

# Building a Rich Domain Model as an Effective Defense by Design

---



**Zoran Horvat**

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com



# Command-Query Separation

## Command

Modifies state  
Returns void

## Query

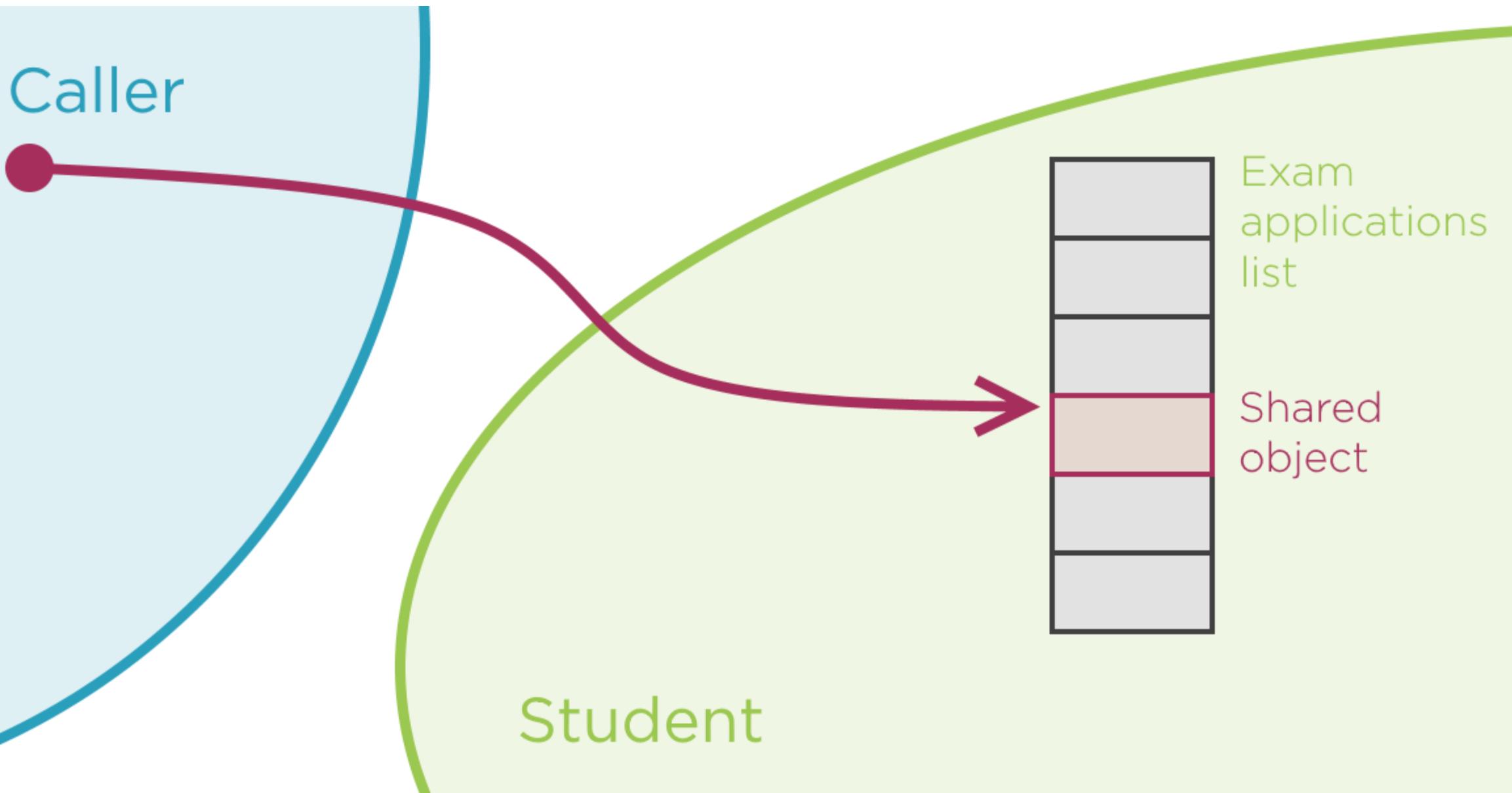
Returns a result  
Does not change state

```
public void ApplyFor(Exam exam)
{
    <no result allowed>
}
```

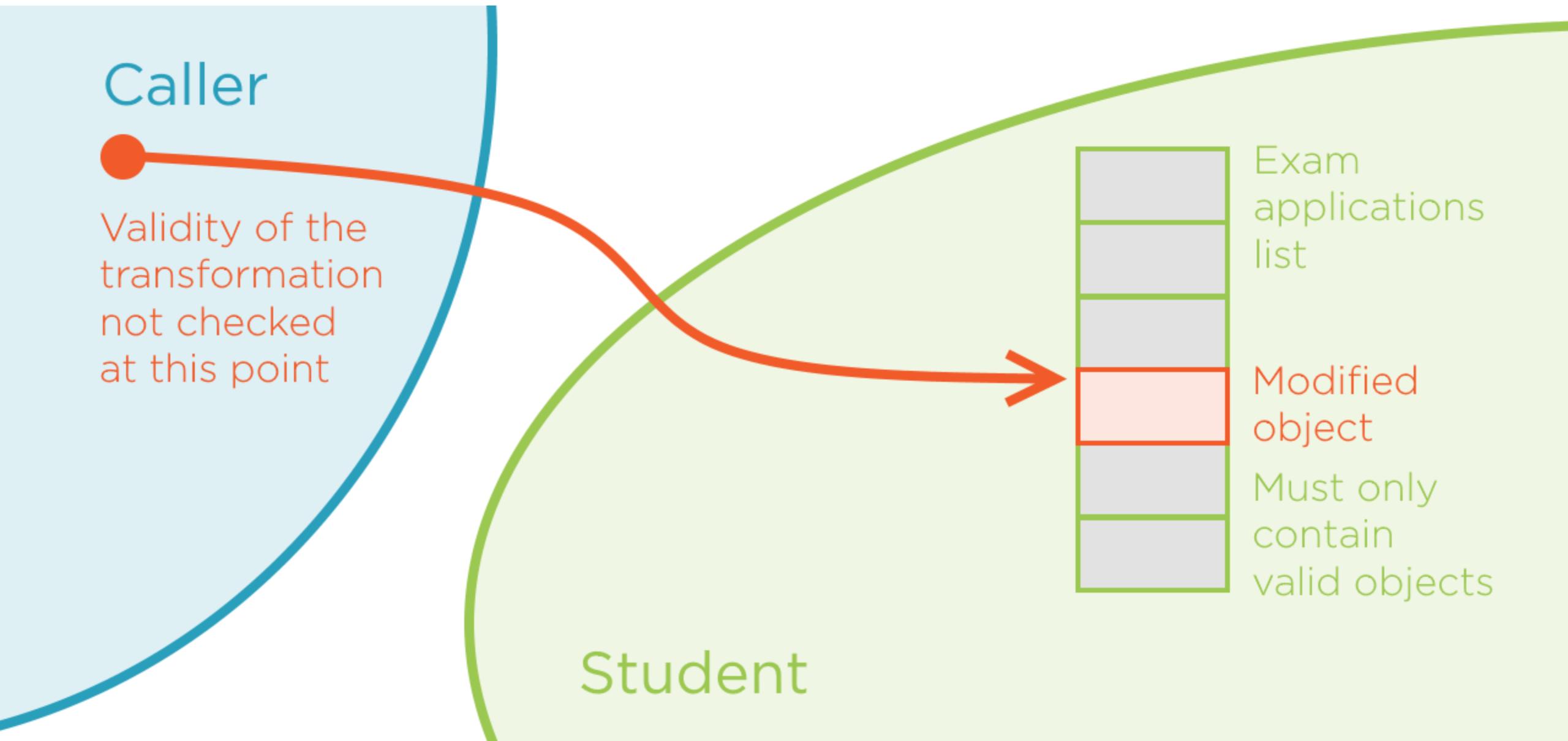
```
public IExamApplication ApplyFor(Exam exam)
{
    <no changes allowed>
}
```



# The Problem of Sharing References



# The Problem of Sharing References



# Aliasing Bugs

## Refer to:

Martin Fowler, AliasingBug

<https://martinfowler.com/bliki/AliasingBug.html>

## The problem

An object modifies  
a shared reference  
without telling the other object

## What can we do?

Reapplying validation  
imposes great increase  
in code complexity



# Mutability and Aliasing Bugs

Cause	Effect
Insisting on full immutability	Increased code complexity



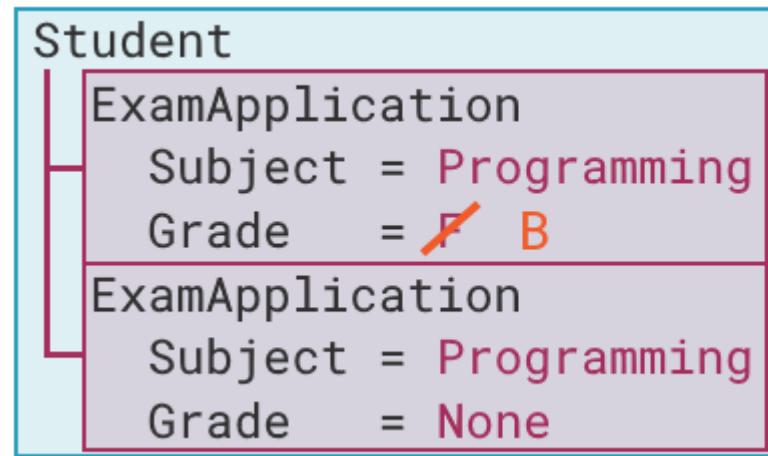
# Mutability and Aliasing Bugs

Cause	Effect
Insisting on full immutability	Increased code complexity
Sharing references between two objects	One modifies the object without telling the other (a.k.a. Aliasing bug)



# Mutability and Aliasing Bugs

Cause	Effect
Insisting on full immutability	Increased code complexity
Sharing references between two objects	One modifies the object without telling the other (a.k.a. Aliasing bug)
Modifying a shared object	May lead to faulty execution



Mutation



# Mutability and Aliasing Bugs

Cause	Effect
Insisting on full immutability	Increased code complexity
Sharing references between two objects	One modifies the object without telling the other (a.k.a. Aliasing bug)
Modifying a shared object	May lead to faulty execution
Avoiding aliasing bugs	Deploy defense all around, or Turn shared objects immutable



# Constrained Mutability

## Assigning a grade

There is no prior grade



Modifies existing grade



## Applying for an exam

There is no prior grade



There is a prior F grade



There is a prior grade better than F



# Constrained Mutability

```
ExamApplication  
Subject = Programming  
Grade   = None
```

```
ExamApplication  
Subject = Programming  
Grade   = F
```

```
ExamApplication  
Subject = Programming  
Grade   = None
```

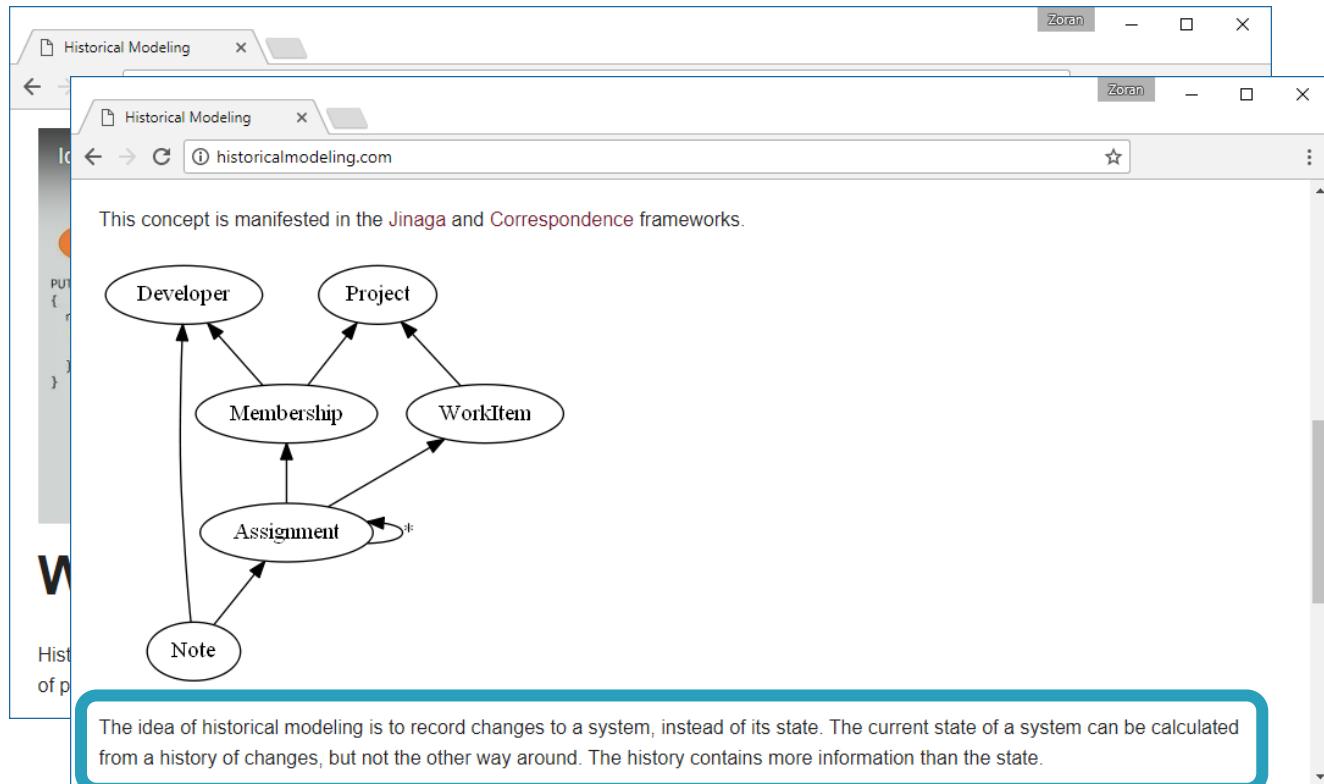
```
ExamApplication  
Subject = Programming  
Grade   = B
```

An idea:  
Only allow changes  
in forward direction  
(And don't use mutable  
objects as index keys!)



# Historical Modeling

Michael L. Perry, Historical Modeling  
<http://historicalmodeling.com>



“ The idea of historical modeling is to record changes to a system, instead of its state.

The current state of a system can be calculated from a history of changes, but not the other way around.

The history contains more information than the state.



# Historical Modeling



## Insert-only, append-only storage

- Forbids any data updates
- Change is inserted as the new record
- State is reconstructed from history of changes



# Historical Modeling



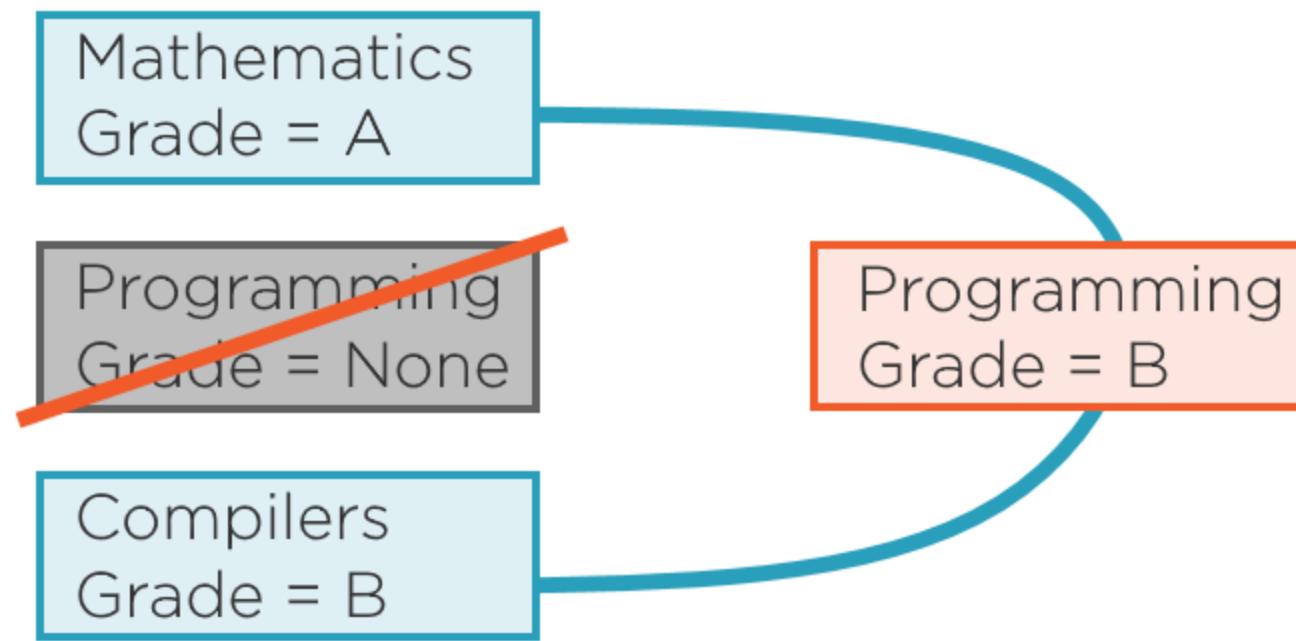
## Event Sourcing in Domain-driven Design

- Refer to Martin Fowler, EventSourcing  
[https://martinfowler.com/eaaDev/  
EventSourcing.html](https://martinfowler.com/eaaDev/EventSourcing.html)
- Persist events that happen, not state
- Replay events, query by time, etc.
- Recreate state from history of events



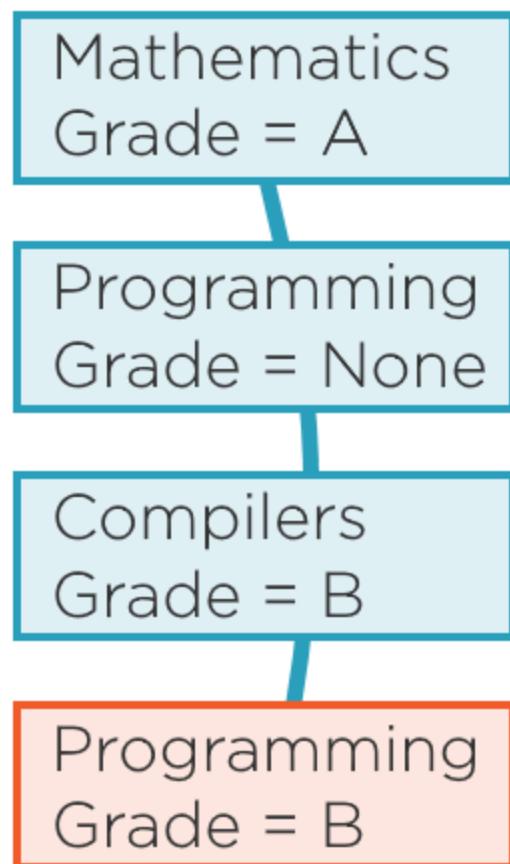
# Historical Modeling

Exam applications



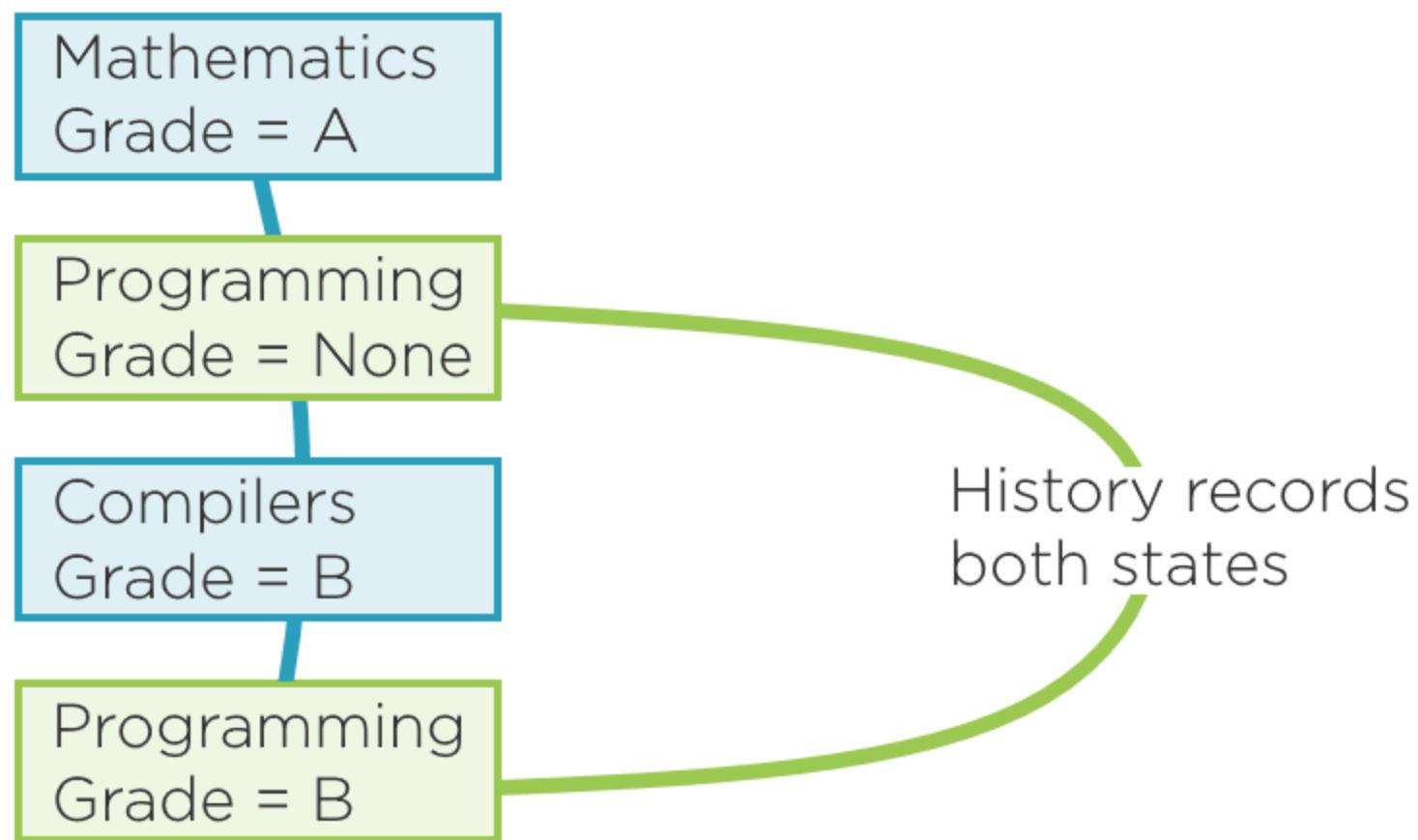
# Historical Modeling

Exam applications



# Historical Modeling

Exam applications



# Historical Modeling

Exam applications

Mathematics  
Grade = A

Programming  
Grade = None

Compilers  
Grade = B

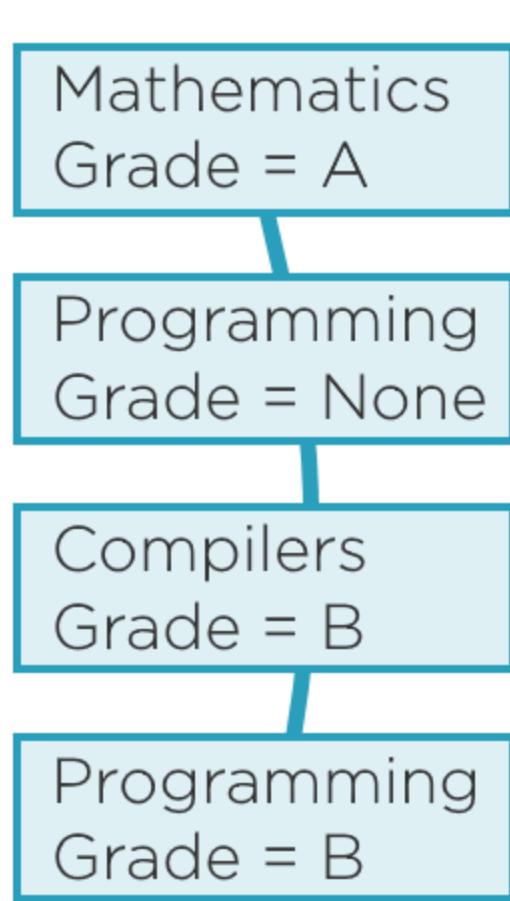
Programming  
Grade = B

History records  
both states

The latest one contains  
fresh information

# Historical Modeling

Exam applications



Read history  
bottom-up

Use stack,  
not list

# Summary



## Dealing with state mutations

- Entirely immutable object dismisses aliasing bugs
- Consumers become more complex
- Immutable consumer complicates its own consumer even more
- Complexity builds up the call chain



# Summary



## Constrained mutability

- Some mutations are allowed
- Object only passes through the sequence of valid mutations
- Object never goes back to a lesser state
- Example: Set-once properties
- Overall code complexity is reduced



# Summary



## Historical model

- Do not remember current state
- Append immutable changes instead
- History contains more information than the state
- State can be reconstructed from history of changes
- Reduced code complexity
- Safe from aliasing bugs



# Summary



## Persisting complex models

- Think of what is important
- Mission-critical domain models should remain persistence-ignorant
- Less important models can be persistence-friendly

***Next module***

*Alternative Workflows and Exceptions*

