

# Cracking the Code to Secure Software

 @DanielDeogun @DanielSawano





Daniel Deogun

**omega  
point.**



Daniel Sawano

**AVANZA** 



*@DanielDeogun @DanielSawano #SecureByDesign*

# What's Cracking the Code... all about?

*“A mindset and strategy for creating secure software by focusing on good design”*

*- Secure by Design*



@DanielDeogun @DanielSawano #SecureByDesign

**omega  
point.**

**AVANZA** 

The AVANZA logo features the word "AVANZA" in a bold, sans-serif font, followed by a graphic of four vertical bars of increasing height.

# What we'll cover today

- ☐ Solve a real security problem using good design
- ☐ Immutable mutability
- ☐ Detecting accidental leakage of sensitive data





# Case 1: Cross Site Scripting (XSS)

Some website



Webform

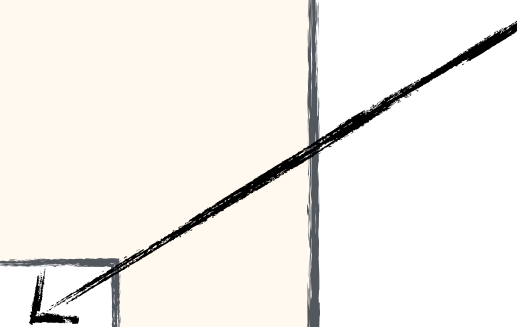
Phone #

Input:

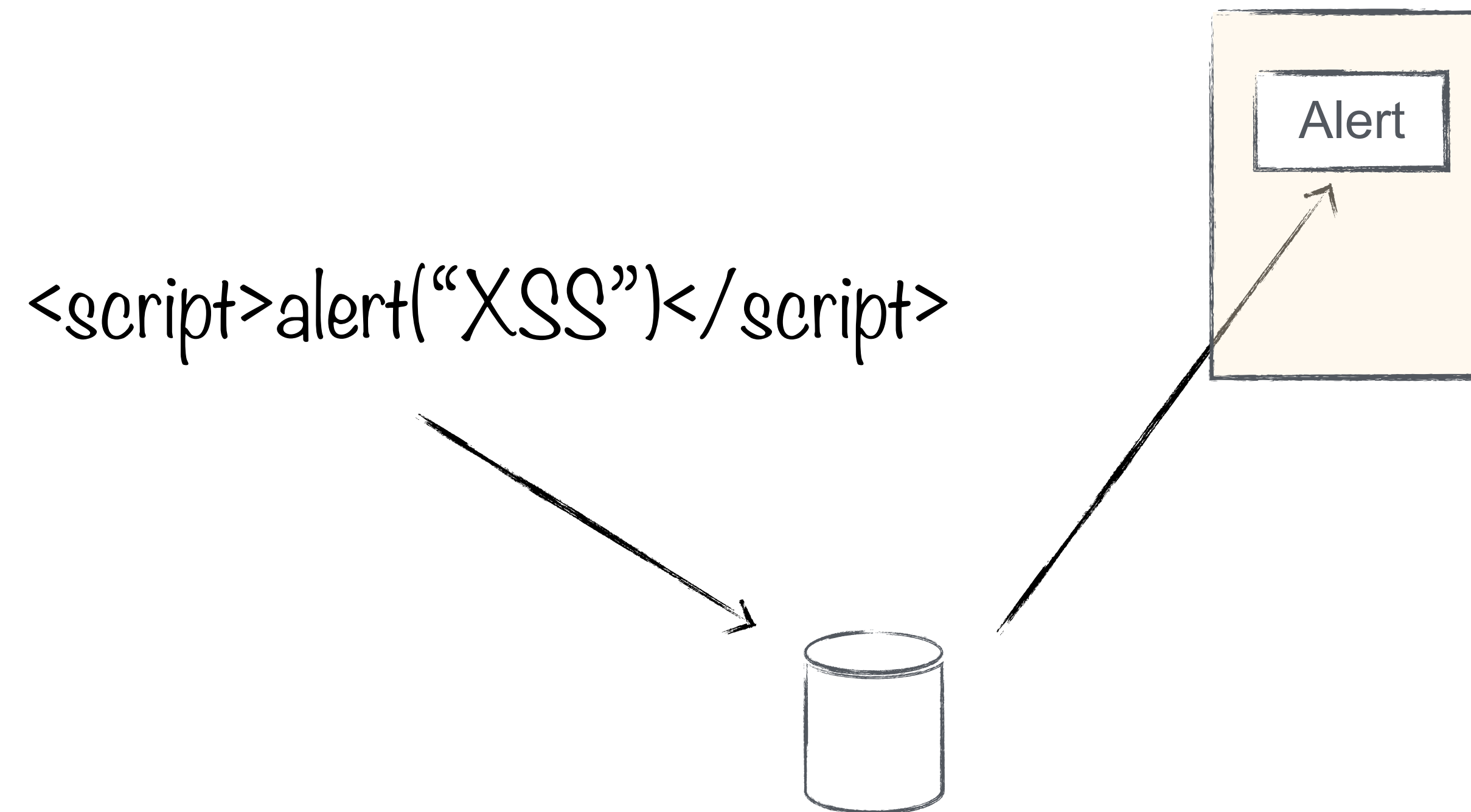
+46 8 545 106 90

or

`<script>alert("XSS")</script>`

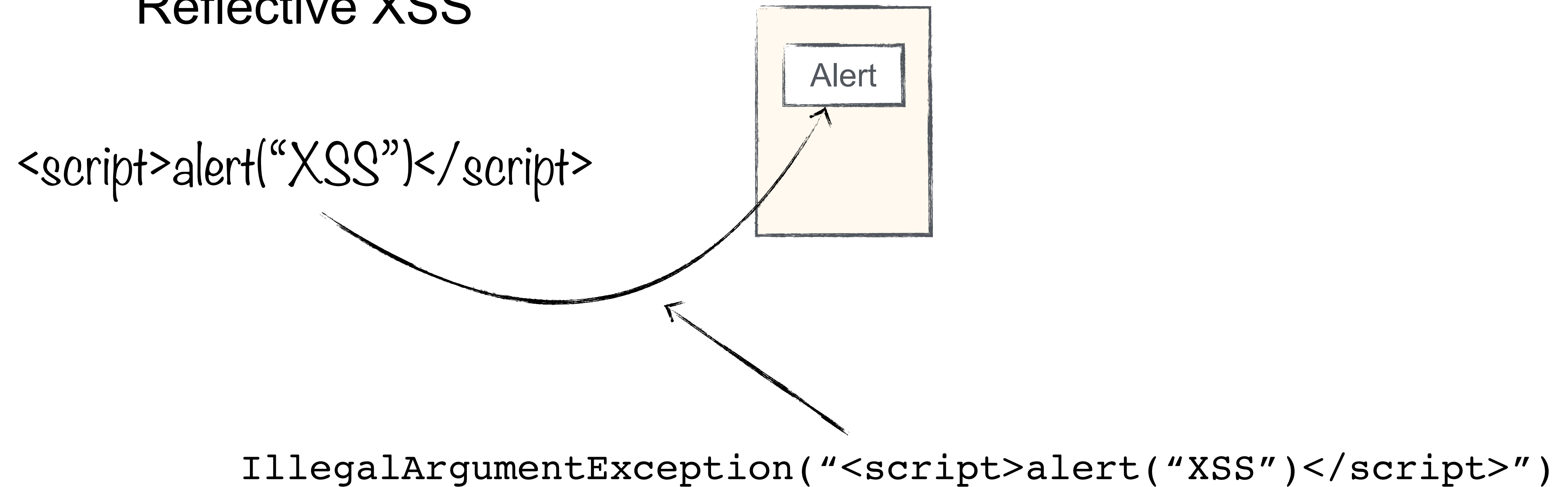


# Stored XSS



# Reflective XSS

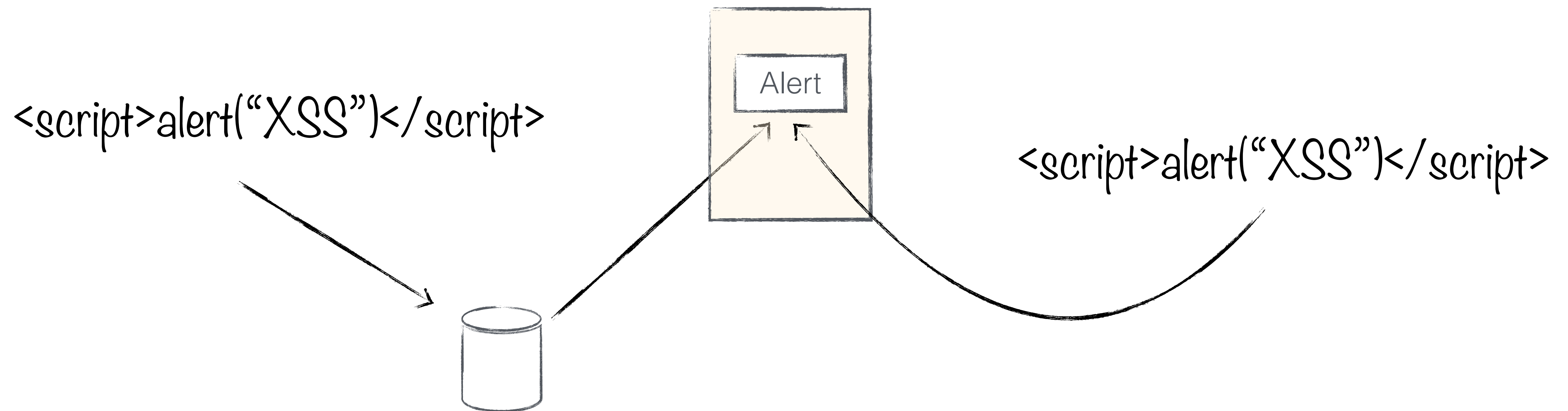
## Reflective XSS





# Technical Analysis

“Phone number” isn’t escaped properly when rendered on the website – hence, it gets interpreted as code!



# Technical Solution

Escape phone number so it can be rendered as text

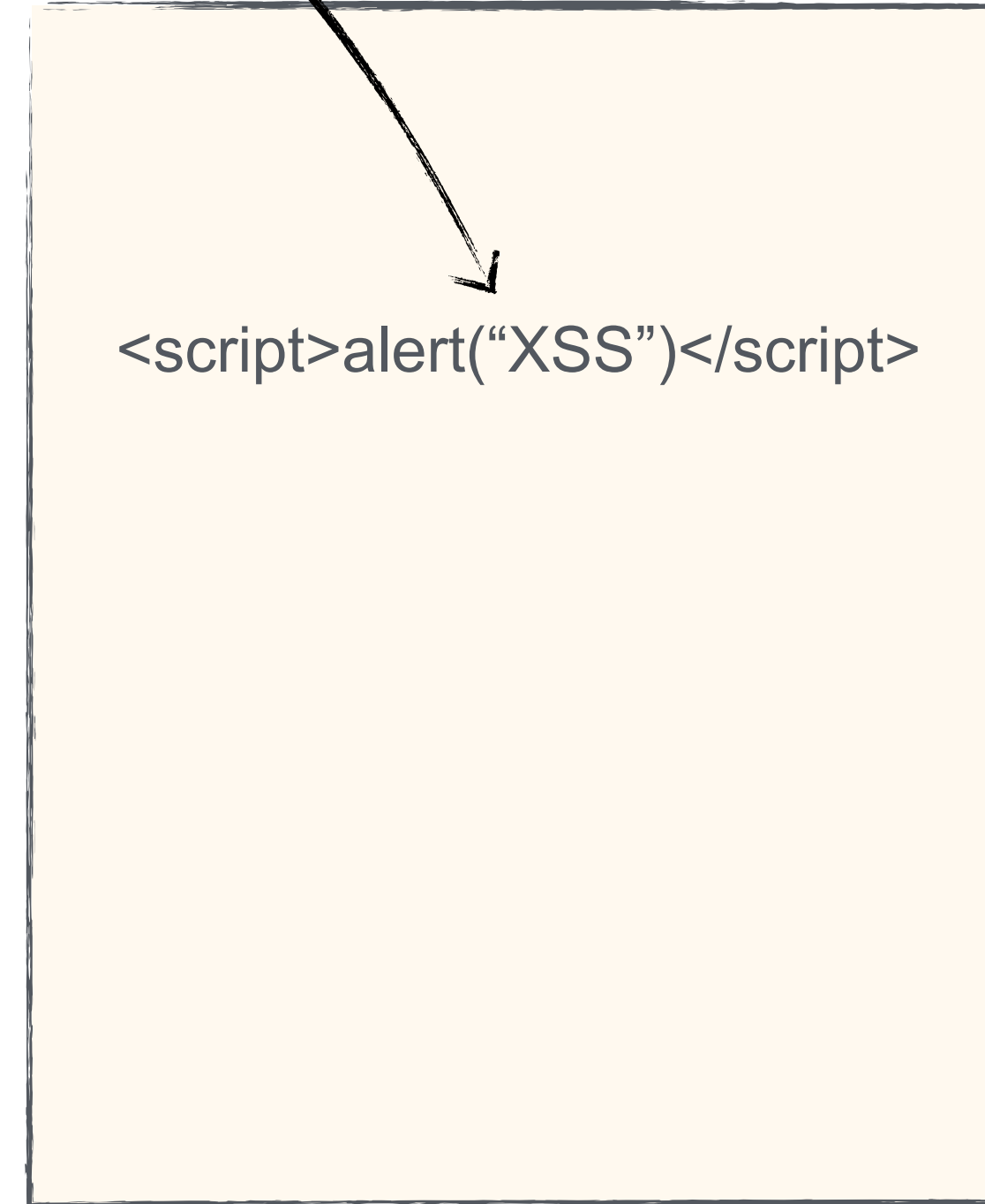
`<script>alert("XSS")</script>`



`&lt;script&gt;alert(&ldquo;XSS&rdquo;)&lt;/script&gt;`

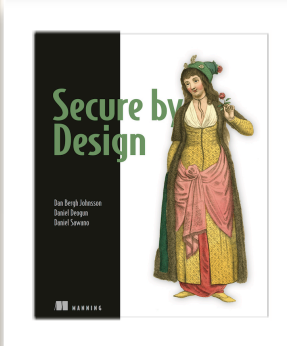
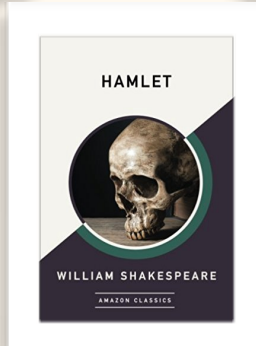
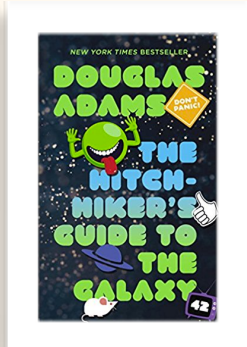


`<script>alert("XSS")</script>`



# Case 2: Buying -1 books

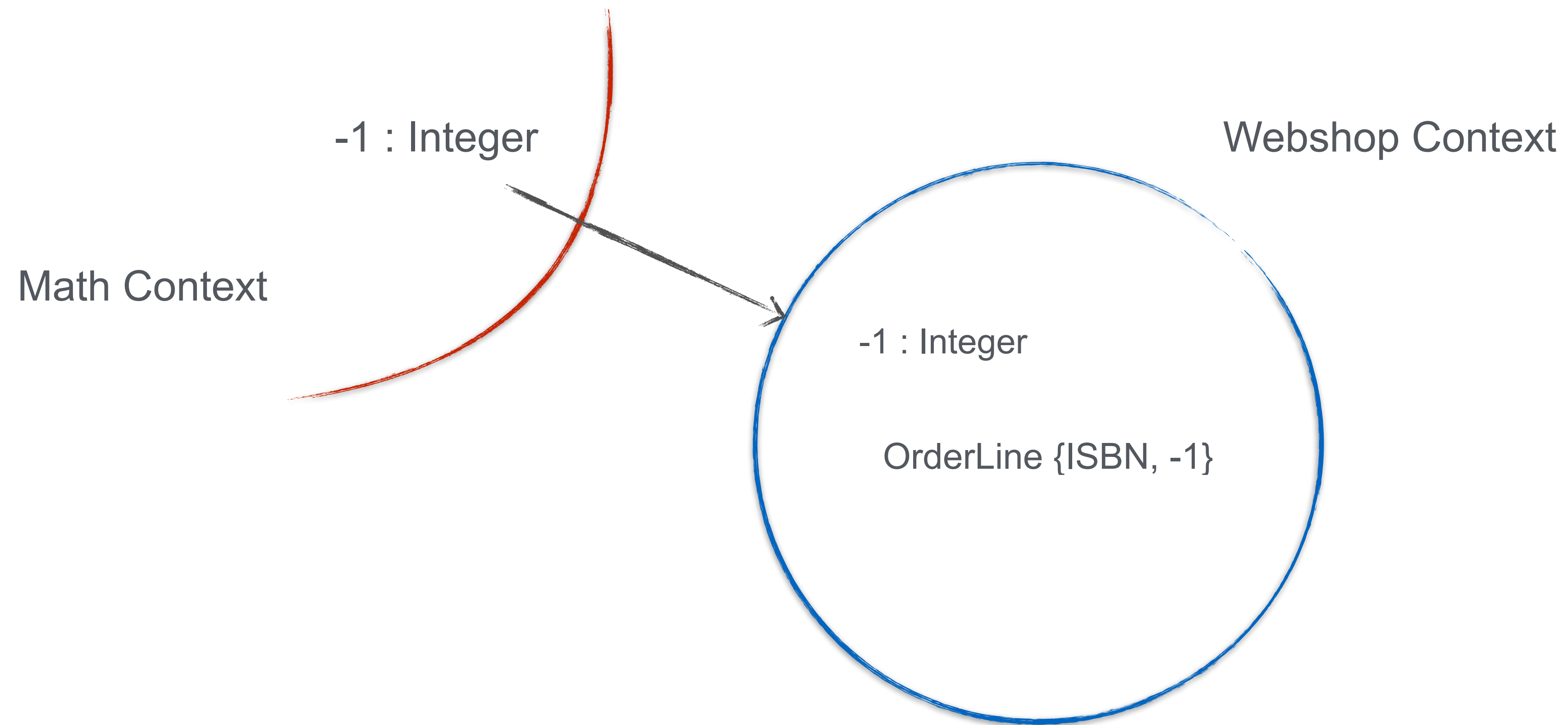


Shopping Cart		
	1 Secure by Design	\$49.99
<hr/>		
	-1 Hamlet	\$40.50
<hr/>		
	1 Hitchhiker's Guide to the Galaxy	\$30.00
Total		\$39.49





# Analysis



# But Quantity isn't an integer...

## Integers form an Abelian Group

- Closure:  $a + b = \text{integer}$
- Associativity:  $a + (b + c) = (a + b) + c$
- Commutativity:  $a + b = b + a$
- Identity:  $a + 0 = a$
- Inverse:  $a + (-a) = 0$

## Quantity

- a concept that's well defined with strict boundaries
- not closed under addition
- cannot be negative



# Domain Primitives

*“A value object so precise in its definition that it, by its mere existence, manifests its validity is called a Domain Primitive.”*

- Secure by Design

- Building block that's native to *your domain*
- Valid in the *current context*
- *Immutable* and resemble a value object in DDD



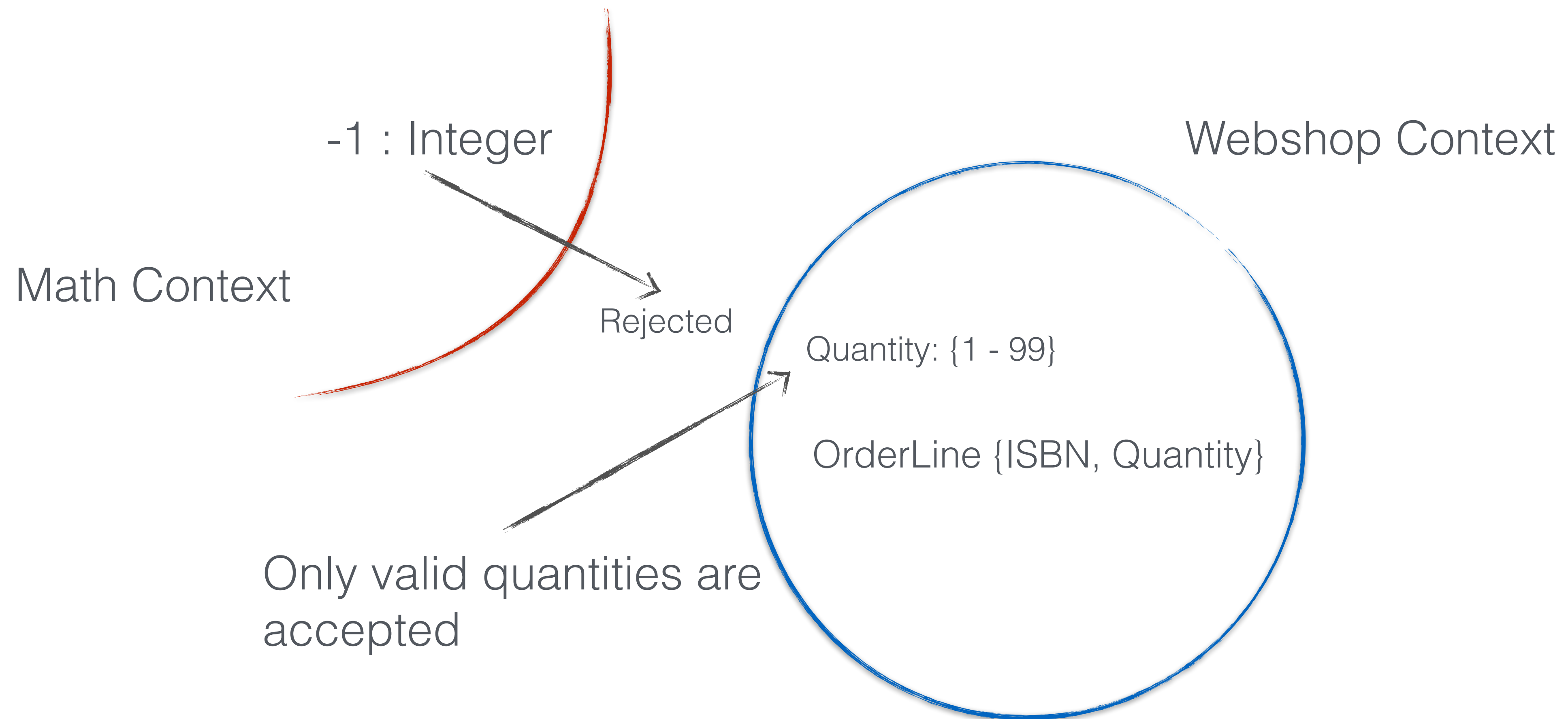


# Quantity as a Domain Primitive

```
public final class Quantity {  
    private final int value;  
  
    public Quantity(final int value) {  
        inclusiveBetween(1, 99, value);  
  
        this.value = value;  
    }  
  
    //Domain specific quantity operations...  
}
```



# Invalid quantities are rejected



# Domain Primitives tighten your design

Domain Primitives tighten your design by explicitly stating requirements and assumptions.

They also make it harder to inject data that doesn't meet the expectations.

Let's see if this pattern allows us to address XSS attacks implicitly.





# We want to prevent invalid phone numbers...

Webform

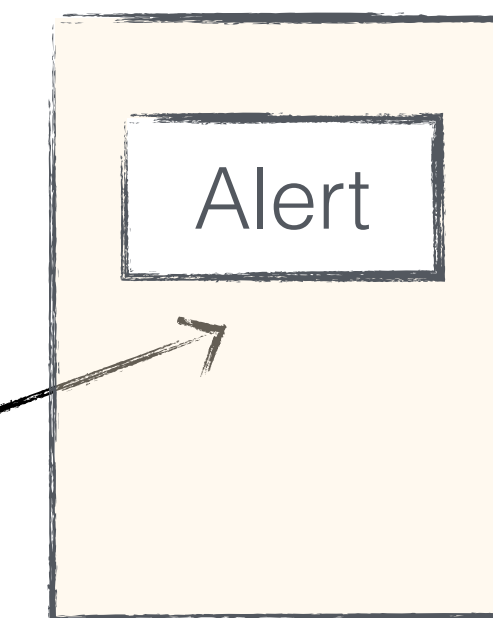
Phone #

Input:

+46 8 545 106 90

or

<script>alert("XSS")</script>



```
public void register(final String phoneNumber) {  
    // Register phone number logic  
}
```



# But `String` Accepts Anything!

Could be anything!

Attackers look at this

Input:

+46 8 545 106 90

or

<script>alert("XSS")</script>

```
public void register(final String phoneNumber) {  
    // Register phone number logic  
}
```

Developers mostly look at this to understand the intention



# Use a Domain Primitive Instead

Can only be valid phone numbers  
by definition!

Input:

+46 8 545 106 90 ✓

or

~~<script>alert("XSS")</script>~~

```
public void register(final PhoneNumber phoneNumber) {  
    // Register phone number logic  
}
```



# Domain Primitives prevent XSS

The PhoneNumber domain primitive enforce domain rule validation at creation time.

This reduces the attack vector to data that meets the rules in the context where it's used.

`<script>alert("XSS")</script>` doesn't meet the rules and rejected by design.

But what about escaping – do we need it?





# But...

... it becomes a lot of classes!



13

... isn't it overly complex?



14

... what about performance?



15





# What we'll cover today

- ☒ Solve a real security problem using good design
- ☐ Immutable mutability
- ☐ Detecting accidental leakage of sensitive data



# CIA



**Confidentiality** – data must only be disclosed to authorized users

**Integrity** – data modification is only allowed in an authorized manner

**Availability** – data must be available when needed



# Availability and Mutable State

Mutable state makes it difficult to apply horizontal scaling of an application.

Ensuring availability along with mutable state is hard.

So, is there a design pattern that both facilitates availability and mutability?



# Design Stereotypes in DDD

**Value objects** are immutable objects that don't have a conceptual identity – we only care about its value, e.g. a business card or a \$100 bill. We replace value objects with **Domain Primitives** to make them secure.

**Entities** are objects that aren't identified by their attributes, but rather by their identity and lifespan – for example, a customer or a court case.

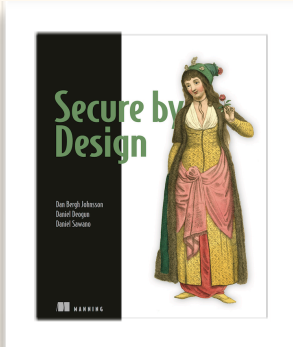
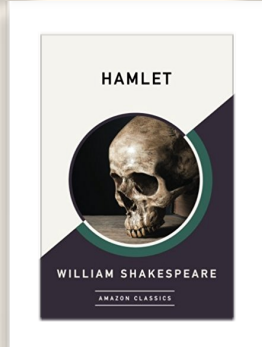
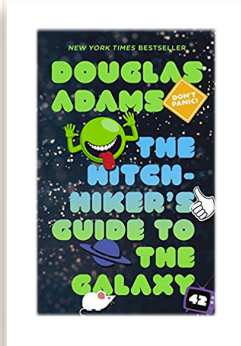


# How should we represent an Order?

An order may change state (open, closed, paid, etc).

Should it be an entity or domain primitive?

How can we solve the problems that comes with mutability?

Shopping Cart		
	1 Secure by Design	\$49.99
<hr/>		
	1 Hamlet	\$40.50
<hr/>		
	1 Hitchhiker's Guide to the Galaxy	\$30.00
		Total \$120.49





# Entity Snapshots

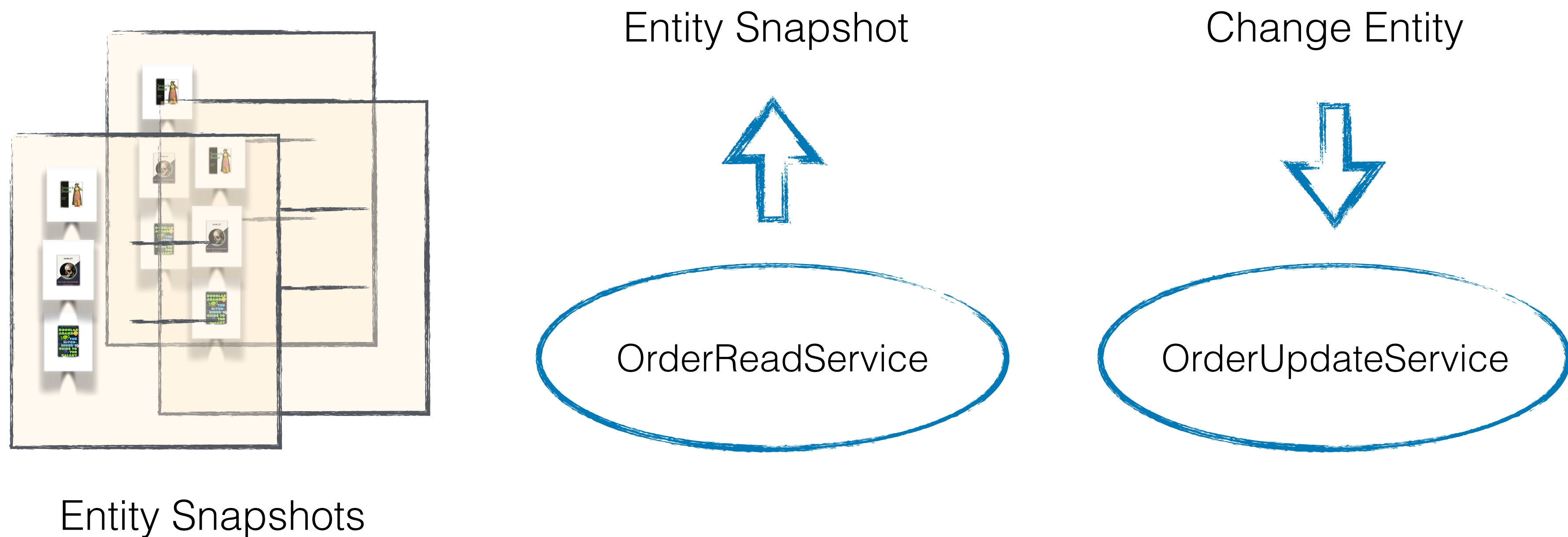
Entities are often mutable by design, but we don't need to implement it as a mutable object in code.

If we separate mutating operations from read operations, the representation of an entity can be immutable.

This makes the entity “look” like a Domain Primitive that facilitate availability and scalability!



# Order as an Entity Snapshot





# But...

... what about performance?



[5]

... isn't it overly complex?



[4]



@DanielDeogun @DanielSawano #SecureByDesign

omega  
point.

AVANZA 



# Entity Snapshots

- Removes many of the issues with mutable state such as
  - Availability
  - Consistency
- Gets all benefits from Domain Primitives



# What we'll cover today

- ☒ Solve a real security problem using good design
- ☒ Immutable mutability
- ☐ Detecting accidental leakage of sensitive data





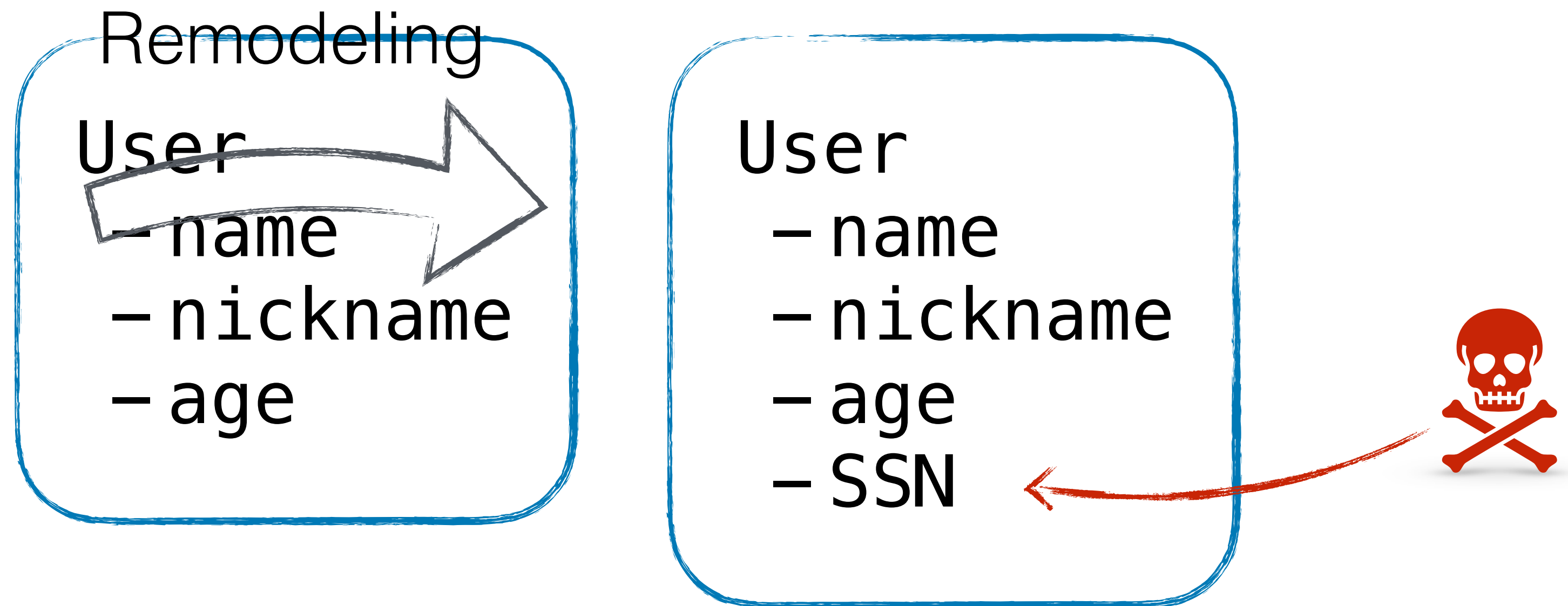
# Accidental Leakage

Typical causes:

- Logs
- Session persistence
- Evolving domain model



# Evolving domain model






# Read-once Object

```
public final class SensitiveValue implements Externalizable {  
  
    private transient final AtomicReference<String> value;  
  
    public SensitiveValue(final String value) {  
        // Check domain-specific invariants  
        this.value = new AtomicReference<>(value);  
    }  
  
    public String value() {  
        return notNull(value.getAndSet(null), "Sensitive value has already been consumed");  
    }  
  
    @Override  
    public String toString() {  
        return "SensitiveValue{value=*****}";  
    }  
  
    @Override  
    public void writeExternal(final ObjectOutput out) {  
        throw new UnsupportedOperationException("Not allowed on sensitive value");  
    }  
  
    @Override  
    public void readExternal(final ObjectInput in) {  
        throw new UnsupportedOperationException("Not allowed on sensitive value");  
    }  
}
```



# What we'll cover today

-  Solve a real security problem using good design
-  Immutable mutability
-  Detecting accidental leakage of sensitive data



# Summary

Many security weaknesses can be avoided using *Secure by Design*

- **Domain Primitives**

- *significantly reduce the attack surface*
- *facilitate security in depth*
- *reduce the risk of injection attacks*

- **Read-once objects**

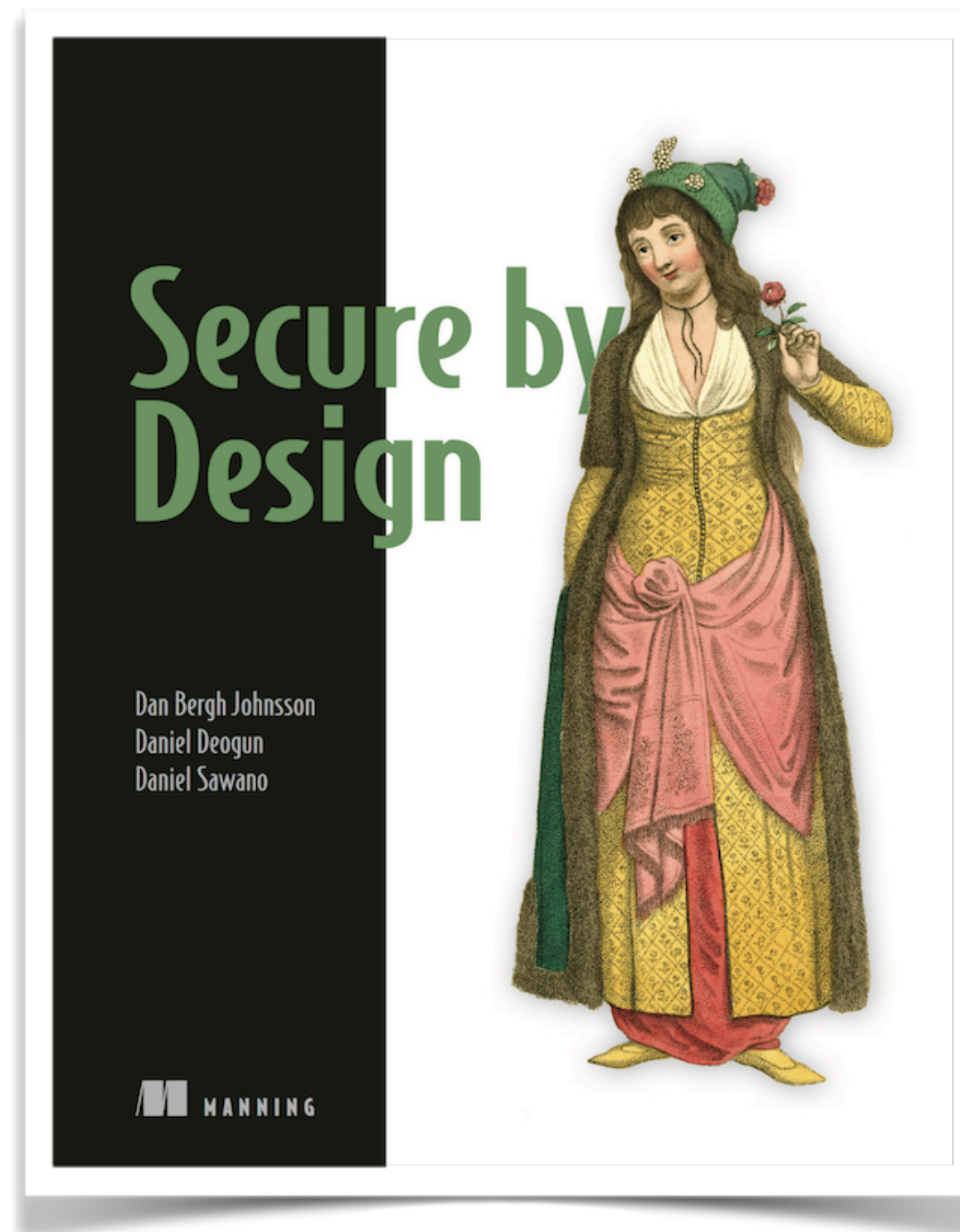
- *detects accidental data leakage*

- **Entity Snapshot**

- *immutable*
- *takes on similar properties of a domain primitive*
- *facilitate availability and scalability*







[bit.ly/secure-by-design](https://bit.ly/secure-by-design)

48% E-book Discount Code for GeeCon! **sbdgc17**



@DanielDeogun @DanielSawano #SecureByDesign



# QA



[2]



@DanielDeogun @DanielSawano #SecureByDesign

**omega**  
**point.**

**AVANZA** 

# References

- [1] <https://www.flickr.com/photos/stewart/461099066> by Stewart Butterfield under license <https://creativecommons.org/licenses/by/2.0/>
- [2] <https://flic.kr/p/9ksxQa> <https://creativecommons.org/licenses/by-nc-nd/2.0/>
- [3] <https://flic.kr/p/2pzb2T> <https://creativecommons.org/licenses/by/2.0/>
- [4] <https://flic.kr/p/7Ro4HU> <https://creativecommons.org/licenses/by/2.0/>
- [5] <https://flic.kr/p/eGYhMw> <https://creativecommons.org/licenses/by/2.0/>
- [6] CIA, <https://goo.gl/images/DRzRcp>

