



SWE 621
SPRING 2025

FOLLOWING A DESIGN

LOGISTICS

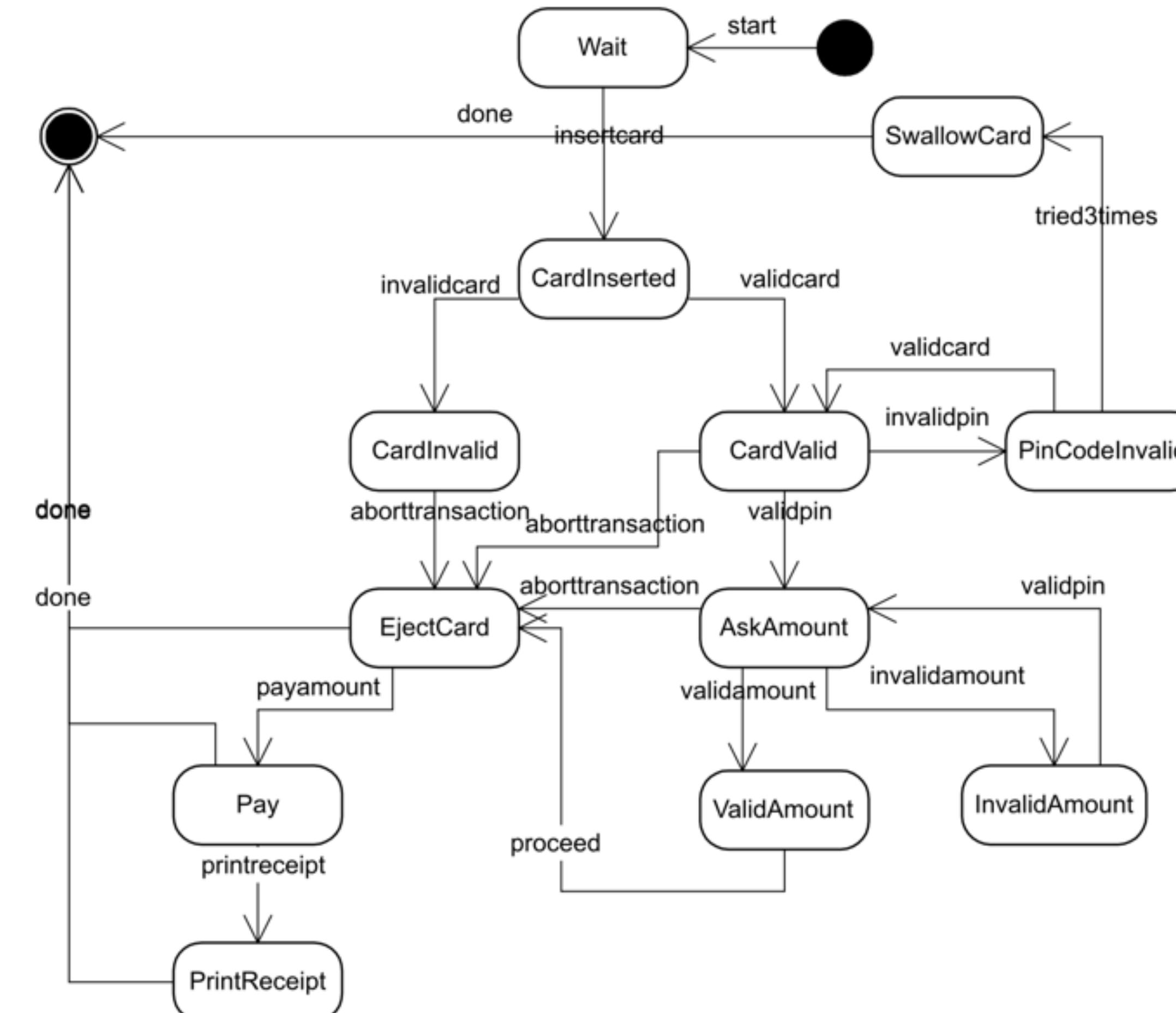
- ▶ HW4 due next week

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.

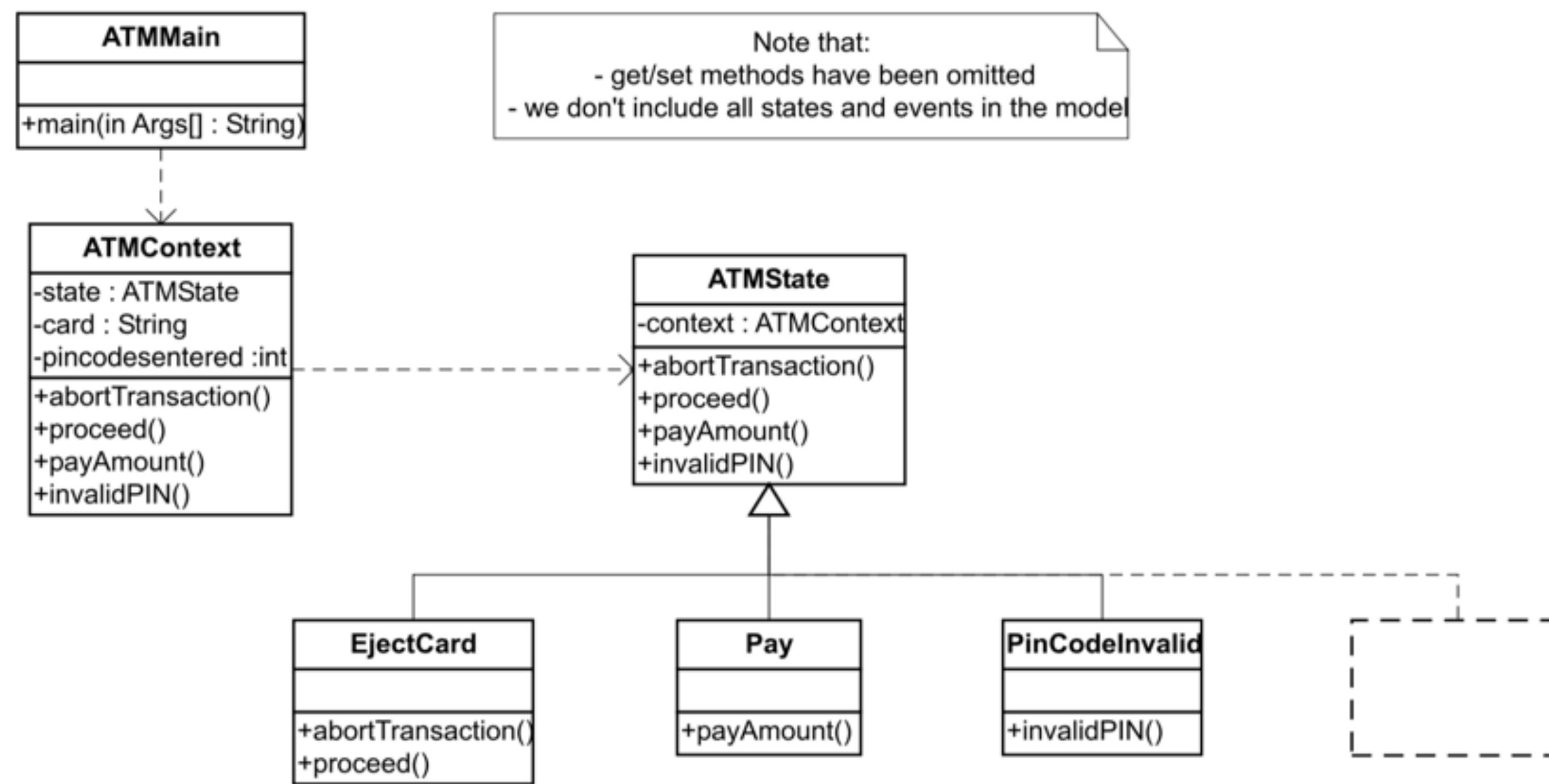
EXAMPLE: HOW SOFTWARE EVOLVES OVER TIME

- ▶ 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

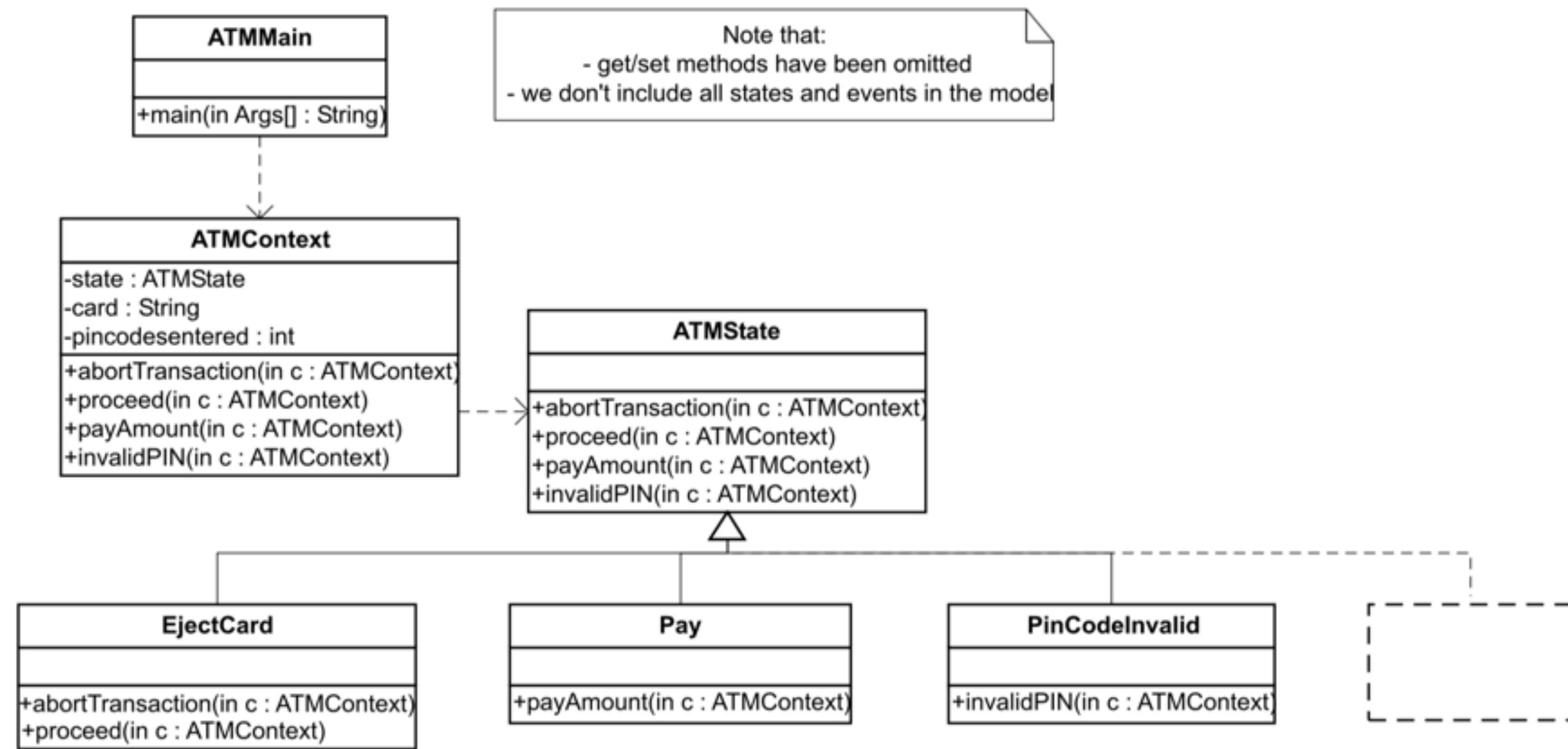


V1: STATE PATTERN

- ▶ 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

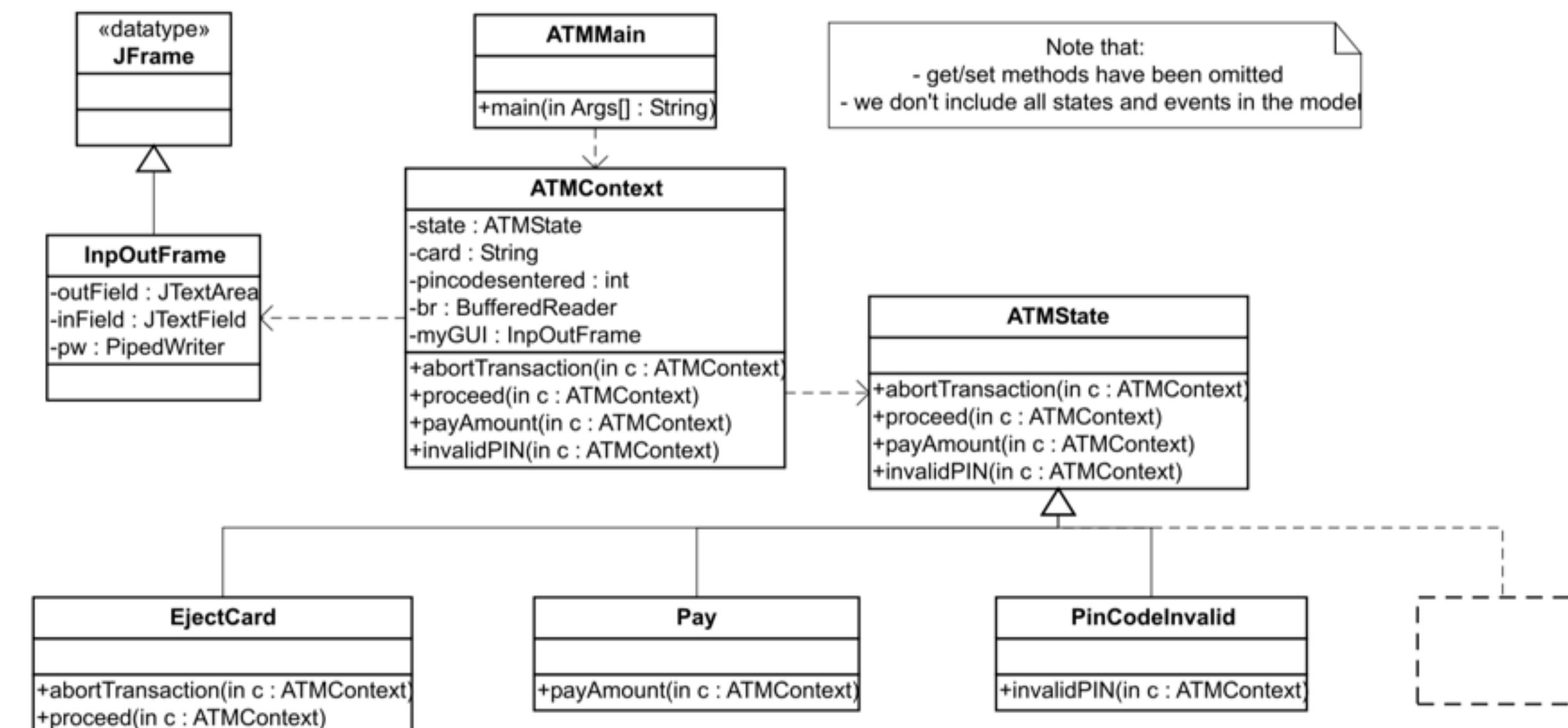
- ▶ Each state class is only created once
- ▶ Removed the context property from ATMState, added context parameter in each event method

V3: MULTIPLE INSTANCES

- ▶ Goals

- ▶ Parallelism:
enable each
simulator to
run in a
separate
thread

- ▶ UI: support
multiple
simulators



V3: MULTIPLE INSTANCES

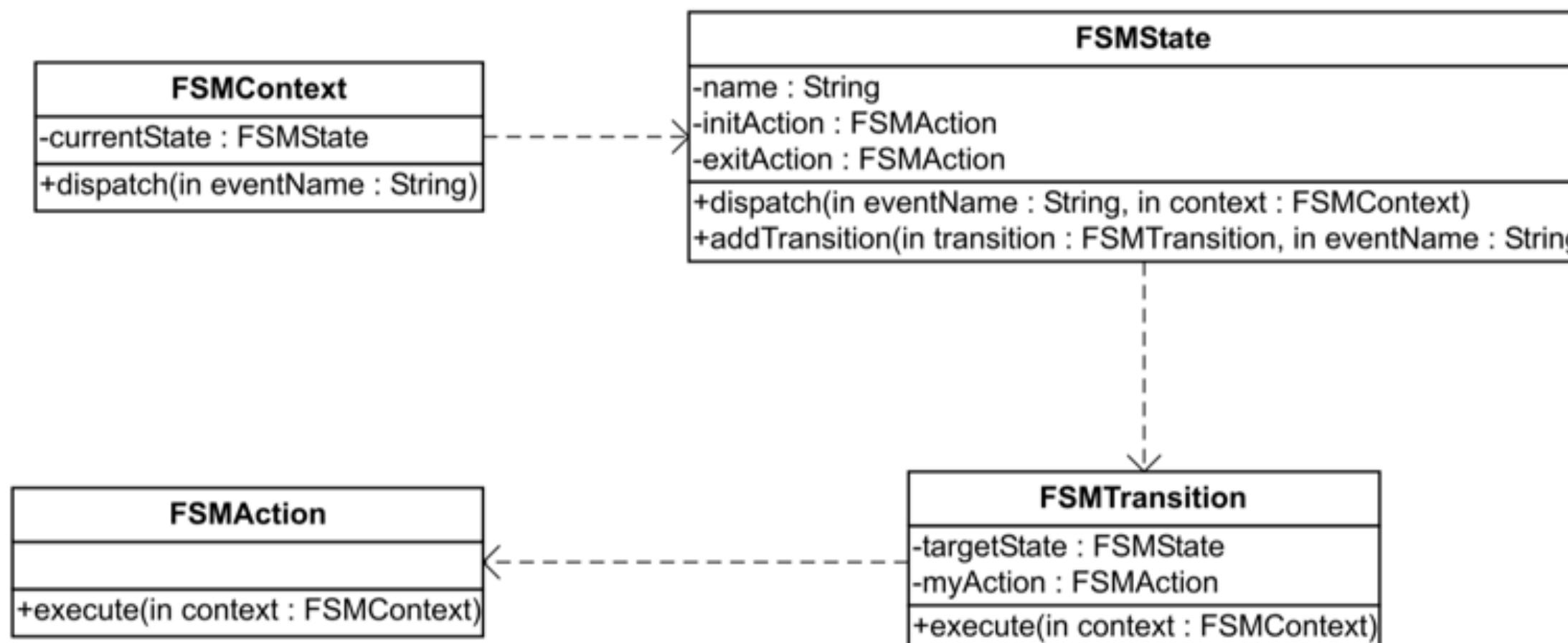
- ▶ Replaced command line with GUI, each containing multiple windows
- ▶ Each window associated with ATMContext
- ▶ GUI connected to ATMContext with pipes and filters
 - ▶ Whenever a user enters data, can read from IOStream from GUI just as if it were the command line

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
}
```

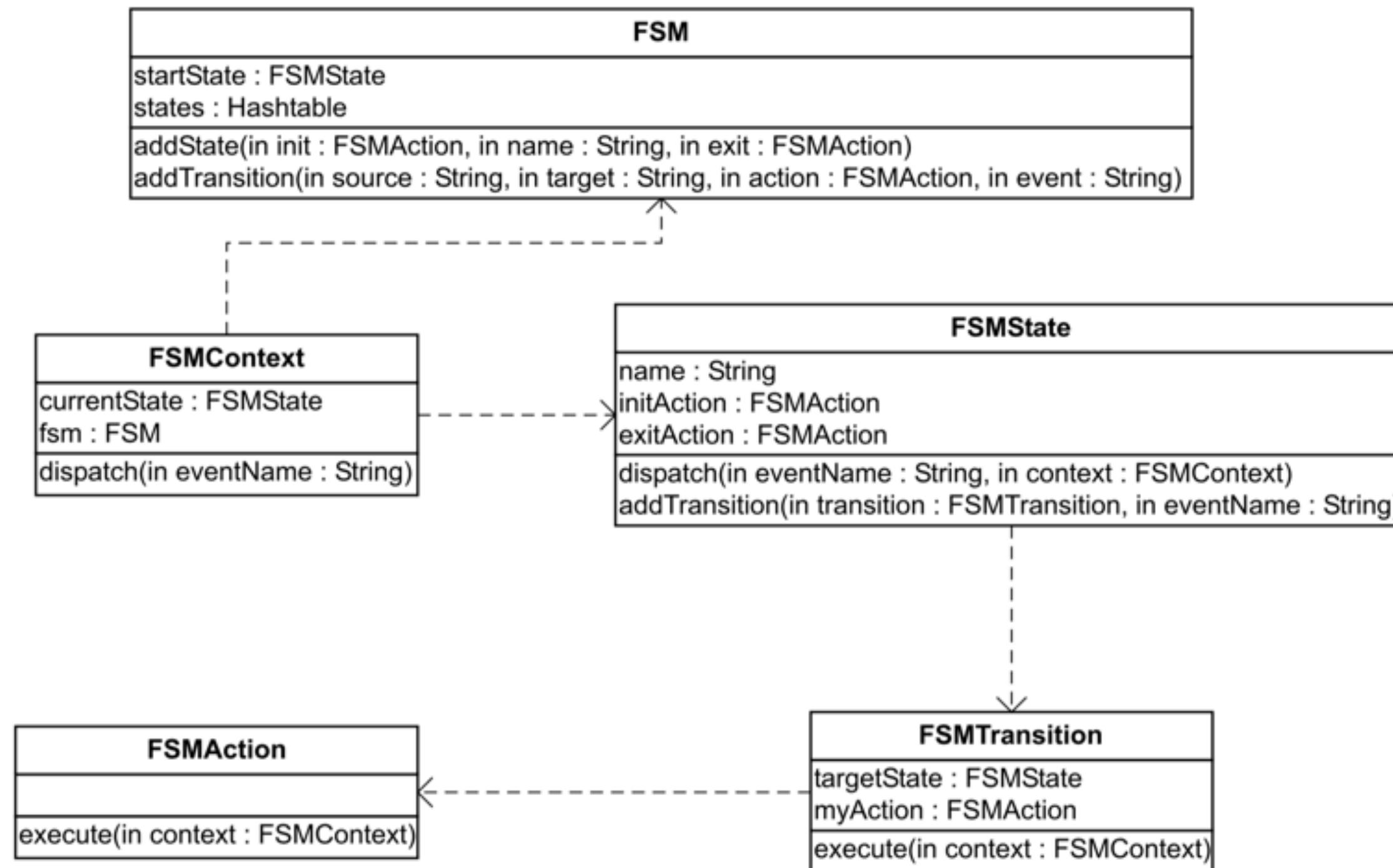


V4: DELEGATION BASED APPROACH

- ▶ Use 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 separates responsibility of storing FSM from dispatching



SUMMARY OF EVOLUTION

► Later decisions revised earlier

Version	Decision	Effect on system
v1	1.1 1.2 1.3 1.4	Use the State pattern Put data in context class Make context a property of ATMState 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 2.2	Make instances of State static 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 3.2 3.3	Create a GUI Replace System.in and System.out calls with calls to the GUI 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 The changes needed in the event methods are relatively small
v4	4.1 4.2 4.3	Refactor the system to use delegation (Van Gurp and Bosch, 1999). Use the command pattern to separate behaviour from structure 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

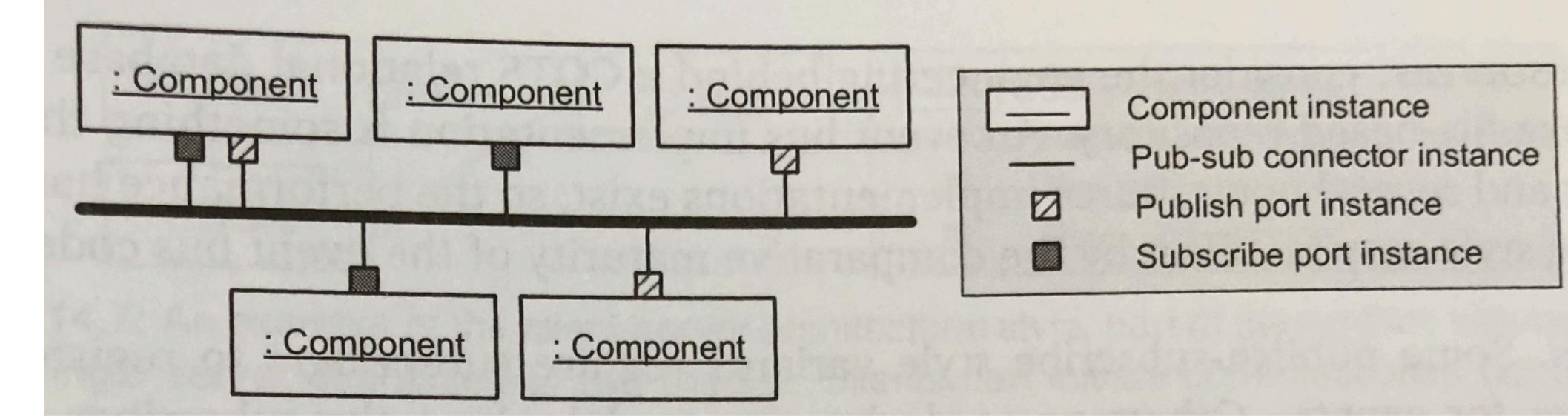
- ▶ Software architectural erosion (or decay): the gap between the architecture **as designed** as an **as built**
 - ▶ e.g., intended to be a pipes and filters architecture, but isn't entirely
- ▶ Consequences of design decision are no longer achieved
 - ▶ if decision helped enable maintainability,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

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

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

client/src/lua/mod.rs

97	- });
84	+ pub(crate) struct OutputHandler;
85	+
86	+ impl OutputHandler {

Timidger on May 6, 2019 Member

Why is this output handling code in `lua/mod.rs` to begin with? Shouldn't it be in `wayland_obj/output.rs`?

client/src/lua/mod.rs

112	+ } else {
113	+ // TODO We may not always want to add a new screen
114	+ // see how awesome does it and fix this.
115	+ trace!("Allocating screen for new output");

Timidger on May 6, 2019 Member

Either remove or give more information (e.g. resolution, positioning, etc.). The more information the easier it is to debug potential problems later.

(Side note: I'm ok with adding debug information that is not present in AwesomeWM. It should just make sense of course to add it, as we don't want to fill up user's harddrives with needless debug prints)

MAKING DESIGN CONSISTENT WITH CODE

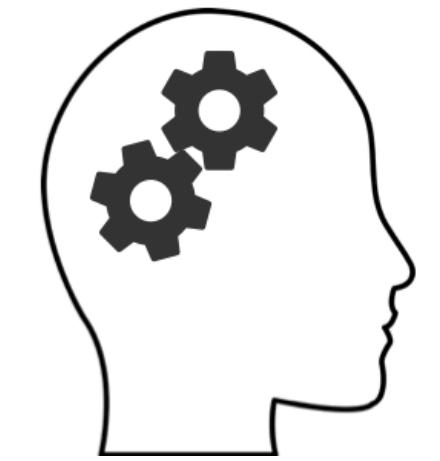
Working with software docs

- ICSE 2006, PLATEAU 2010
- Developers are encouraged to use **documentation** to store and learn about design.
 - Often non-existent, outdated, and **untrustworthy**.
- Developers may consult **senior team members**.
 - Left teams, forgotten the specifics, or too occupied.
- Developers have to manually **reverse engineer** code.
 - Tedious, time-consuming, error-prone.
 - Differentiating accidental patterns from intentional ones.
 - Understanding the rationale one of the hardest problems developers face.

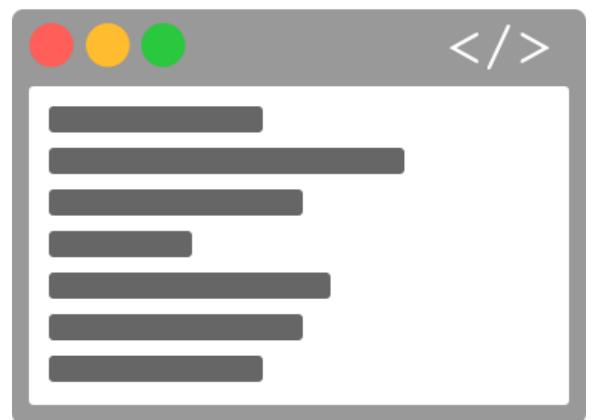
Documentation



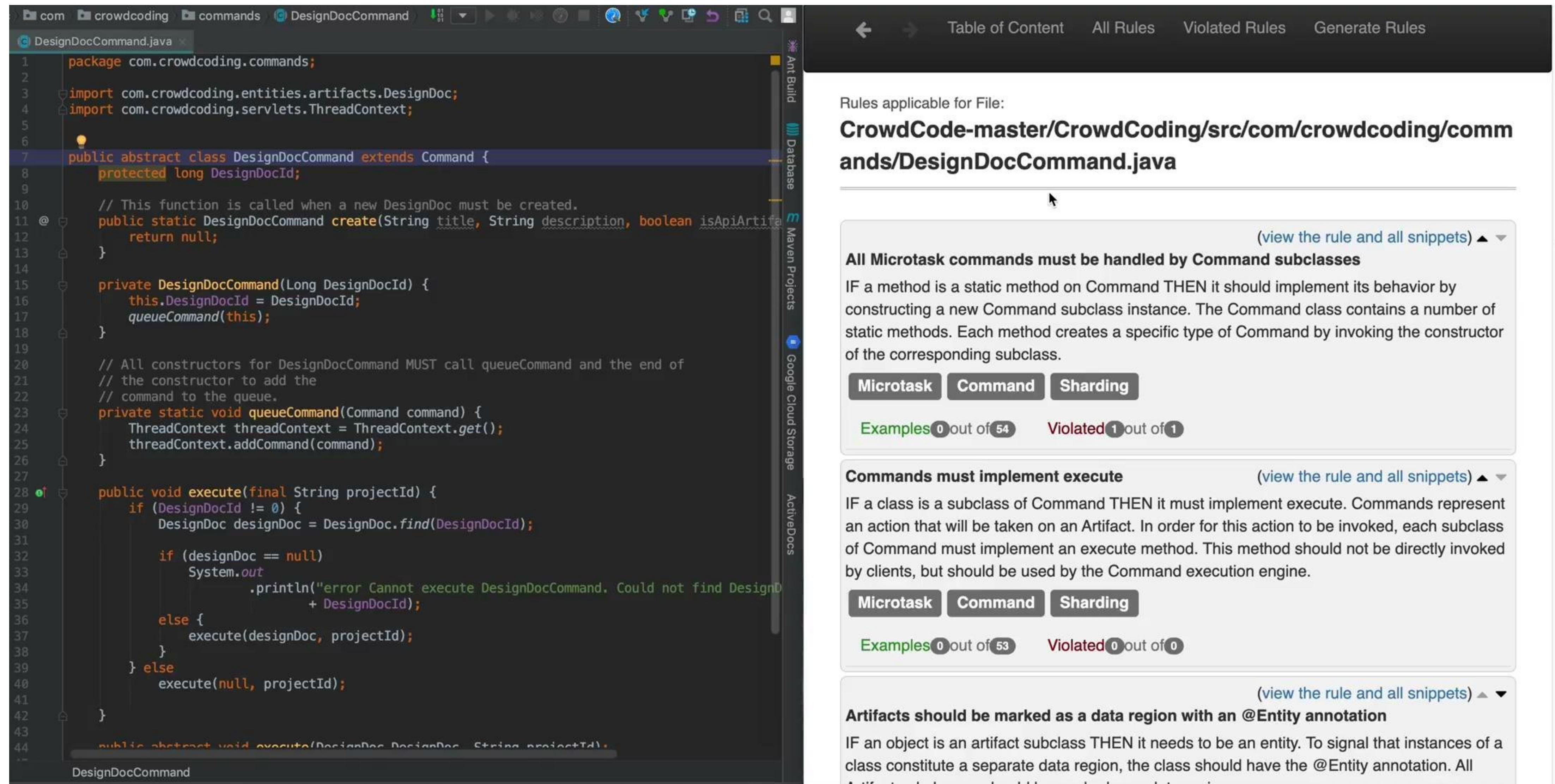
Software Design



Software Code



Active Documentation



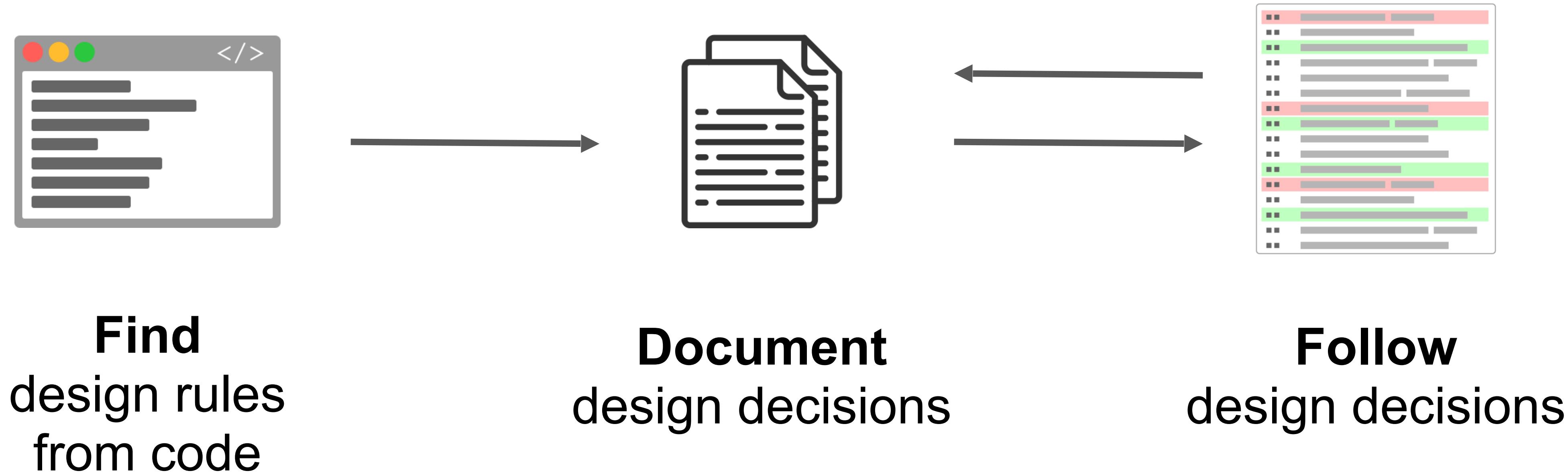
The screenshot shows an IDE interface with a Java code editor on the left and an active documentation panel on the right. The code editor displays `DesignDocCommand.java` with the following content:

```
1 package com.crowdcoding.commands;
2
3 import com.crowdcoding.entities.artifacts.DesignDoc;
4 import com.crowdcoding.servlets.ThreadContext;
5
6 public abstract class DesignDocCommand extends Command {
7     protected long DesignDocId;
8
9     // This function is called when a new DesignDoc must be created.
10    public static DesignDocCommand create(String title, String description, boolean isApiArtifact) {
11        return null;
12    }
13
14    private DesignDocCommand(Long DesignDocId) {
15        this.DesignDocId = DesignDocId;
16        queueCommand(this);
17    }
18
19    // All constructors for DesignDocCommand MUST call queueCommand and the end of
20    // the constructor to add the
21    // command to the queue.
22    private static void queueCommand(Command command) {
23        ThreadContext threadContext = ThreadContext.get();
24        threadContext.addCommand(command);
25    }
26
27    public void execute(final String projectId) {
28        if (DesignDocId != 0) {
29            DesignDoc designDoc = DesignDoc.find(DesignDocId);
30
31            if (designDoc == null)
32                System.out
33                    .println("error Cannot execute DesignDocCommand. Could not find DesignDoc"
34                           + DesignDocId);
35            else {
36                execute(designDoc, projectId);
37            }
38        } else
39            execute(null, projectId);
40
41    }
42
43    public abstract void execute(DesignDoc designDoc, String projectId);
44}
```

The active documentation panel on the right shows rules applicable for the file `DesignDocCommand.java`:

- All Microtask commands must be handled by Command subclasses**
IF a method is a static method on Command THEN it should implement its behavior by constructing a new Command subclass instance. The Command class contains a number of static methods. Each method creates a specific type of Command by invoking the constructor of the corresponding subclass.
[\(view the rule and all snippets\)](#)
Tags: **Microtask** **Command** **Sharding**
Examples: 0 out of 54 | Violated: 1 out of 1
- Commands must implement execute**
IF a class is a subclass of Command THEN it must implement execute. Commands represent an action that will be taken on an Artifact. In order for this action to be invoked, each subclass of Command must implement an execute method. This method should not be directly invoked by clients, but should be used by the Command execution engine.
[\(view the rule and all snippets\)](#)
Tags: **Microtask** **Command** **Sharding**
Examples: 0 out of 53 | Violated: 0 out of 0
- Artifacts should be marked as a data region with an @Entity annotation**
IF an object is an artifact subclass THEN it needs to be an entity. To signal that instances of a class constitute a separate data region, the class should have the @Entity annotation. All
[\(view the rule and all snippets\)](#)

Our Solution - Active Documentation



[Mehrpor et al. Submitted to UIST 2024]

[Mehrpor et al. ESEC/FSE 2020]

[Mehrpor et al. VL/HCC 2019]

Actively Following Design Decisions in Code

Documentation should be "**active**".

- Design decisions are **actively checked** against code.
- An **active** link between the documentation and code is generated.
- Developers can **actively update** the documentation.



ActiveDocumentation

The screenshot shows an IDE interface with a code editor on the left and an 'Active Documentation' panel on the right.

Code Editor (Left):

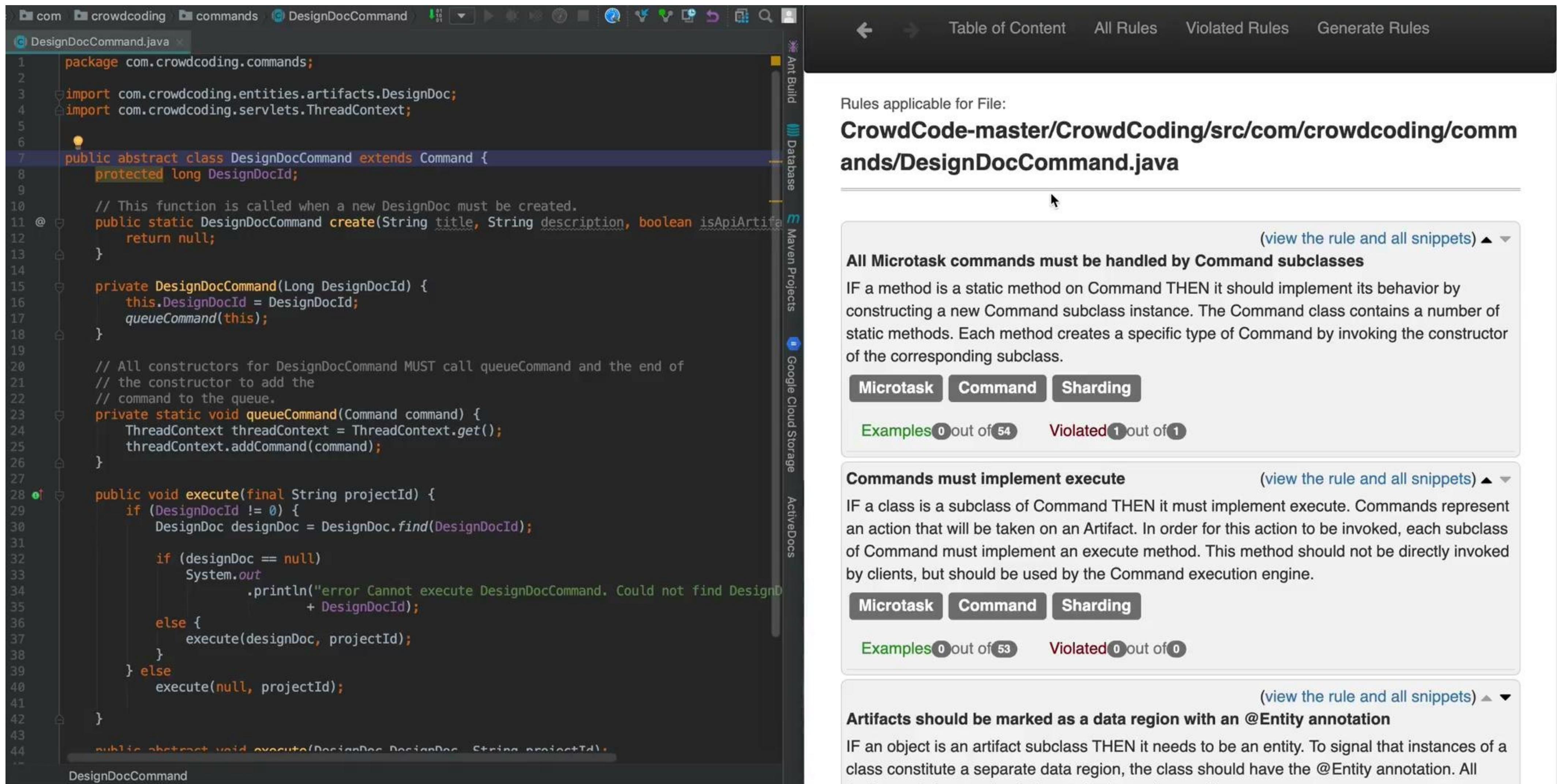
```
1 package com.crowdcoding.commands;
2
3 import com.crowdcoding.entities.artifacts.DesignDoc;
4 import com.crowdcoding.servlets.ThreadContext;
5
6
7 public abstract class DesignDocCommand extends Command {
8     protected long DesignDocId;
9
10    // This function is called when a new DesignDoc must be created.
11    public static DesignDocCommand create(String title, String description, boolean isApiArtifact) {
12        return null;
13    }
14
15    private DesignDocCommand(Long DesignDocId) {
16        this.DesignDocId = DesignDocId;
17        queueCommand(this);
18    }
19
20    // All constructors for DesignDocCommand MUST call queueCommand and the end of
21    // the constructor to add the
22    // command to the queue.
23    private static void queueCommand(Command command) {
24        ThreadContext threadContext = ThreadContext.get();
25        threadContext.addCommand(command);
26    }
27
28    public void execute(final String projectId) {
29        if (DesignDocId != 0) {
30            DesignDoc designDoc = DesignDoc.find(DesignDocId);
31
32            if (designDoc == null)
33                System.out
34                    .println("error Cannot execute DesignDocCommand. Could not find DesignDoc with id: " + DesignDocId);
35            else {
36                execute(designDoc, projectId);
37            }
38        } else
39            execute(null, projectId);
40
41    }
42
43    // todo: implement void execute(DesignDoc designDoc, String projectId);
44
45}
```

Active Documentation Panel (Right):

- Table of Content:** All Rules, Violated Rules, Generate Rules.
- Active Documentation:** A heading.
- Tags:** A grid of letters A through U, with 'All' highlighted.
- Rules:** A list of 8 rules, each with a number and a description.

1	All Microtask commands must be handled by Command subclasses
2	Commands must implement execute
3	Artifacts should be marked as a data region with an @Entity annotation
4	Microtasks must have a reference to the Artifact that it belongs to
6	Communication between artifacts should be indirection through a Command
8	Objects to be sent to an external service should have a corresponding DTO that needs to be transferred has a DTO, i.e. ObjectMapper should not be in Entity.

ActiveDocumentation: Active Link



The screenshot shows an IDE interface with a Java file named `DesignDocCommand.java` open. The code implements the Command design pattern, with methods for creating design documents and executing them. The right side of the interface displays active documentation for the file, listing rules and their violations.

Rules applicable for File:
CrowdCode-master/CrowdCoding/src/com/crowdcoding/commands/DesignDocCommand.java

All Microtask commands must be handled by Command subclasses (view the rule and all snippets) ▲▼

IF a method is a static method on Command THEN it should implement its behavior by constructing a new Command subclass instance. The Command class contains a number of static methods. Each method creates a specific type of Command by invoking the constructor of the corresponding subclass.

Microtask **Command** **Sharding**

Examples 0 out of 54 Violated 1 out of 1

Commands must implement execute (view the rule and all snippets) ▲▼

IF a class is a subclass of Command THEN it must implement execute. Commands represent an action that will be taken on an Artifact. In order for this action to be invoked, each subclass of Command must implement an execute method. This method should not be directly invoked by clients, but should be used by the Command execution engine.

Microtask **Command** **Sharding**

Examples 0 out of 53 Violated 0 out of 0

Artifacts should be marked as a data region with an @Entity annotation (view the rule and all snippets) ▲▼

IF an object is an artifact subclass THEN it needs to be an entity. To signal that instances of a class constitute a separate data region, the class should have the @Entity annotation. All

DesignDocCommand

ActiveDocumentation: Active Check

The screenshot illustrates the ActiveDocumentation interface, which integrates a code editor and a rule checker. On the left, a code editor displays Java code for `ADTCommand.java` and `DesignDocCommand.java`. The code includes methods for command execution and queueing. On the right, a rule checker interface lists several rules:

- All Microtask commands must be handled by Command subclasses**: (view the rule and all snippets) ▲ ▼
IF a method is a static method on Command THEN it should implement its behavior by constructing a new Command subclass instance. The Command class contains a number of static methods. Each method creates a specific type of Command by invoking the constructor of the corresponding subclass.
Microtask Command Sharding
Examples 55 Violated 0
- Commands must implement execute**: (view the rule and all snippets) ▲ ▼
IF a class is a subclass of Command THEN it must implement execute. Commands represent an action that will be taken on an Artifact. In order for this action to be invoked, each subclass of Command must implement an execute method. This method should not be directly invoked by clients, but should be used by the Command execution engine.
Microtask Command Sharding
Examples 53 Violated 1
- Artifacts should be marked as a data region with an @Entity annotation**: (view the rule and all snippets) ▲ ▼
IF an object is an artifact subclass THEN it needs to be an entity. To signal that instances of a class constitute a separate data region, the class should have the @Entity annotation. All Artifact subclasses should be marked as a data region.
Entity Sharding Persistence
Examples 4 Violated 0
- Microtasks must have a reference to the Artifact that it belongs to**: (view the rule and all snippets) ▲ ▼
IF a class is a subclass of Microtask THEN it needs a field representing the reference to the associated entity. Each Microtask represents work to be done on an Artifact. As such, it needs to be connected back to its owning artifact through a reference to the Artifact. Without the reference, they need to have an ID of the artifact and for submitting they need to load the data beforehand.
Entity Microtask Objectify Persistence
Examples 4 Violated 0
- Communication between artifacts should be indirection through a Command**: (view the rule and all snippets) ▲ ▼
IF an Artifact needs to communicate with another artifact THEN it should create a Command describing the

User Study

- Conducted a user study with 18 participants.
Goal: Add new code to an existing codebase while following design decisions.
- **Task:** Add a new feature in Microtask prgramming codebase given ActiveDocumentation and traditional documentation.
- **Results:**
 - ActiveDocumentation helped participants work with design decisions:
 - **Quickly – 3 times faster** in starting editing the code ($U = 12.5, p < 0.05$), **28% faster** in finishing the task ($U = 16.5, p < 0.05$)
 - **Successfully – 98% fewer** incorrect lines of code ($p < 0.044$)
 - Used **example snippets** to learn how to follow decisions.
 - Used links to **violated snippets** to locate parts of code that need change.
 - Used **instant feedback** to verify changes.

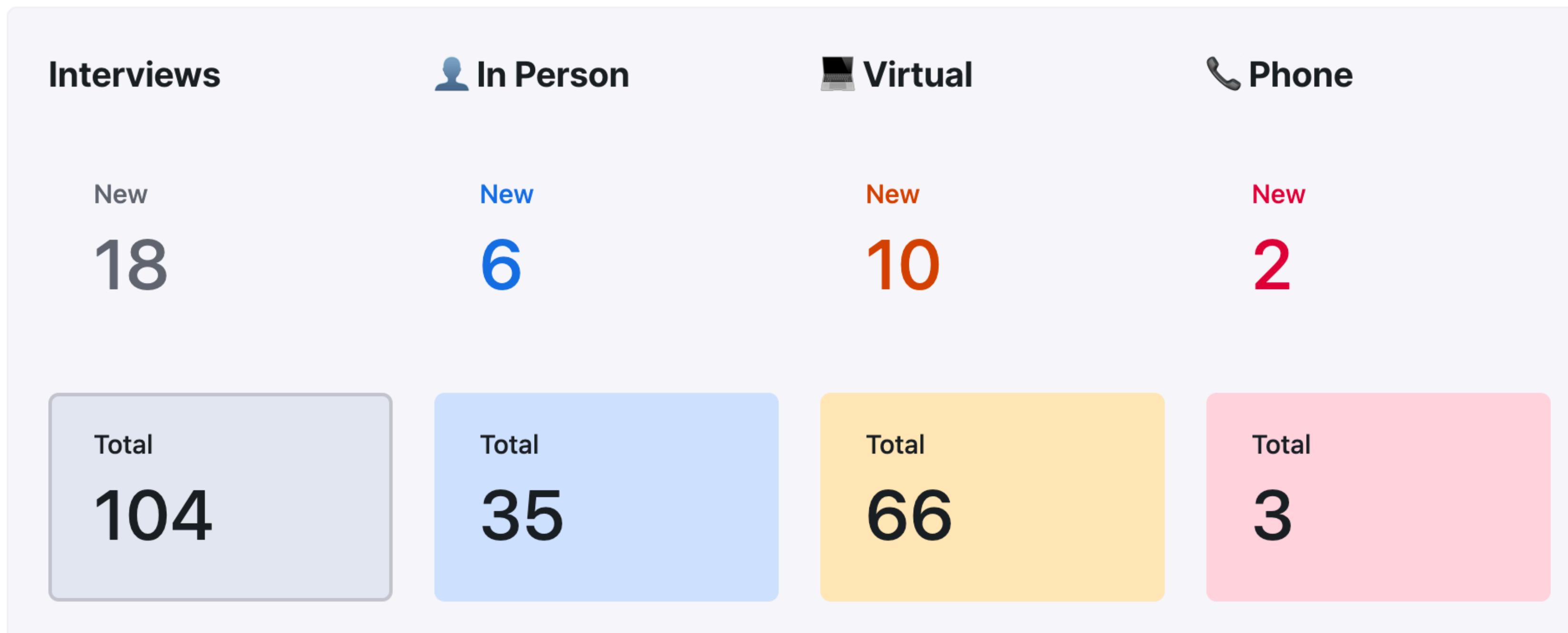
Active Documentation

We help software developers to change code other developers wrote in less time.



Team 3577

Engineering managers at small- or medium-sized companies working on software products that require long-term maintenance **will use** our documentation-building process **to** reduce knowledge transfer time and cost.



NSF Lineage:

NSF 1845508, NSF 1703734

We developed a new static-analysis based technique to **keep design docs in sync with code** and use data mining techniques to infer design from code.

Team



- **Consuelo Lopez, EL**

+16 years experience in different roles in software industry.



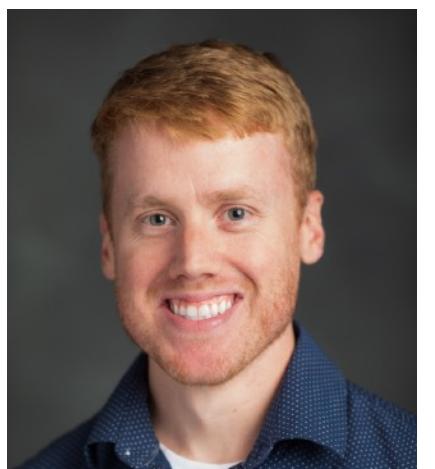
- **Thomas LaToza, Co-EL**

Professor and an expert in designing new types of developer tools.



- **Sahar Mehrpour, TL**

Researcher specializing in documentation tools with 7+ years of experience.



- **Austin Henley, IM**

Former CTO of an acquired startup and VP of Engineering at a Series C company.

Tool Adoption Process

- Big companies
 - Big companies have their own internal tools org, and build tools in-house
 - Or have complex policies and paperwork for adoption
 - Buy developer tools from other big companies.
- Most adoption is more bottom up
 - Team leads or senior engineers learn about tools through social media
 - Engineers try out tool (for free) on their own hobby / OSS projects
 - Engineers suggest tool to team lead or other manager or decision maker for feedback
 - [Sometimes] Gather alternative tools for comparison
 - [Sometimes] Trial tool internally or compare against alternative tools
 - Build business case: cost of tool vs. value of saving
 - Decision maker makes decision

When do teams pay for tools?



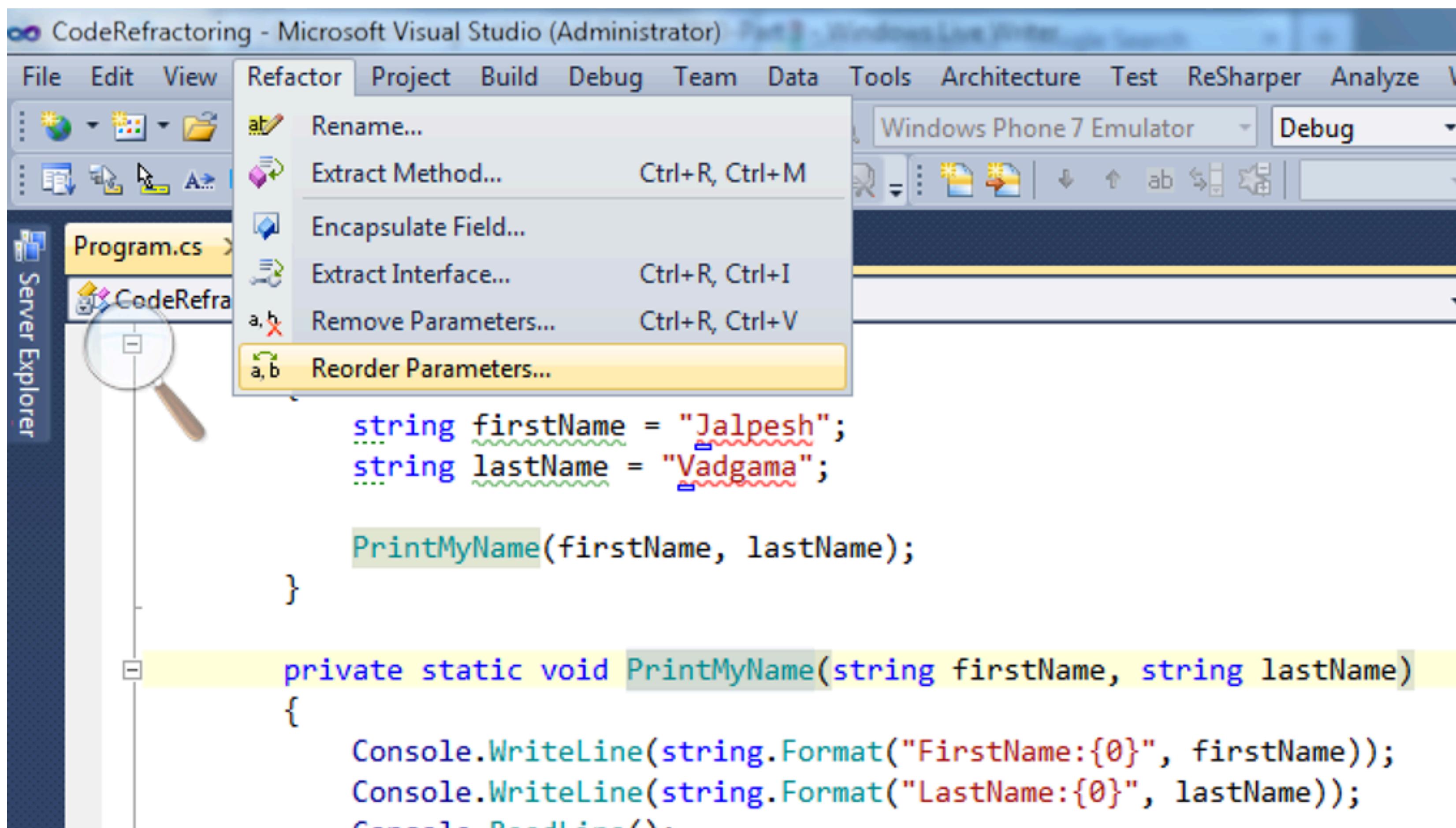
“Companies are willing to pay for tools that makes developers do less work.”

- Hard to sell a developer tool as a new startup - have to build trust & reputation
- Teams are spending more than \$10 - 20 on developer tools, mostly from big companies
- Scared of being on the bleeding edge and adopting something that will disappear or not yet validated

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 the implementation
- ▶ Code that is inconsistent with the 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

IN-CLASS ACTIVITY: STEP 1

- ▶ Your team is growing & looking to hire 3 new junior engineers. You're in charge of helping onboard them.
- ▶ Based on the concepts about design & architecture you've learned in class so far, outline a plan.
 - ▶ What design documents will you create? What sections will these documents contain and how well they help to describe the design & architecture to new engineers?
 - ▶ What processes & practices will you put in place to help onboard engineers into your project?
 - ▶ What key performance indicators (KPIs) will you collect to evaluate what is working, or not working, in your onboarding efforts?
- ▶ Deliverables
 - ▶ Description of the design docs, processes, and KPIs you will use to onboard new engineers

IN-CLASS ACTIVITY: STEP 2

- ▶ Combine with another group
- ▶ Compare & contrast your approaches:
 - ▶ How are your doc structures similar or different?
 - ▶ How are your process & practices similar or different?
 - ▶ How are KPIs similar or different?
- ▶ Deliverables: Analysis of key differences & your recommendations on the best ways to proceed