SWE 621

SPRING 2025

FOLLOWING A DESIGN

LOGISTICS

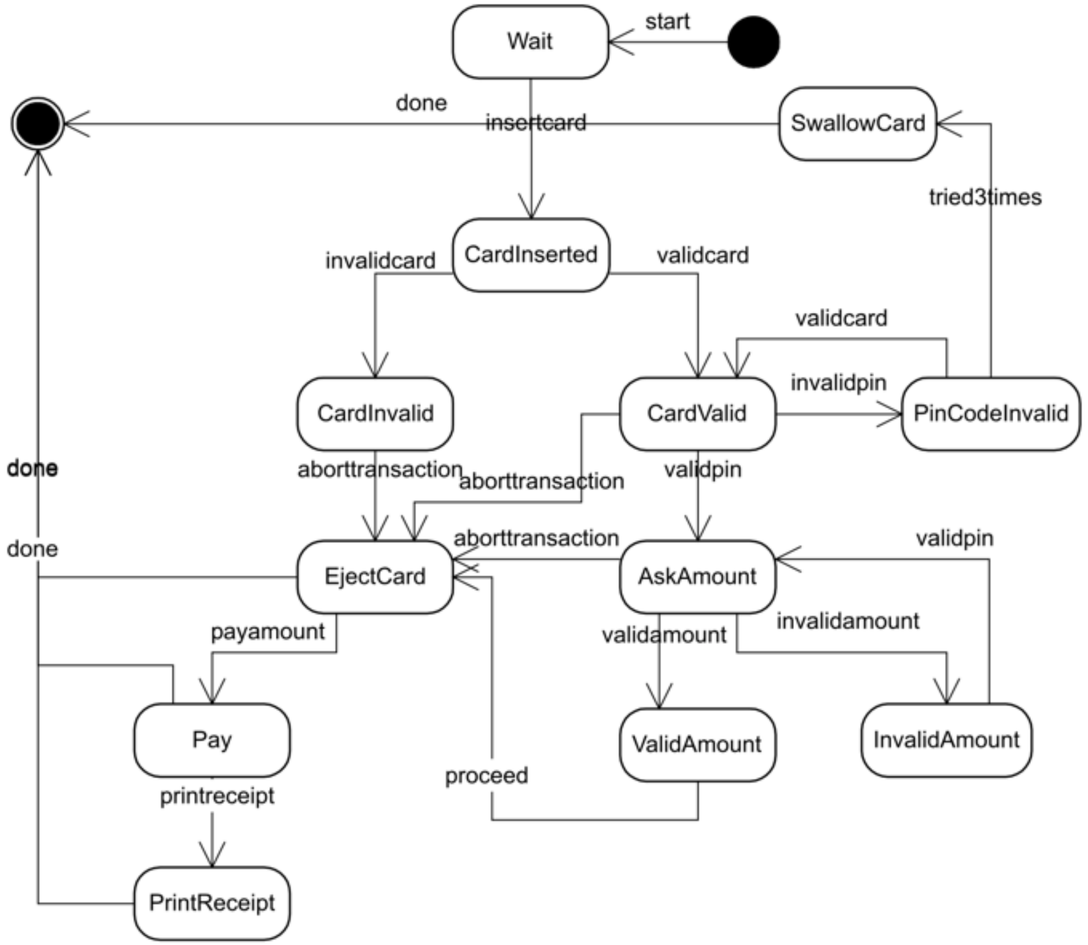
- HW4 due next week

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.

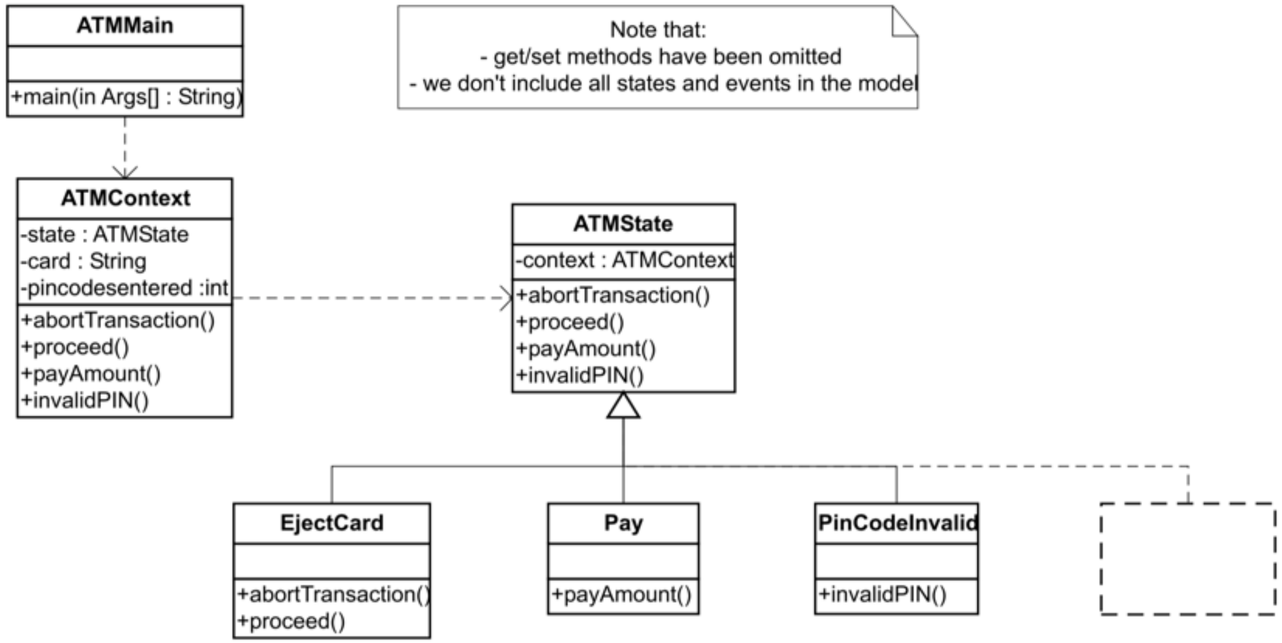
EXAMPLE: HOW SOFTWARE EVOLVES OVER TIME

- 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

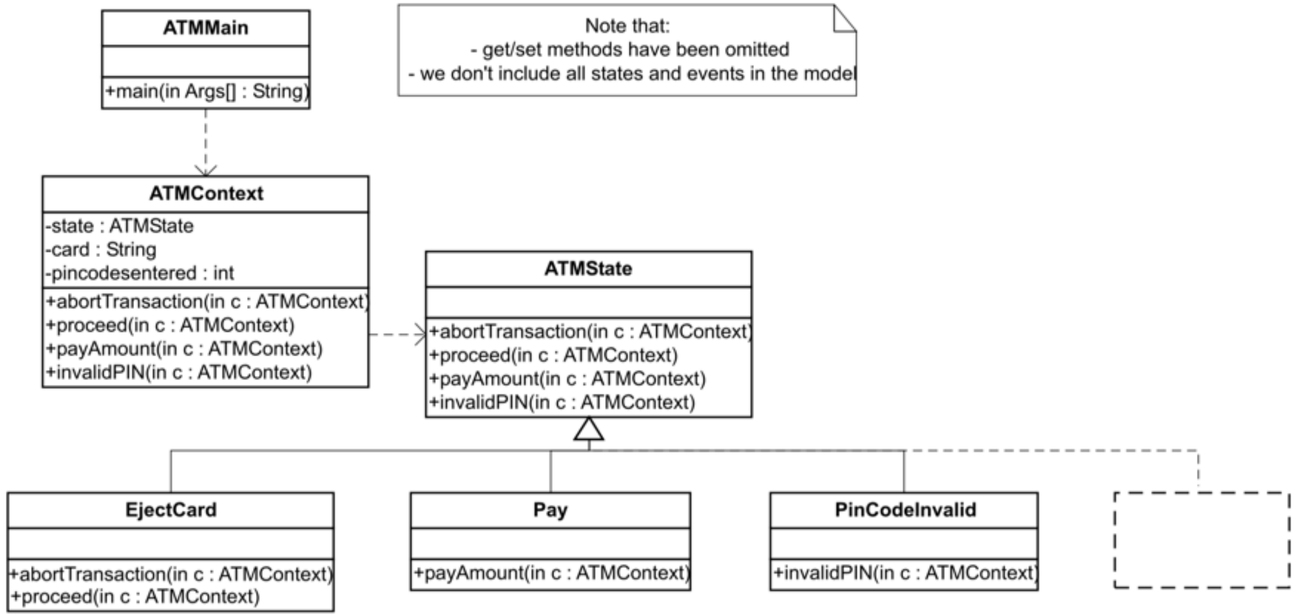


V1: STATE PATTERN

- 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

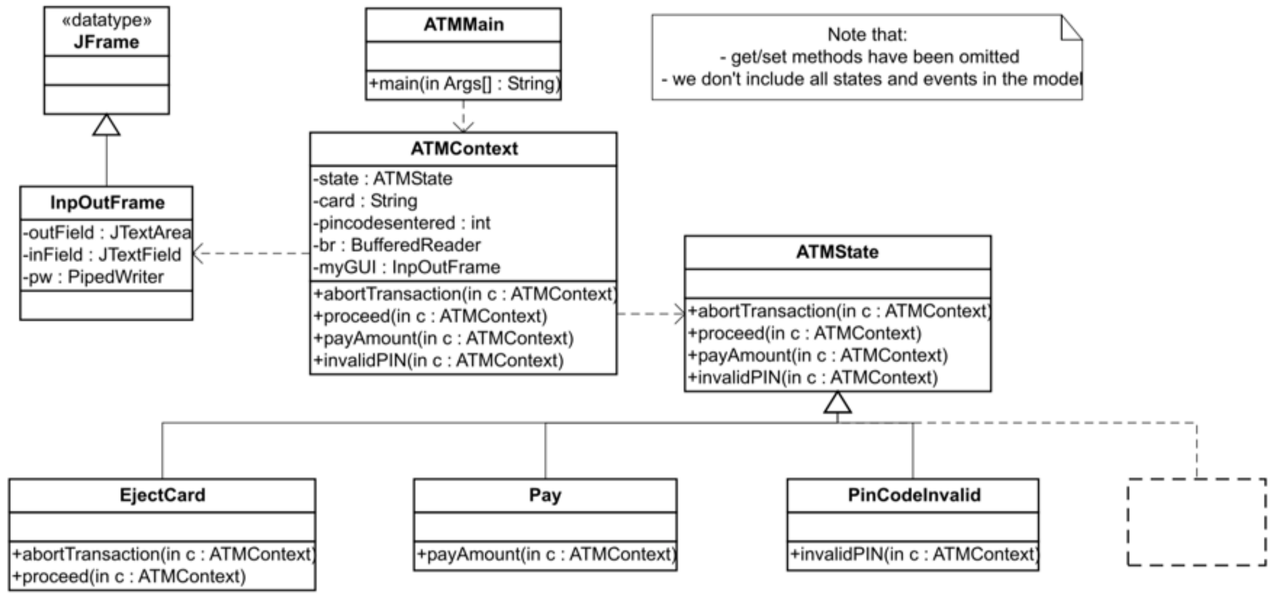


V2: FLYWEIGHT

- 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

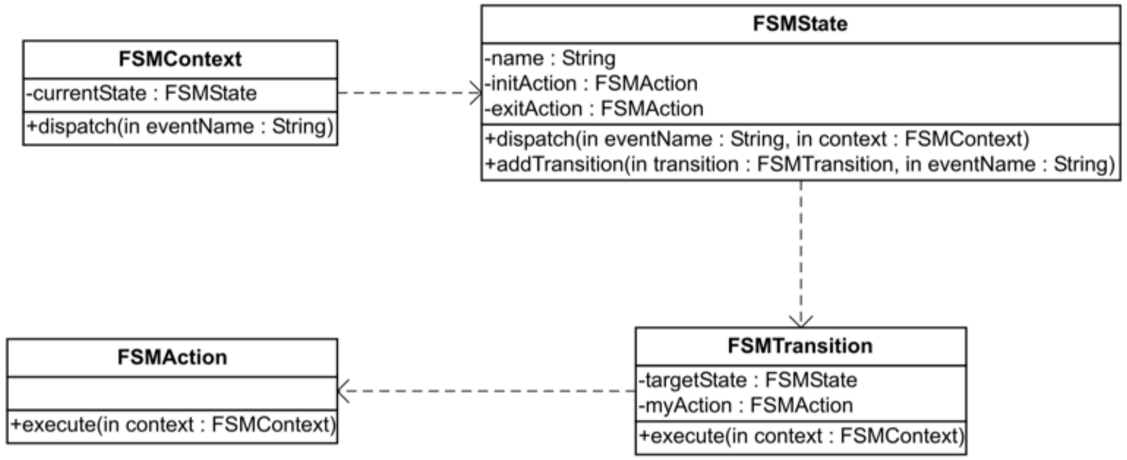
- Replaced command line with GUI, each containing multiple windows
- Each window associated with ATMContext
- GUI connected to ATMContext with pipes and filters
  - Whenever a user enters data, can read from IOStream from GUI just as if it were the command line

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 definitons
}
...//rest of the class
}
```

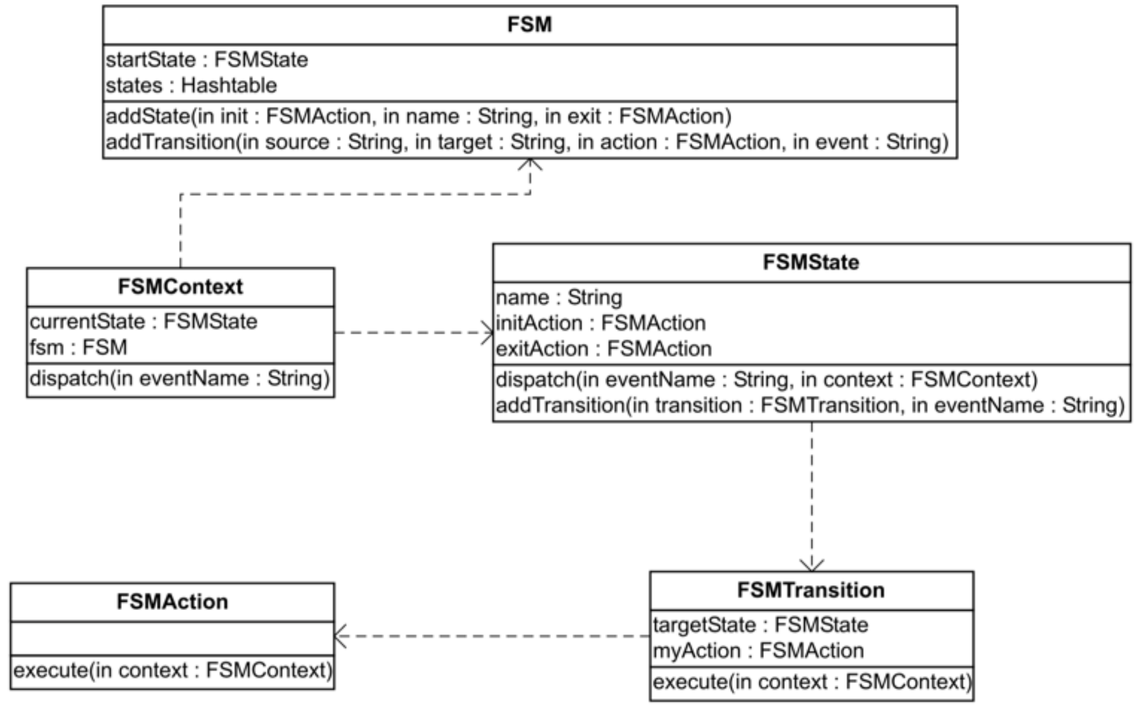


V4: DELEGATION BASED APPROACH

- 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 dispatchi

SUMMARY OF EVOLUTION

- Later decisions revised earlier

Version	Decision		Effect on system
v1	1.1	Use the State pattern	For each state in a FSM, a subclass of State has to be created
	1.2	Put data in context class	Each event method in the State subclasses refers to the Context class to access data
	1.3	Make context a property of ATMState	The context is available to all State instances
	1.4	Use command line for UI	The code is littered with calls to System.in and System.out
v2	2.1	Make instances of State static	The keyword static needs to be put before instantiations of State subclasses
	2.2	Remove context property from ATM-State and use parameter in event method instead	All event methods need to be edited
v3	3.1	Create a GUI	A class is added to the system
	3.2	Replace System.in and System.out calls with calls to the GUI	All event methods need to be revised
	3.3	Apply the pipes and filters for communication between GUI and simulator	The changes needed in the event methods are relatively small
v4	4.1	Refactor the system to use delegation (Van Gurp and Bosch, 1999).	New classes are created that model the behaviour of states and transitions. All existing State subclasses are removed from the system.
	4.2	Use the command pattern to separate behaviour from structure	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)
	4.3	Introduce state exit and entry events to the FSM model	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

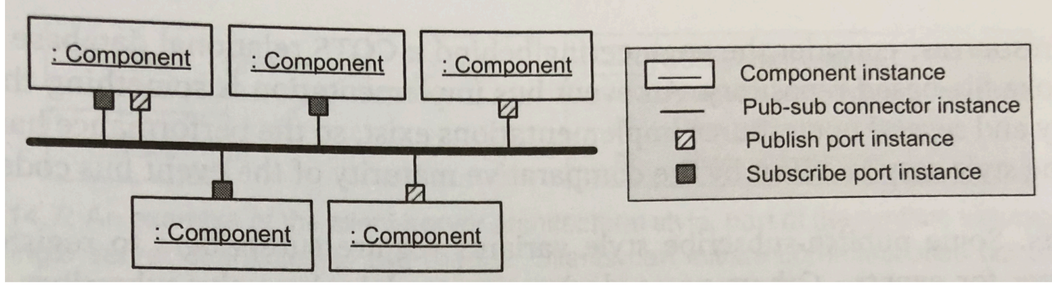
- Software architectural erosion (or decay): the gap between the architecture **as designed** as an **as built**
  - e.g., intended to be a pipes and filters architecture, but isn't entirely
- Consequences of design decision are no longer achieved
  - if decision helped enable maintainability, it does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

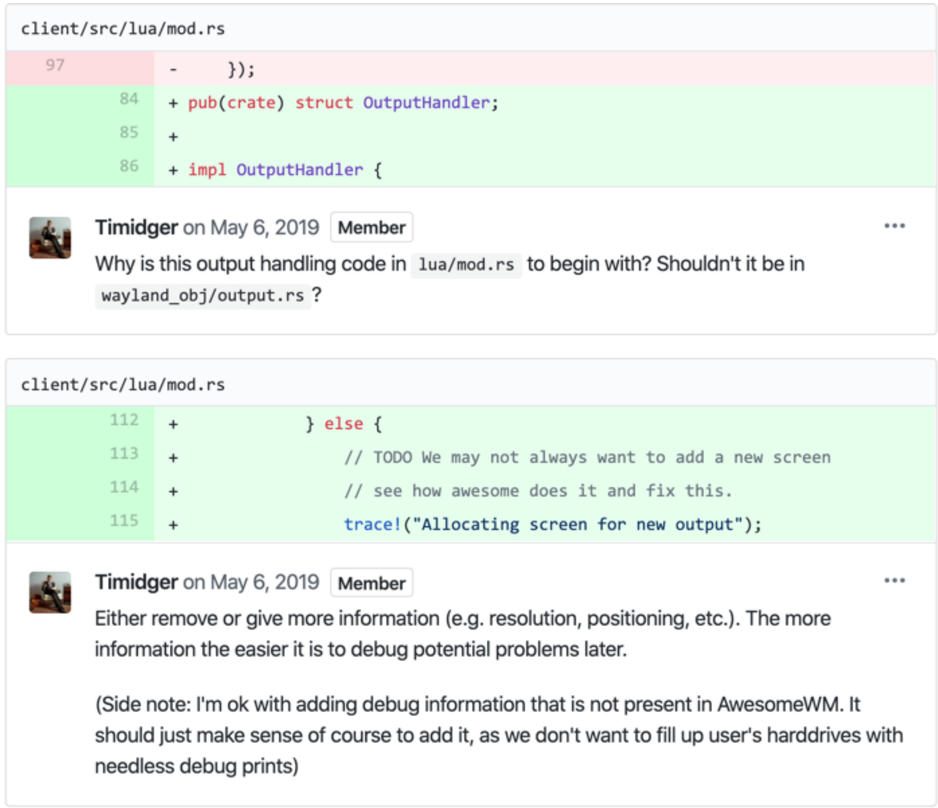
- Prevent code decay
  - Better communicate design to developers
  - Check that changes are consistent with design
- Fix code decay after it occurs
  - Refactor code to be consistent with design
  - Change code to be consistent with design changes

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



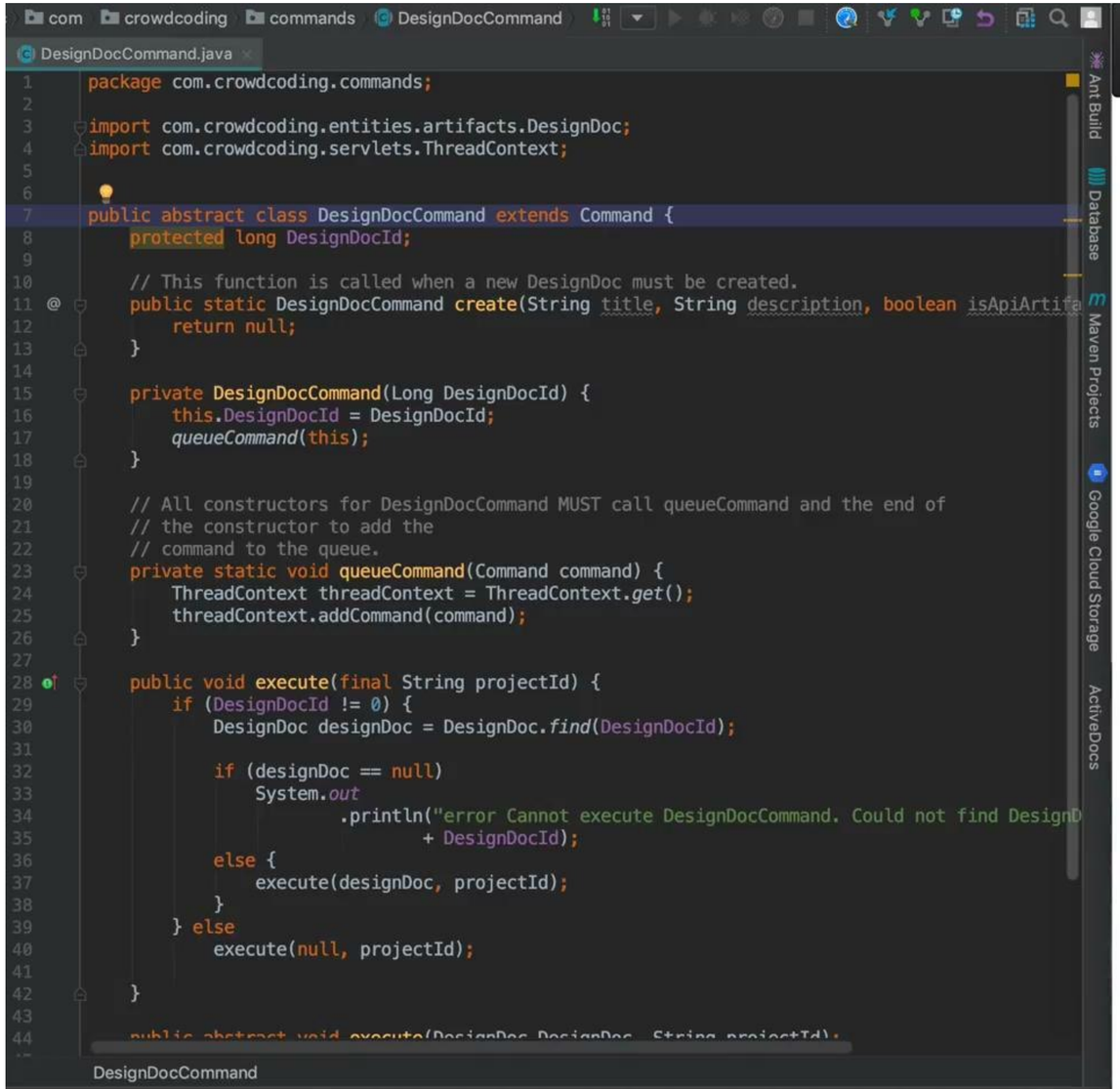
MAKING DESIGN CONSISTENT WITH CODE

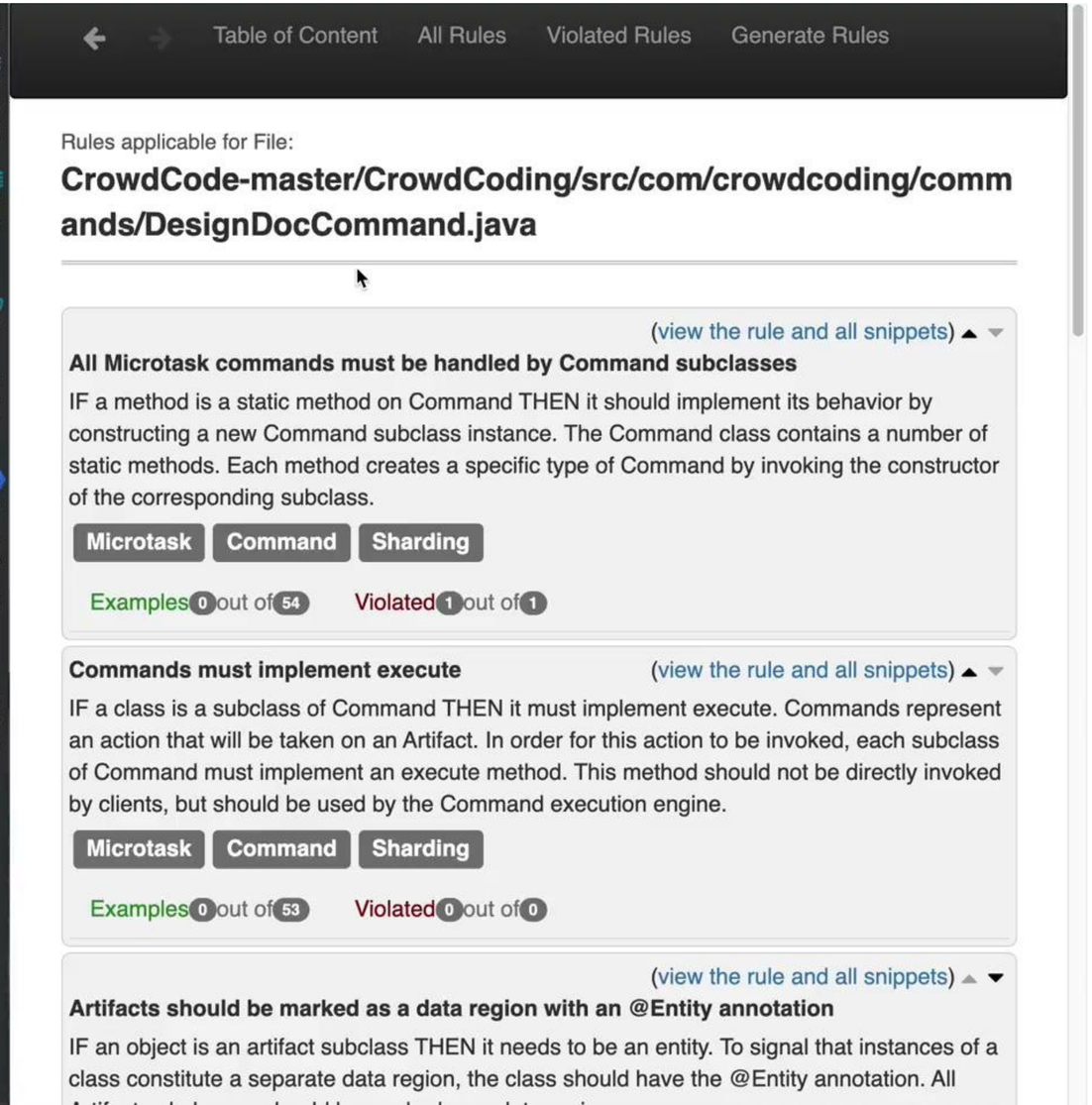
Working with software docs

- 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.

Active Documentation





Our Solution - Active Documentation

Find
design rules
from code

[Mehrpour et al. Submitted to UIST 2024]

Document
design decisions

[Mehrpour at al. ESEC/FSE 2020]

Follow
design decisions

[Mehrpour et al. VL/HCC 2019]

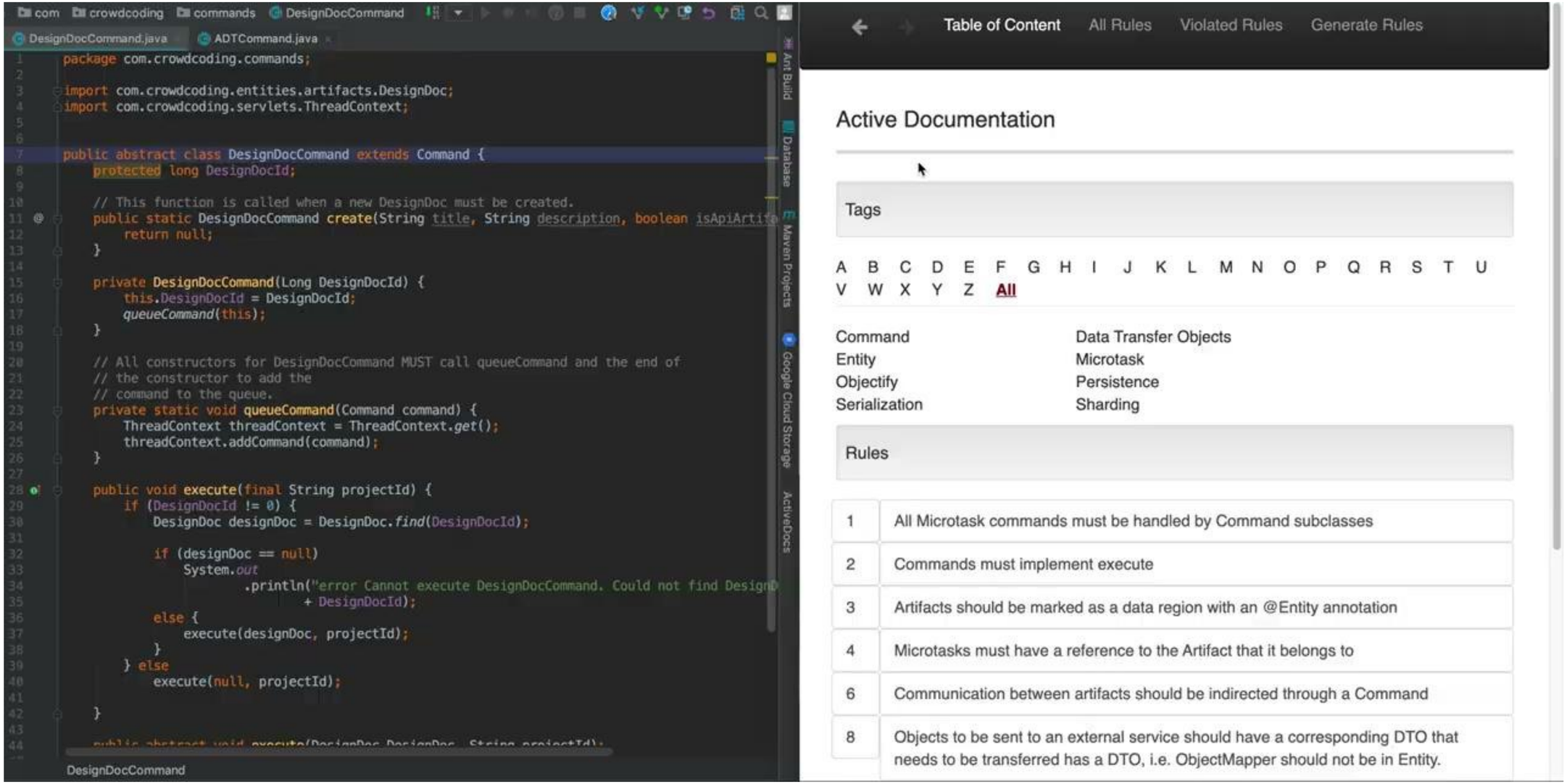
Actively Following Design Decisions in Code

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.

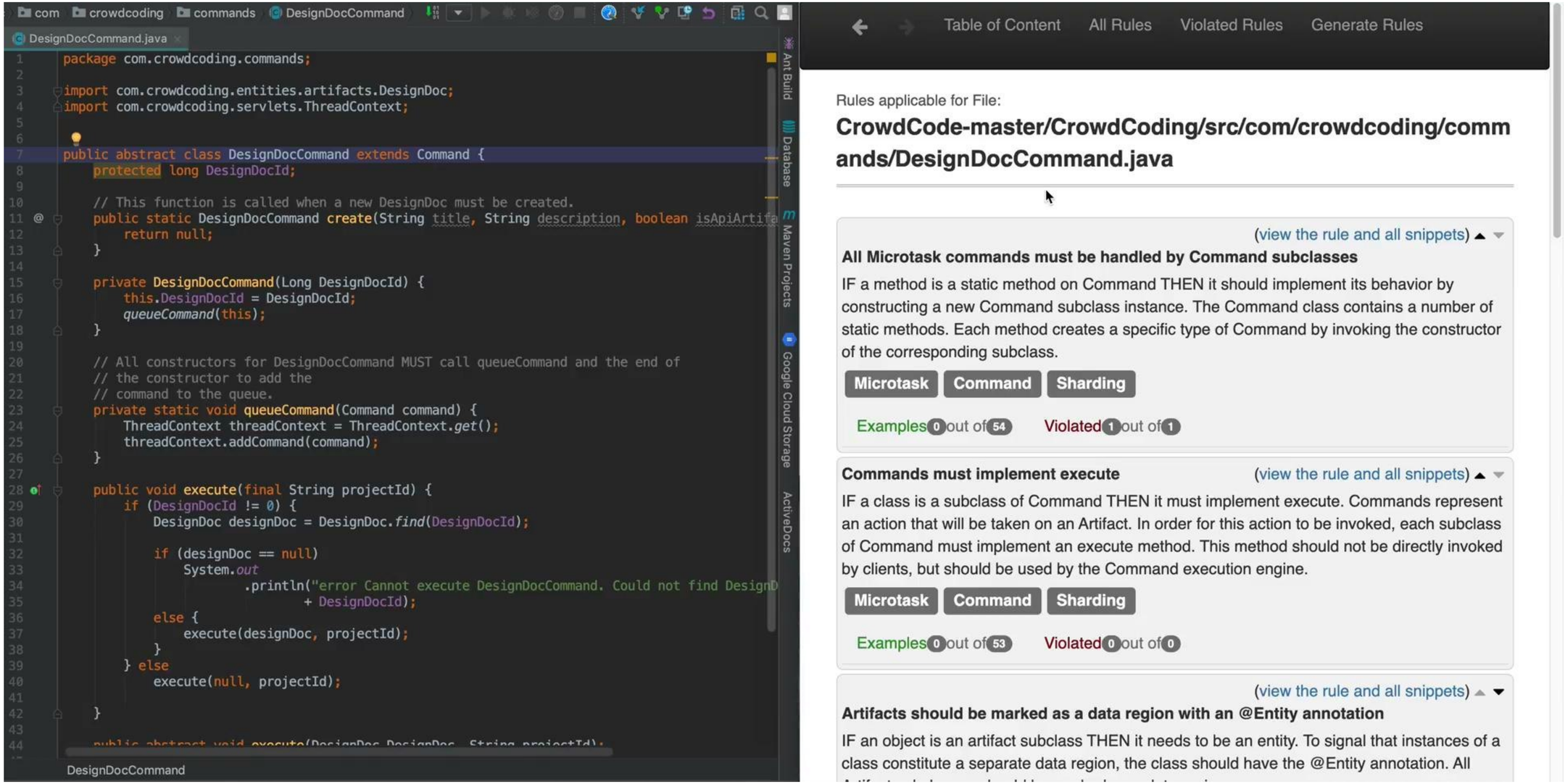
[Mehrpour et al. VL/HCC 2019]

ActiveDocumentation



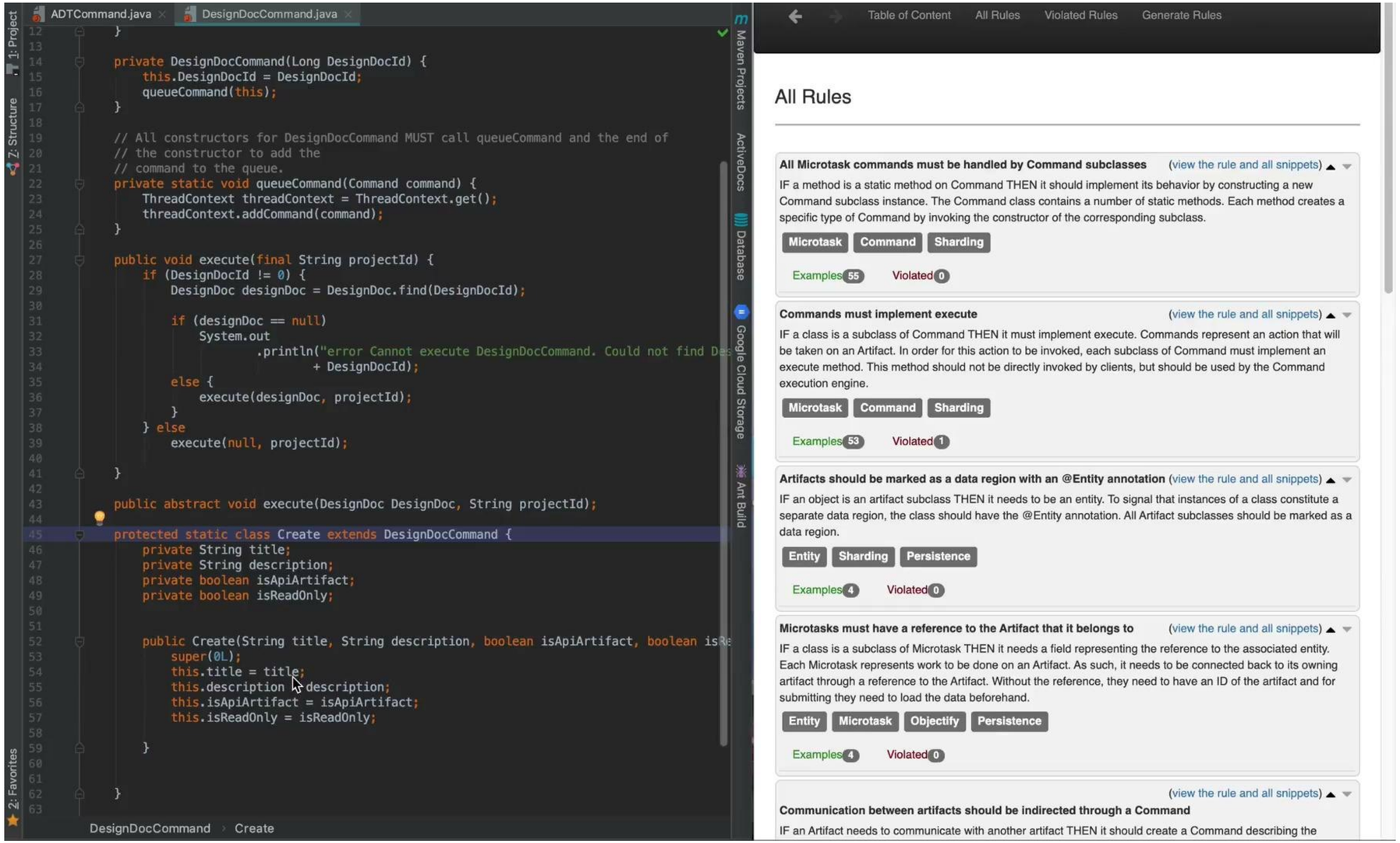
[Mehrpour et al. VL/HCC 2019]

ActiveDocumentation: Active Link



[Mehrpour et al. VL/HCC 2019]

ActiveDocumentation: Active Check



[Mehrpour et al. VL/HCC 2019]

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.

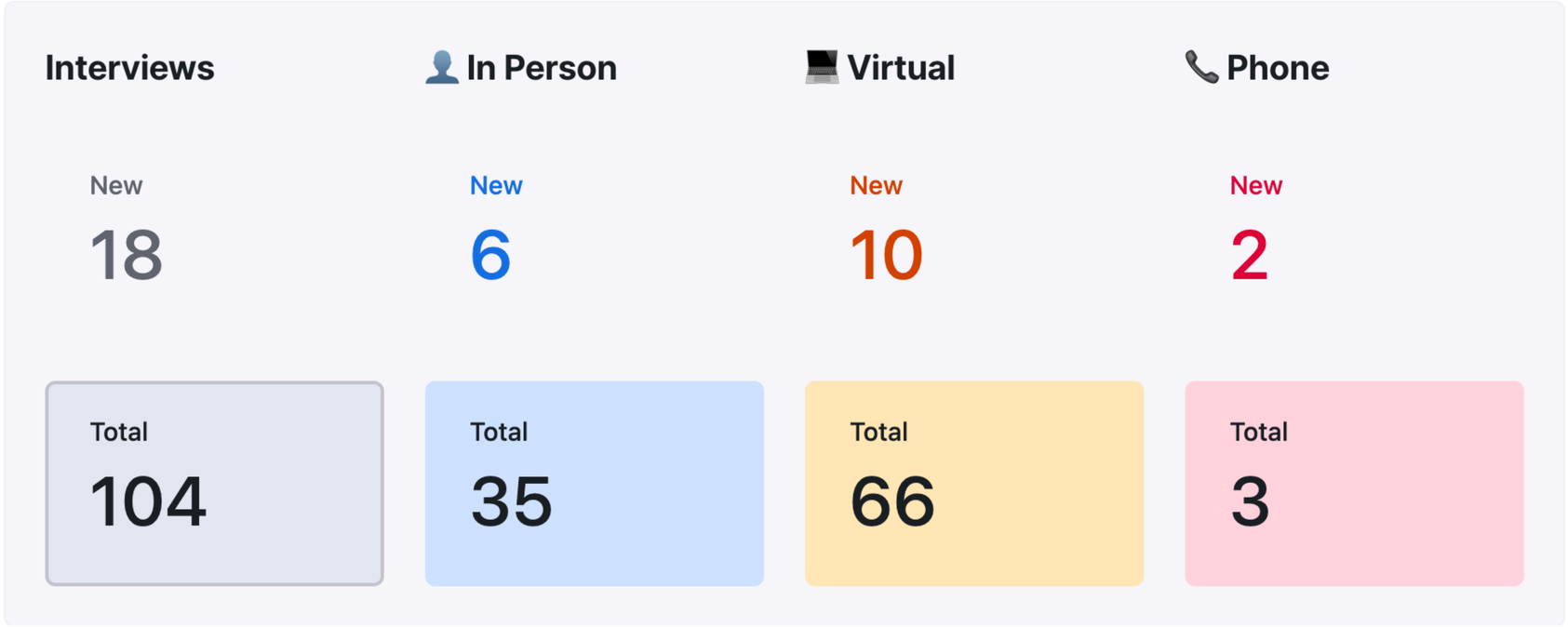
[Mehrpour et al. VL/HCC 2019]

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

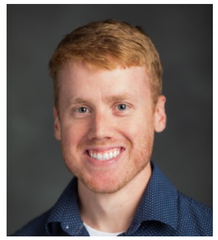
  +16 years experience in different roles in software industry.

- 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?

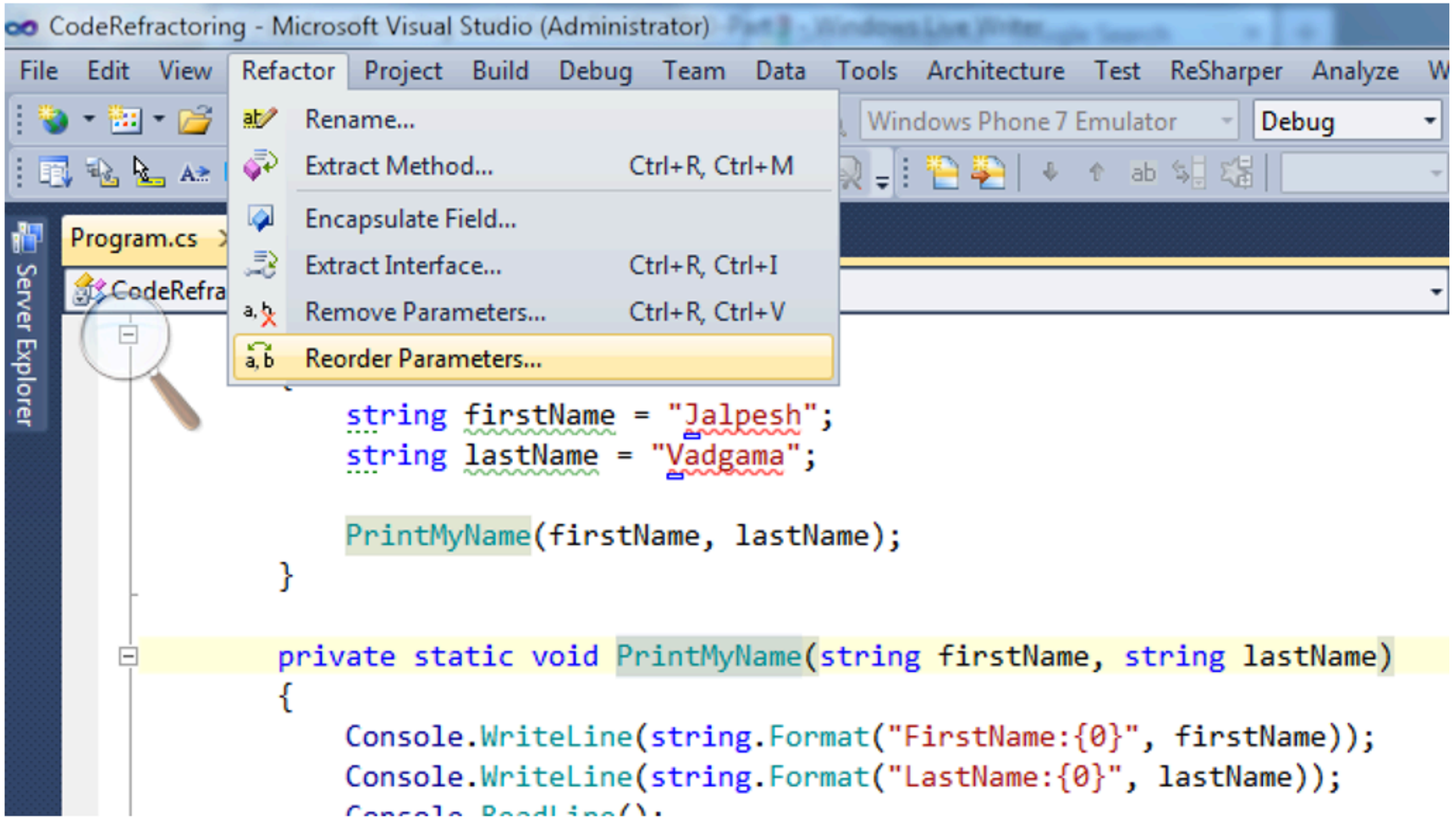
“Companies are willing to pay for tools that makes developers do less work.”

- 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