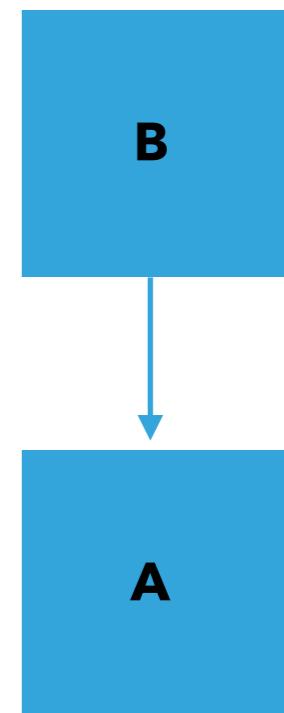
FOLLOWING A DESIGN

LOGISTICS

- ▶ HW4 due next week
- ▶ No class meeting next week
 - ▶ Lecture will be recorded and posted online

IN CLASS EXERCISE: INVERSION OF CONTROL

- ▶ Problem: imagine a layered architecture, where the low level A layer should not depend on higher-level B. But when some event in A happens (e.g, network packet arrives), B should find out.
- ▶ Propose a design to achieve this.

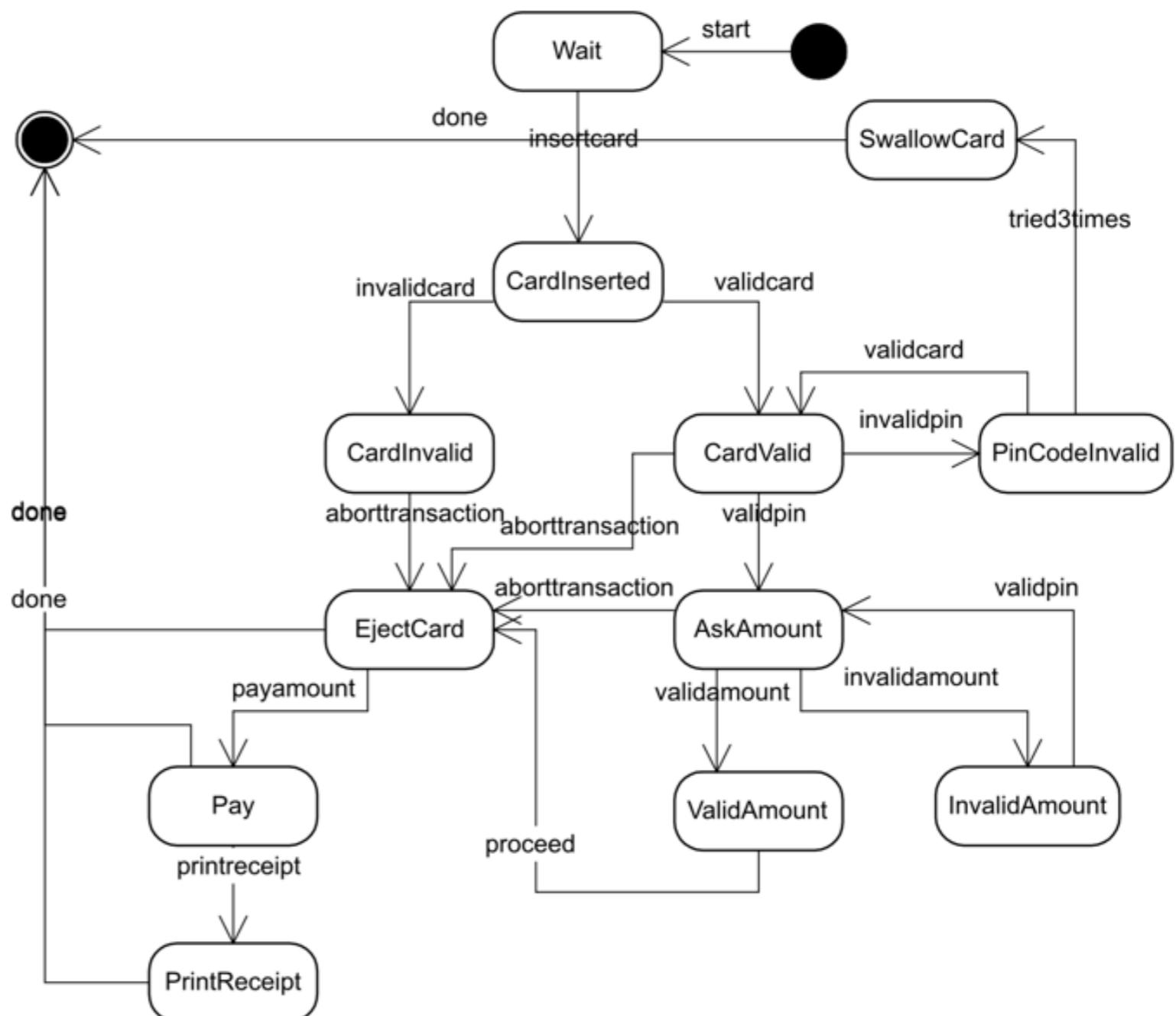


FOLLOWING A DESIGN

- ▶ So far we've considered how design choices can help system achieve quality attributes
 - ▶ abstractions, architectural styles, design patterns
 - ▶ by minimizing risk, by following domain model, hiding decisions likely to change
- ▶ What happens when a developer makes a code change that **fails** to follow the constraints imposed by the design decision?
 - ▶ How do you **prevent** developers from not following design decisions?
 - ▶ What happens when the design decision should change?
 - ▶ Requirement changes may lead to decisions no longer being effective.
 - ▶ May find better design choices as better understand problem.

EXAMPLE: HOW SOFTWARE EVOLVES OVER TIME

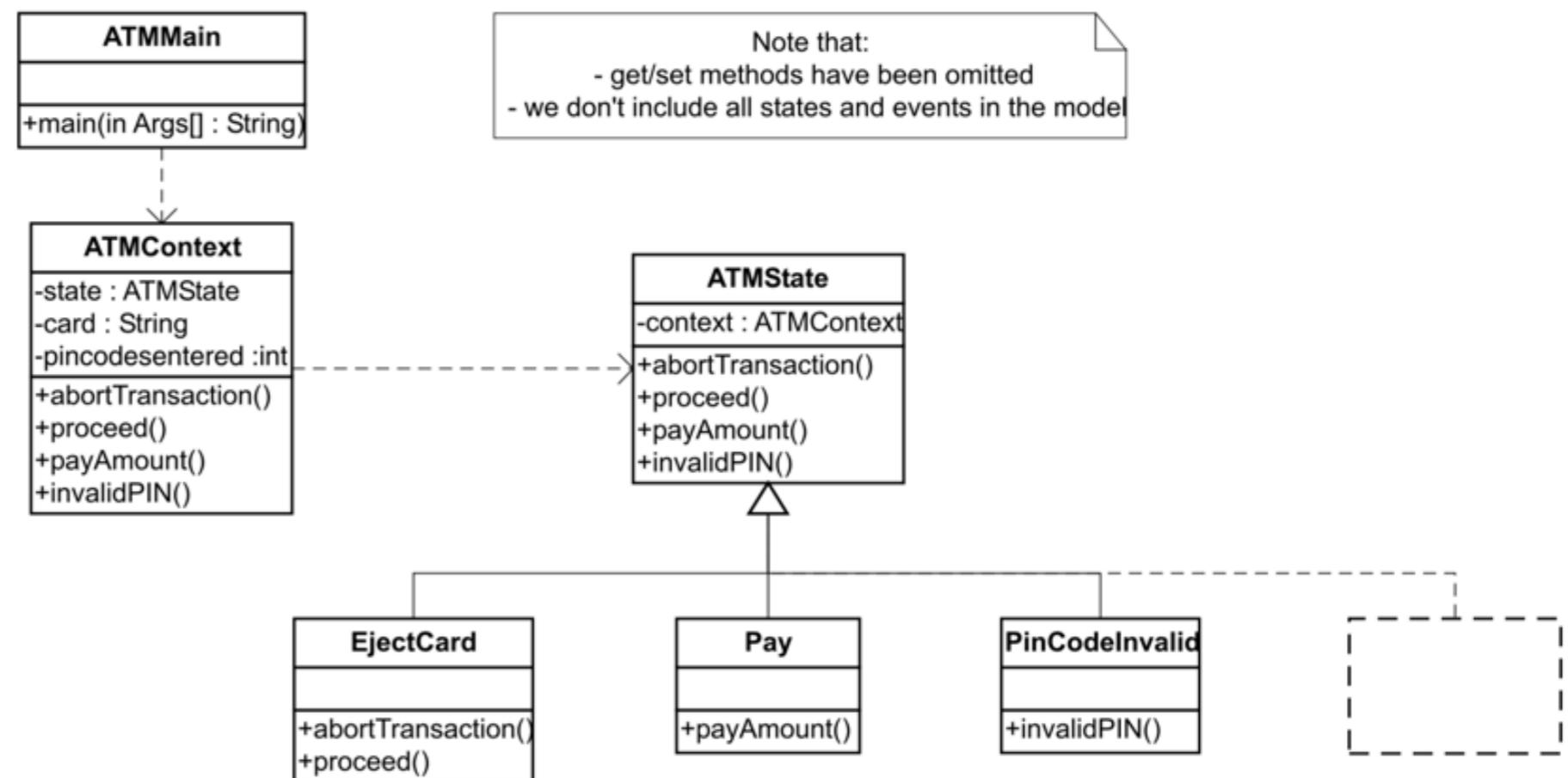
- ▶ ATM Simulator
 - ▶ Describes behavior of ATM machine as user interacts with machine



V1: STATE PATTERN

Decisions

- ▶ Use the state pattern
- ▶ Put data in context class
- ▶ Make context a property of ATMState
- ▶ Use command line for UI



V1: STATE PATTERN

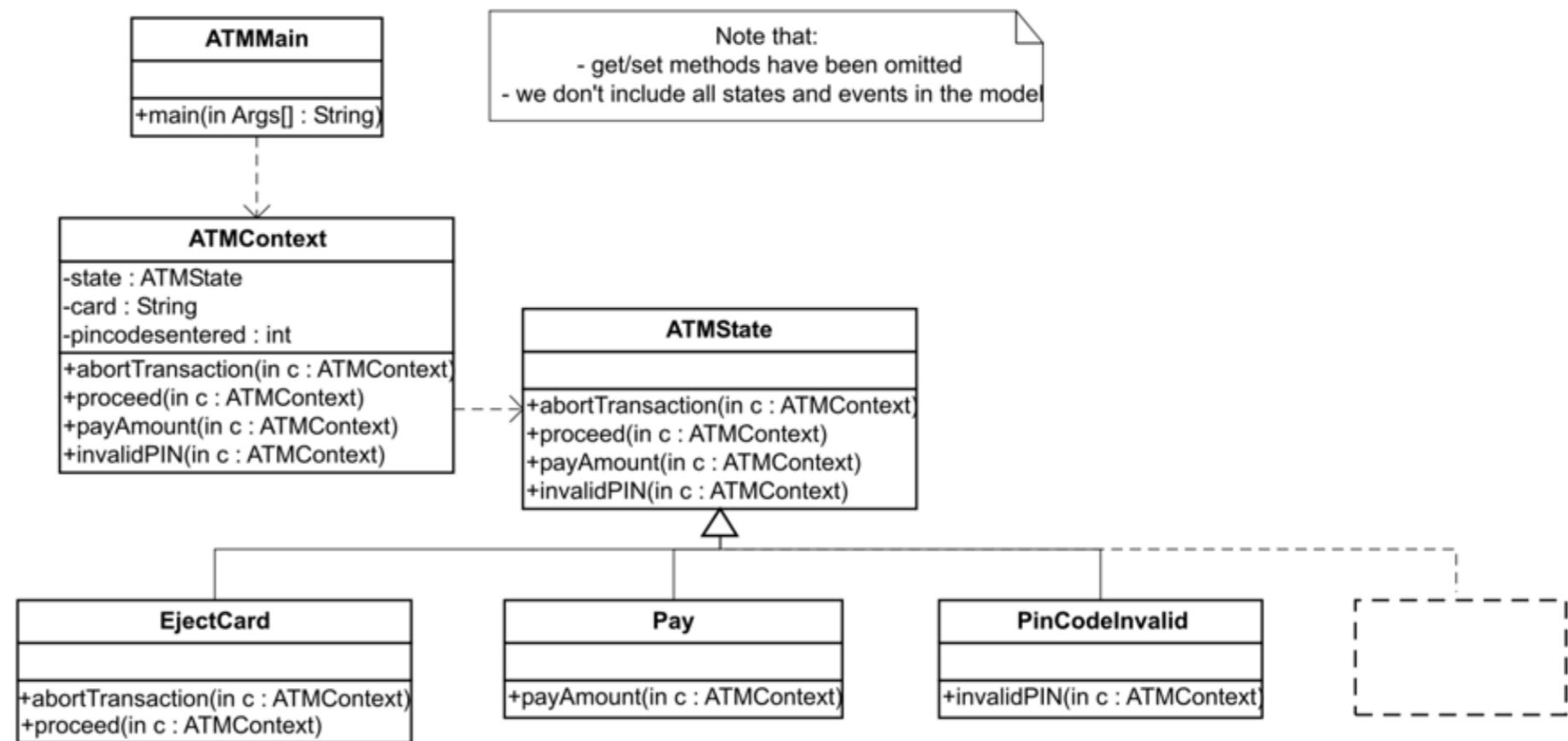
- ▶ ATMContext stores variables used by ATMState subclasses
 - ▶ Need to be shared between subclasses
 - ▶ Everything needs references to context class
- ▶ ATMContext contains many methods that only forward the call to the current state
- ▶ ATMContext does not check whether a particular event is supported by the current state
 - ▶ Potential for defects

V2: FLYWEIGHT

► Goals

- Memory usage: instantiate each state class only once

- Performance: reduce startup time for simulator



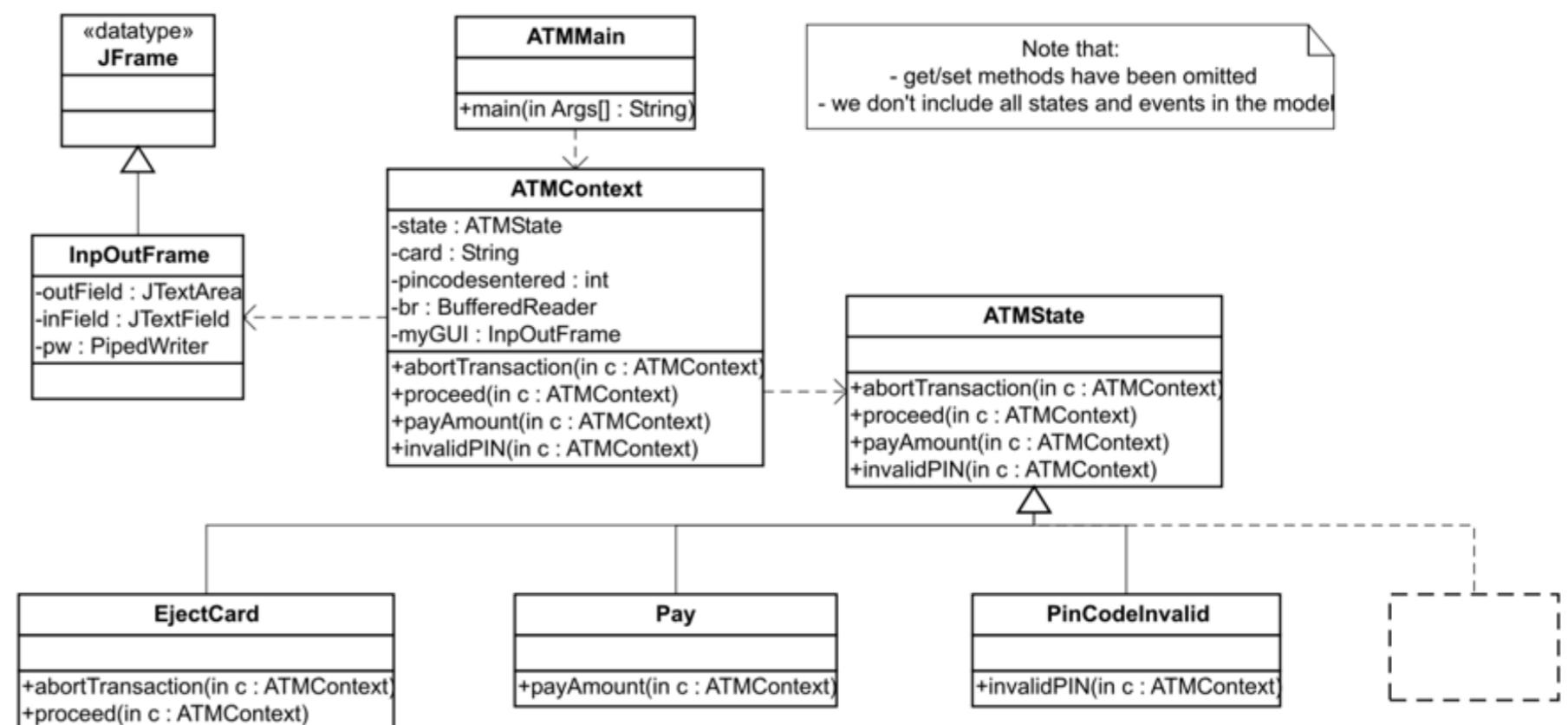
V2: FLYWEIGHT

- ▶ Each state class is only created once
- ▶ Removed the context property from ATMState, added context parameter in each event method

V3: MULTIPLE INSTANCES

► Goals

- Parallelism: enable each simulator to run in a separate thread
- UI: support multiple simulators



V3: MULTIPLE INSTANCES

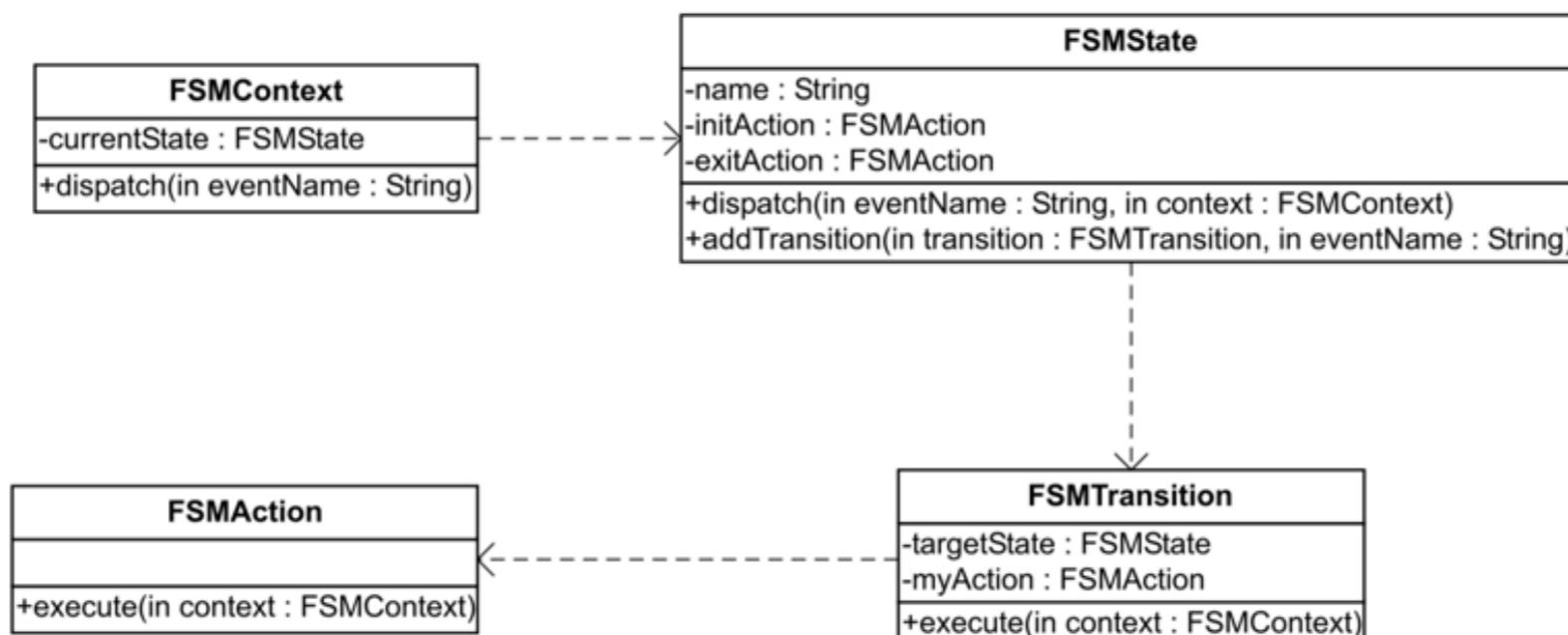
- ▶ Replaced command line with GUI, each containing multiple windows
- ▶ Each window associated with ATMContext
- ▶ GUI connected to ATMContext with pipes and filters
 - ▶ Whenever a user enters data, can read from IOStream from GUI just as if it were the command line

V4: DELEGATION-BASED APPROACH

- ▶ Goals
 - ▶ Configurability: allow for adding new states and transitions at runtime (e.g., machine runs out of paper)
 - ▶ Separation of concerns: decouple state machine further

V4: DELEGATION-BASED APPROACH

```
public class ATMSimulator extends FSMContext {  
    static FSMState ejectcard = new FSMState("ejectcard");  
    static FSMState pay = new FSMState("pay");  
    static FSMState pincodeinvalid = new FSMState("pincodeinvalid");  
    static FSMState cardvalid = new FSMState("cardvalid");  
    ...// more state definitions  
    static { // static -> it's executed only once  
        pincodeinvalid.setInitAction(  
            new AbstractFSMAction() { // Inner class definition  
                public void execute(FSMContext fsmc) {  
                    ...// desired behavior  
                }  
            }  
        );  
        pincodeinvalid.addTransition(cardvalid, new DummyAction(), "validcard");  
        ...// more transition and action definitions  
    }  
    ...//rest of the class  
}
```



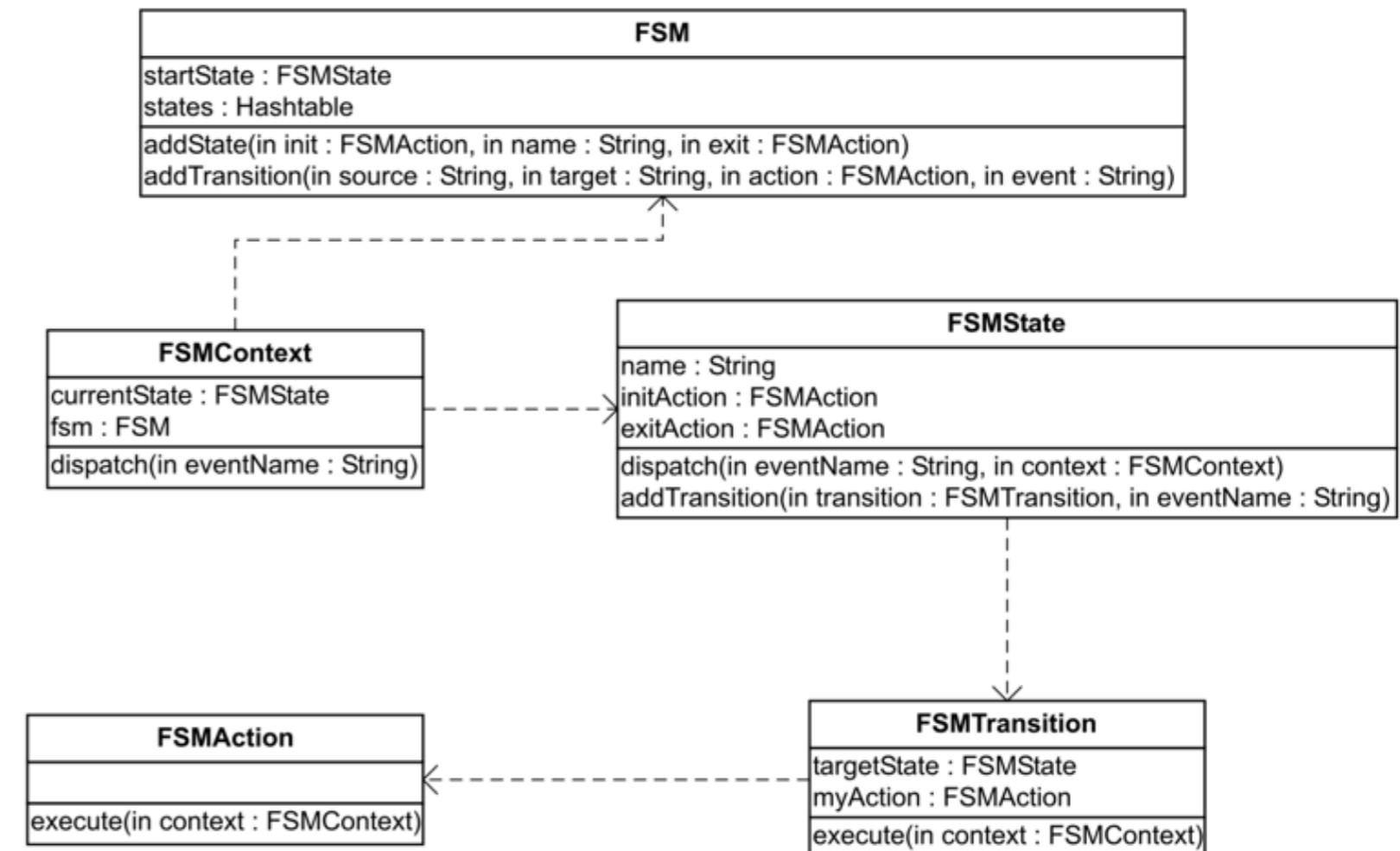
V4: DELEGATION BASED APPROACH

- ▶ User delegation rather than inheritance
 - ▶ States no longer subclass FSMState
 - ▶ Transitions are now first class
 - ▶ Transitions delegate behavior to Action

V5: DECOUPLING

► Goals

- Reduce use of static



- Introduce FSM, which separate responsibility of storing FSM from dispatching events

SUMMARY OF EVOLUTION

▶ Later decisions revised earlier

Version	Decision	Effect on system
v1	1.1 Use the State pattern 1.2 Put data in context class 1.3 Make context a property of ATMState 1.4 Use command line for UI	For each state in a FSM, a subclass of State has to be created Each event method in the State subclasses refers to the Context class to access data The context is available to all State instances The code is littered with calls to System.in and System.out
v2	2.1 Make instances of State static 2.2 Remove context property from ATMState and use parameter in event method instead	The keyword static needs to be put before instantiations of State subclasses All event methods need to be edited
v3	3.1 Create a GUI 3.2 Replace System.in and System.out calls with calls to the GUI 3.3 Apply the pipes and filters for communication between GUI and simulator	A class is added to the system All event methods need to be revised
v4	4.1 Refactor the system to use delegation (Van Gorp and Bosch, 1999). 4.2 Use the command pattern to separate behaviour from structure 4.3 Introduce state exit and entry events to the FSM model	New classes are created that model the behaviour of states and transitions. All existing State subclasses are removed from the system. For each event method in the State subclasses, an inner class needs to be created that implements the FSMAction interface. An instance of such classes needs to be associated with the appropriate transition(s) The event dispatching mechanism needs to be changed to support this type of events
v5	5.1 Introduce factory classes for states and transitions	A new class is created. The initialisation code for FSMs can be made non static and becomes much simpler

SUMMARY OF EVOLUTION

- ▶ Design decisions changed over time
 - ▶ Driven by making a particular usage or scenario easier
 - ▶ Reasons may not be apparent without knowing these scenarios
- ▶ Easy to lose track of decisions
 - ▶ Constant change makes it harder to stay up to date with the current version of each design decision
 - ▶ Risk that might make change inconsistent with design
 - ▶ Risk that when changing a decision might not update everything required

SOFTWARE EVOLUTION

- ▶ As requirements are added and change, code must implement these changes.
- ▶ This requires making changes to system that are either
 - ▶ consistent with the existing design
 - ▶ changing decisions to better accommodate these new requirements, updating the relevant implementation

ARCHITECTURAL EROSION

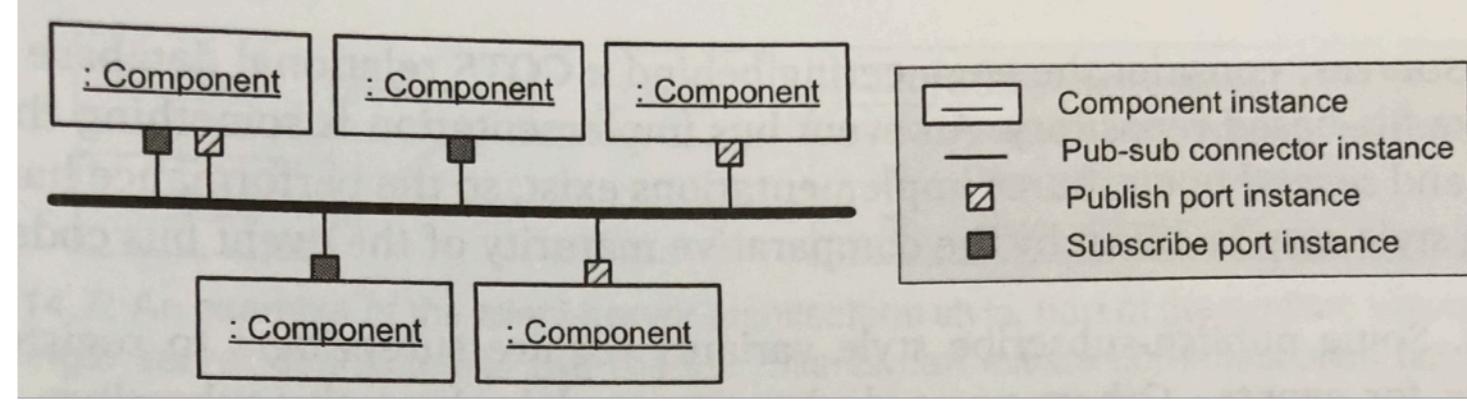
- ▶ Software architectural erosion (or decay): the gap between the architecture **as designed** as an **as built**
 - ▶ e.g., intended to be a pipes and filters architecture, but isn't entirely
- ▶ Consequences of design decision are no longer achieved
 - ▶ if decision helped enable maintainability, it does not longer
- ▶ May lead to behaviorally observable defects, but not necessarily

CODEBASES TEND TO DECAY OVER TIME

- ▶ Study of large software system, as observed through commit data
- ▶ Over time
 - ▶ Increase in # of files touched per commit
 - ▶ Increase in # of modules touched per commit
 - ▶ These increases lead to increased effort to make change
 - ▶ Relationship between edits and defects introduced

S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng. (TSE)*, 27(1):1–12, Jan 2001.

AN EXAMPLE



- ▶ You've built a system following the publish / subscribe architectural style.
- ▶ Wanted to enable adding and removing components without impacting existing code
- ▶ Constraints
 - ▶ Components do not know why an event is published
 - ▶ Subscribing components do not know **who** published event, depending on event type rather than specific publisher

IN CLASS ACTIVITY

- ▶ Imagine a publish subscribe system which contains the following events
 - ▶ UserInput, ScreenResize, AppStart, AppClosing
- ▶ Imagine a developer who implements functionality which should execute whenever the screen resizes.
 - ▶ To do this, they look for a message from the RenderLoop class rather than looking for a ScreenResize event.
- ▶ What are potential consequences of this?

TECHNICAL DEBT

- ▶ Sometime you know that you've broken the design, but still decide to do it anyway.
- ▶ Why? Schedule pressure.
- ▶ But.... then have to live with the consequences
 - ▶ Changes get more expensive

MANAGING TECHNICAL DEBT

- ▶ Debt metaphor: deferred some of the work necessary to complete changes to the future
 - ▶ It passes these tests, but violates design principles that enable extensibility and maintainability.
- ▶ Need to have a plan to pay down debt.
- ▶ Plan work to improve design to make it again consistent with design.

WHAT TO DO ABOUT CODE DECAY?

- ▶ Prevent code decay
 - ▶ Better communicate design to developers
 - ▶ Check that changes are consistent with design
- ▶ Fix code decay after it occurs
 - ▶ Refactor code to be consistent with design
 - ▶ Change code to be consistent with design changes

BETTER COMMUNICATE DESIGN TO DEVELOPERS

- ▶ How does a developer know that there's a design decision they should follow?
 - ▶ Ask a teammate
 - ▶ Read a comment
 - ▶ Read documentation
 - ▶ e.g., in our codebase, we only create element x by doing y.

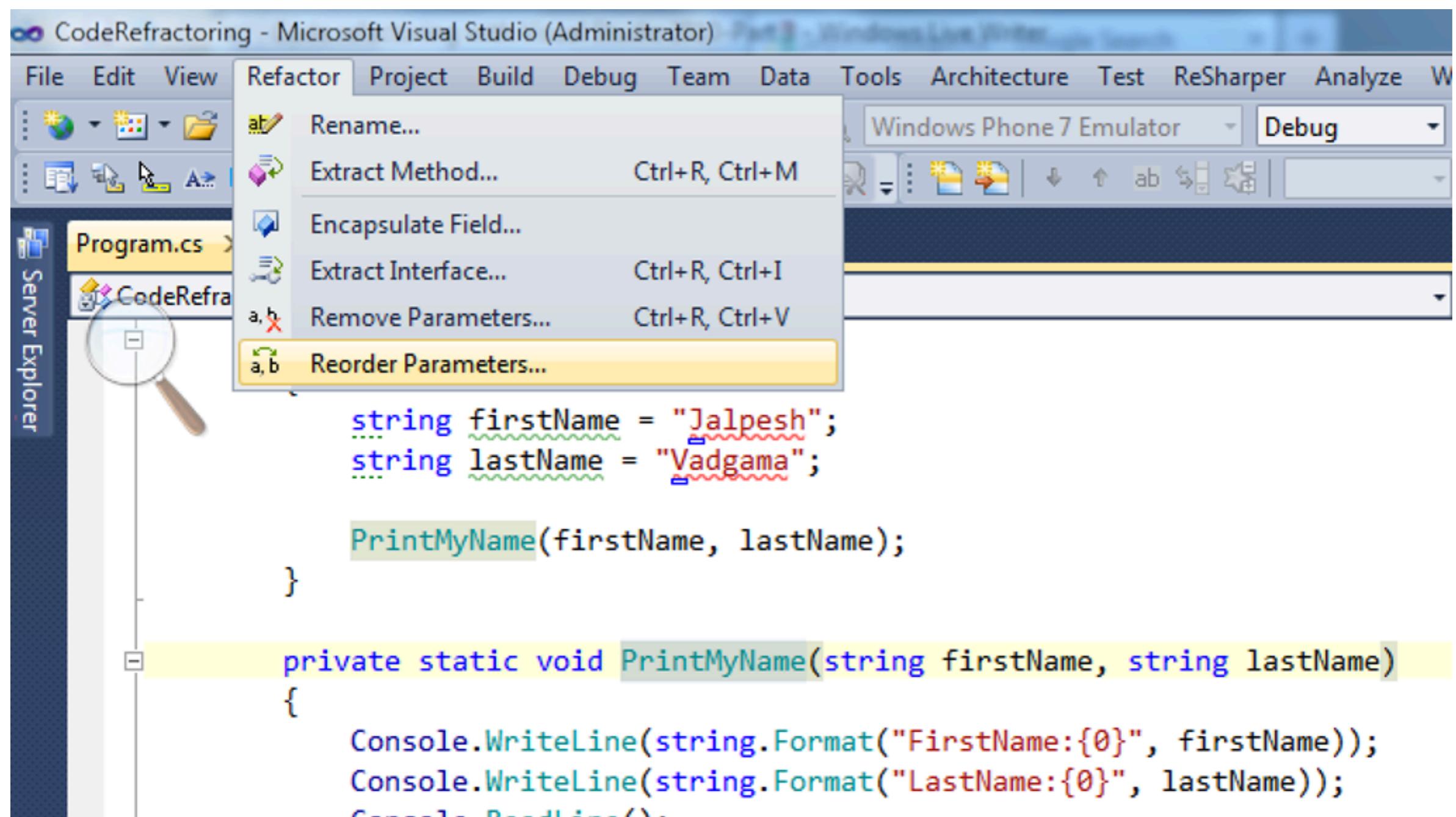
CHECK THAT CHANGES ARE CONSISTENT WITH DESIGN

- ▶ Code reviews offer important quality gate
- ▶ Before any change is committed, another developer must review the a delta of the code change
- ▶ That developer looks for potential defects in the code as well as violations of design decisions.
- ▶ Gives comments, which original developer must then fix before code is committed

FIX CODE DECAY AFTER IT OCCURS

- ▶ Make changes that improve the **design** of the code without changing the **behavior**: refactoring
 - ▶ Goal: before and after change, code should behave **exactly** the same
- ▶ Involves moving and renaming functionality
- ▶ Modern IDEs support automatic low-level refactorings
 - ▶ e.g., move method.
 - ▶ Finds references to functionality and updates
 - ▶ Tries to guarantee that defects are not inserted.
- ▶ Often need to make many low-level changes to achieve higher-level goal
 - ▶ Many may not be supported directly through automated refactoring

EXAMPLE: REFACTORING SUPPORT



SOME EXAMPLES OF REFACTORINGS

- ▶ Encapsulate field - force code to access the field with getter and setter methods
- ▶ Generalize type - create more general types to allow for more code sharing
- ▶ Replace conditional with polymorphism
- ▶ Extract class: moves part of the code from an existing class into a new class.
- ▶ Extract method: turn part of a larger method into a new method.
- ▶ Move method or move field: move to a more appropriate class or source file
- ▶ Rename method or rename field: changing the name into a new one that better reveals its purpose
- ▶ Pull up: move to a superclass
- ▶ Push down: move to a subclass

SUMMARY

- ▶ As software evolves, its requirements may change, necessitating changes to implementation
- ▶ Code that is inconsistent with design introduces code decay, where expected consequences of design decisions are no longer realized
- ▶ Code decay makes code harder to change and can lead to defects
- ▶ To reduce code decay, important to prevent code decay and fix it when it occurs

IN CLASS ACTIVITY

IMPLEMENT COMPOSITE

- ▶ Form group of 2 or 3, pick an OO language (e.g., Java, C++, Python)
- ▶ Start with V5 ATM implementation
- ▶ Goal: make it possible to have multiple ATM implementations for separate ATM machines.
 - ▶ Clients should be able to request an ATM be created without having to depend on **which** ATM implementation is created
 - ▶ Code should focus only on portion of implementation relevant to ATM creation and ATM state management
 - ▶ Deliverables:
 - ▶ Sketch of V6 ATM implementation

DESIGN ACTIVITY: STEP 2: DISCUSSION