Secure by Design

Daniel & Daniel

@DanielDeogun @DanielSawano

DevDays, Vilnius 2017

About us…

Daniel Deogun

Daniel Sawano

Secure by Design

Secure by Design is a new approach to software security that lets you create secure software while still focusing on business features.

Secure by Design

*"Any activity involving **active decision making** should be considered part of the software design process and can thus be referred to as **design**."*

- Johnsson, Deogun, and Sawano

What we'll cover today…

- Domain Primitives
- Entity Snapshots
- Dealing with Legacy Code
- Security in your Pipelines

Design Patterns

Security & tests

Domain Primitives

A value object so precise in its definition that it, by its mere existence, manifests its validity is called a ***domain primitive****.*

Domain Primitives

- A Domain Primitive is very strict in its definition
- If it’s not valid then it cannot exist
- Defined in the *current domain*
- It’s preciseness brings *robustness* in your code
- It’s immutable so it will always be valid

Domain Primitives

```
import static org.apache.commons.lang3.Validate.inclusiveBetween;
import static org.apache.commons.lang3.Validate.notNull;

public final class Quantity {
   private final int value;

   public Quantity(final int value) {
      inclusiveBetween(1, 200, value);
      this.value = value;
   }

   public int value() {
      return value;
   }

   public Quantity add(final Quantity addend) {
      notNull(addend);
      return new Quantity(value + addend.value);
   }

   // ...
}
```

Quantity is not just an int!

- Enforces invariants at creation
- Provides domain operations to
- Encapsulate domain behavior

CIA

[11]

- **Confidentiality** - protecting data from being read by unauthorized users
- **Integrity** - ensures data is changed in an authorized way
- **Availability** - concerns having data available when authorized users need it

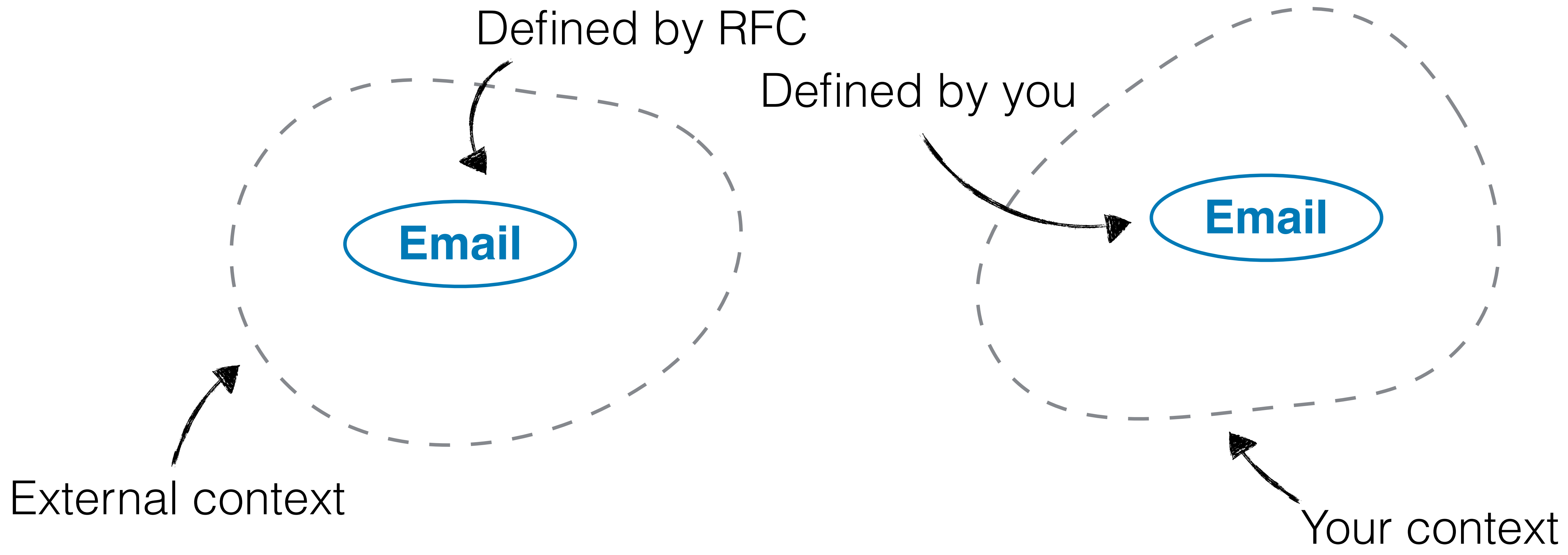
Domain Primitives

```java
import static org.apache.commons.lang3.Validate.inclusiveBetween;
import static org.apache.commons.lang3.Validate.notNull;

public final class Quantity {
   private final int value;

   public Quantity(final int value) {
      inclusiveBetween(1, 200, value);
      this.value = value;
   }

   public int value() {
      return value;
   }

   public Quantity add(final Quantity addend) {
      notNull(addend);
      return new Quantity(value + addend.value);
   }

   // ...
}
```

Quantity is not just an int!

- Enforces invariants at creation
- Provides domain operations to
- Encapsulate domain behavior

Domain Primitives



Domain Primitives

Use Domain Primitives as:

- the smallest building block in your domain model
- to build your Domain Primitive Library
- to harden your code and your APIs

What we'll cover today…

- ✓ Domain Primitives
- **Entity Snapshots**
- Dealing with Legacy Code
- Security in your Pipelines

Design Patterns

Security & tests

Entities

- An entity has an *identity* that doesn't change over time
- The values/data belonging to an entity can change over time
- Typically modeled as mutable objects

Classic Entity

```
public final class Order {

   private final OrderId id;
   private final List<OrderItem> orderItems = new ArrayList<>();

   public Order(final OrderId id) {
      this.id = notNull(id);
   }

   public void addItem(final OrderItem item) {
      notNull(item);
      orderItems.add(item);
   }

   // ...
}
```

Perils of mutable state

- Mutability is a source of security issues
- Consistency in the presence of contention is hard
- Contention can reduce availability

Entity Snapshots

Entity Snapshots are:

- Securing mutable state by making it immutable
- An immutable representation of a mutable entity
- Solves many of the security problems with regular entities

Entity Snapshots

```
public final class Order {

   private final OrderId id;
   private final List<OrderItem> orderItems;

   public Order(final OrderId id, final List<OrderItem> orderItems) {
      noNullElements(orderItems);
      notNull(id);
      this.id = id;
      this.orderItems = unmodifiableList(new ArrayList<>(orderItems));
   }

   public List<OrderItem> orderItems() {
      return orderItems;
   }

   // ...
}
```

Entity Snapshots

```
public final class WritableOrder {

   private final OrderId id;
   private final OrderRepository repository;

   public WritableOrder(final OrderId id, final OrderRepository repository) {
      this.id = notNull(id);
      this.repository = notNull(repository);
   }

   public void addOrderItem(final OrderItem orderItem) {
      notNull(orderItem);
      isOkToAdd(orderItem);
      repository.addItemToOrder(id, orderItem);
   }

   private void isOkToAdd(final OrderItem orderItem) {
      // domain validation logic to ensure it's ok to add order
   }
}
```

What we'll cover today…

- ✓ Domain Primitives
- ✓ Entity Snapshots
- **Dealing with Legacy Code**
- Security in your Pipelines

Design Patterns

Security & tests

Dealing with Legacy Code

3 good design patterns

[6]

Draw the Line

[7]

Harden your APIs

[8]

Declutter Entities

“Draw the Line”

- We need to identify the semantic boundary of a context
- Add a layer that internally translates data to a domain primitive and the back again

```
data -> domain primitive -> data
```

- This way, we have created a validation boundary that protects the inside from bad input
- But, if rejecting data is to harsh, consider logging it for insight

[6]

“Harden the API”

- Create a library of domain primitives
- Express your APIs with your domain primitives
- Never accept generic input if you have specific requirements

[7]

Generic → Specific

```
void buyBook(String, int)  ⟶  void buyBook(ISBN, Quantity)
```

“Decluttering Entities”

```
import static org.apache.commons.lang3.Validate.notNull;
import static org.apache.commons.lang3.Validate.isTrue;
public class Order {

   private final List<Object> items;
   private boolean paid;

   public void addItem(String isbn, int qty) {
        if (this.paid == false) {
            notNull(isbn);
            isTrue(isbn.length() == 10);
            isTrue(isbn.matches("[0-9X]*"));
            isTrue(isbn.matches(“[0-9]{9}[0-9X]”));
            if (inventory.avaliableBooks(isbn, qty)) {
                Book book = bookcatalogue.findBy(isbn);
                items.add(new OrderLine(book, qty));
            }
        }
    }
```

[8]

“Decluttering Entities”

```
import static org.apache.commons.lang3.Validate.notNull;
import static org.apache.commons.lang3.Validate.isTrue;
public class Order {

   private final List<Object> items;
   private boolean paid;

   public void addItem(final ISBN isbn,
                       final Quantity quantity) {
        notNull(isbn);
        notNull(quantity);
        isTrue(notPaid());

        if (inventory.avaliableBooks(isbn, quantity)) {
            Book book = bookcatalogue.findBy(isbn);
            items.add(new OrderLine(book, quantity));
        }
    }
```

[8]

What we'll cover today…

- ✓ Domain Primitives
- ✓ Entity Snapshots
- ✓ Dealing with Legacy Code
- **Security in your Pipelines**

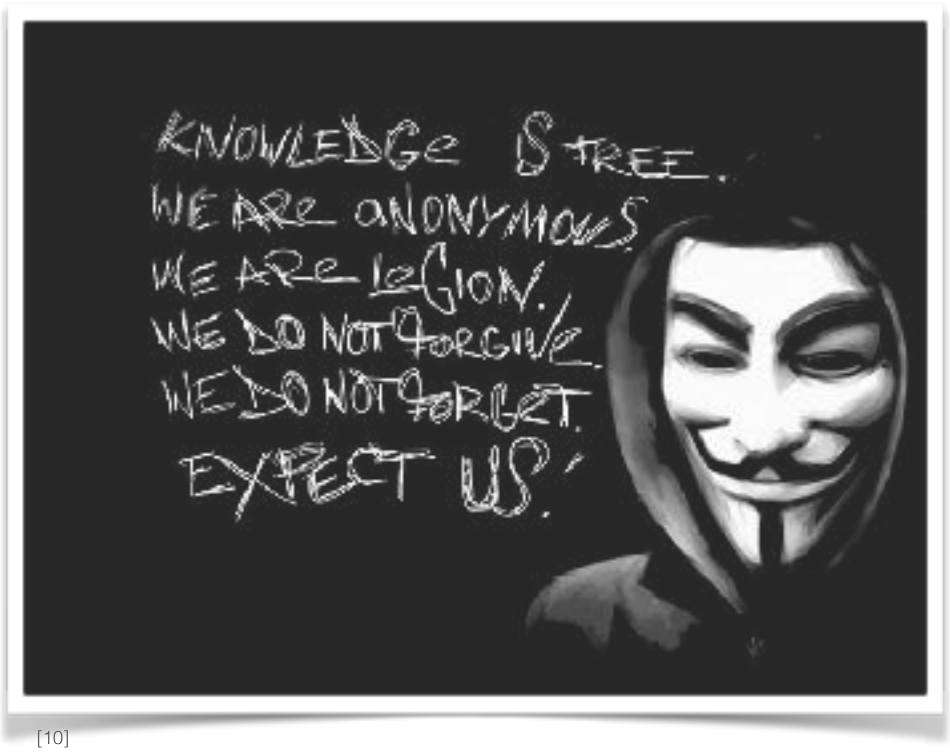
Design Patterns

Security & tests

Security in your Pipelines

- Unit testing

[12]

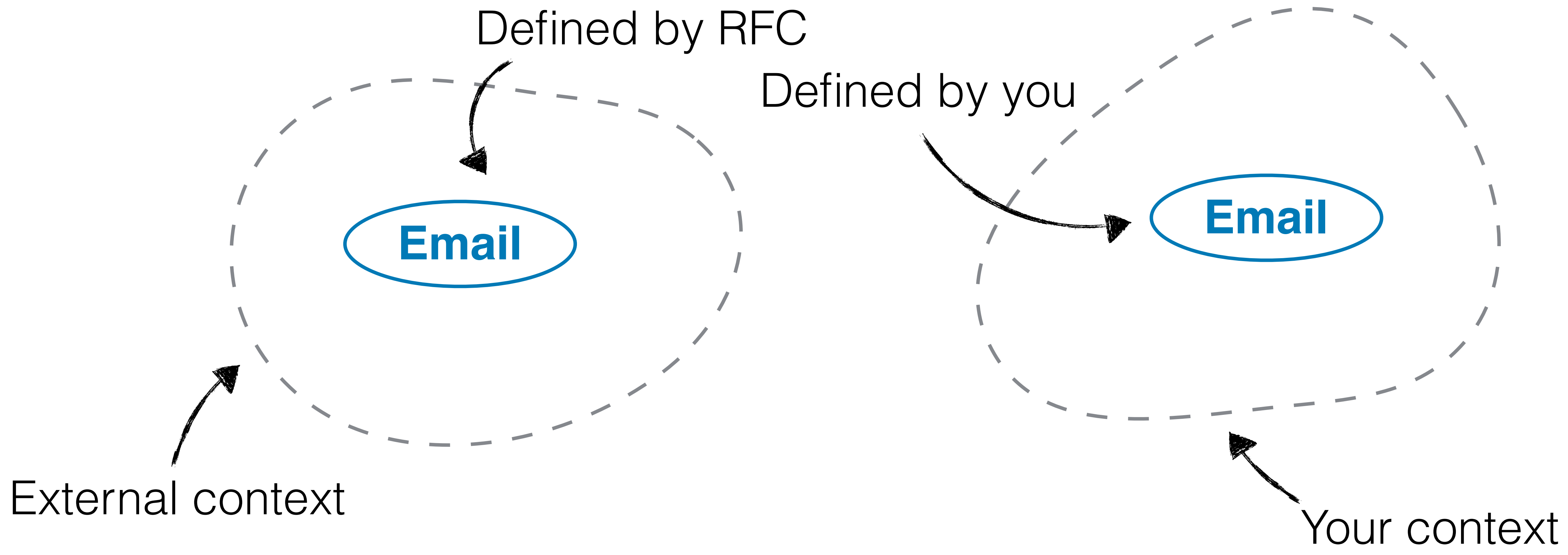


[10]

The Hospital Case

[9]

Email Domain Primitive



The Domain Rules

[5]

- The **format** of an email address must be ***local-part@domain***
- The **local part** cannot be longer than **64 characters**
- The **domain** must be ***hospital.com***
- **Subdomains** are not accepted
- The **minimum** length of an email address is **15 characters**
- The **maximum** length of an email address is **77 characters**
- The **local part** may only contain **alphabetic characters (a-z), digits (0-9), and one period**
- The **local part** may not start or end by a **period**

Testing **Normal** Behavior

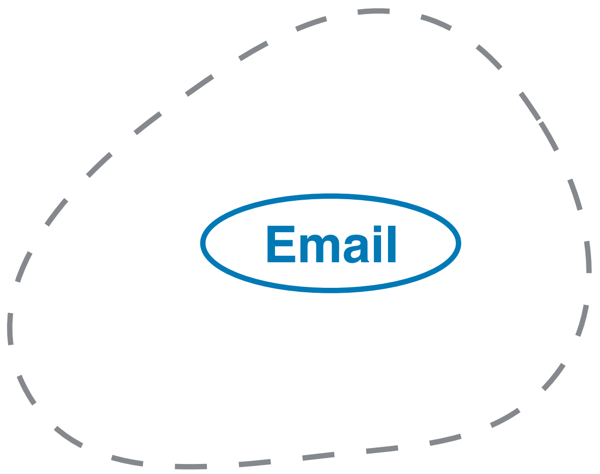
- Focus on input that clearly meets the domain rules

```
class EmailAddressTest {
  @TestFactory
  Stream<DynamicTest> should_be_a_valid_address() {
     return Stream.of(
           "jane@hospital.com",
           "jane01@hospital.com",
           "jane.doe@hospital.com")
           .map(input -> dynamicTest("Accepted: " + input,
                         () -> new EmailAddress(input)));
  }
}
```

1st version of EmailAddress

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      matchesPattern(value.toLowerCase(),
                     "^[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }
   ...
}
```

Testing **Boundary** Behavior - Acceptance



- Accept an address that's exactly 15 characters long.
- Accept an address with a local part that's 64 characters long.
- Accept an address that's exactly 77 characters long.

```
 @TestFactory
Stream<DynamicTest> should_be_accepted() {
   return Stream.of(
         "aa@hospital.com",
         repeat("X", 64) + "@hospital.com")
         .map(input -> dynamicTest("Accepted: " + input,
                     () -> new EmailAddress(input)));
}
```

Testing **Boundary** Behavior - Rejection

- Reject an address that's 14 characters long
- Reject an address with a local part that's 65 characters long
- Reject an address with a local part containing an invalid character
- Reject an address with multiple '@' symbols
- Reject an address with a domain other than *hospital.com*
- Reject an address with a subdomain
- Reject an address with a local part that starts with a period
- Reject an address with a local part that ends with a period
- Reject an address with sequential periods in the local part

Testing **Boundary** Behavior - Rejection

```
 @TestFactory
Stream<DynamicTest> should_be_rejected() {
   return Stream.of(
         "a@hospital.com",
         repeat("X", 65) + "@hospital.com",
         "address_with_invalid_char_in_local_part@hospital.com",
         "jane@doe@hospital.com",
         "jane.doe@hospital.lt",
         "jane.doe@hospital.io",
         "jane.doe@hospital.gov",
         "jane.doe@example.com",
         "jane.doe@cardio.hospital.com",
         ".jane.doe@hospital.com",
         "jane.doe.@hospital.com",
         "jane..doe@hospital.com")
         .map(input -> dynamicTest("Rejected: " + input,
                       assertInvalidEmail(input)));
}
```

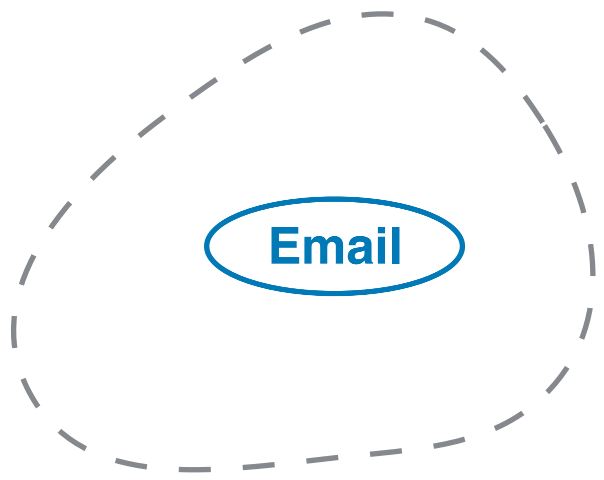
2nd version of EmailAddress

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      matchesPattern(value.toLowerCase(),
    "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }
   ...
}
```

Testing with **Invalid** Input

- Any input that doesn't satisfy the domain rules is considered invalid

- For some reason, `null`, empty strings, or "strange" characters tend to result in unexpected behavior



Testing with **Invalid** Input

```
@TestFactory
Stream<DynamicTest> should_reject_invalid_input() {
   return Stream.of(
         null,
         "null",
         "nil",
         "0",
         "",
         " ",
         "\t",
         "\n",
         "john.doe\n@hospital.com",
         "   @hospital.com",
         "%20@hospital.com",
         "john.d%20e@hospital.com",
         "john.doe.jane@hospital.com",
         "--",
         "e x a m p l e @ hospital . c o m",
         "=0@$*^%;<!->.:\\()&#\"")
         .map(input -> dynamicTest("Rejected: " + input,
            assertInvalidEmail(input)));
}
```

3rd version of EmailAddress

```
public final class EmailAddress {

    public final String value;

    public EmailAddress(final String value) {
        notNull(value, "Input cannot be null");
        matchesPattern(value.toLowerCase(),
 "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

        this.value = value.toLowerCase();
    }
    ...
}
```

Testing with **Extreme** Input

- Testing the extreme is all about identifying weaknesses in the design that makes the application break or behave strangely when handling extreme values.

```
@TestFactory
Stream<DynamicTest> should_reject_extreme_input() {
   return Stream.<Supplier<String>>of(
         () -> repeat("x", 10000),
         () -> repeat("x", 100000),
         () -> repeat("x", 1000000),
         () -> repeat("x", 10000000),
         () -> repeat("x", 20000000),
         () -> repeat("x", 40000000))
         .map(input -> dynamicTest("Rejecting extreme input",
                       assertInvalidEmail(input.get())));
}
```

Security weaknesses caused by inefficient backtracking

```
"^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$"
```

V.S

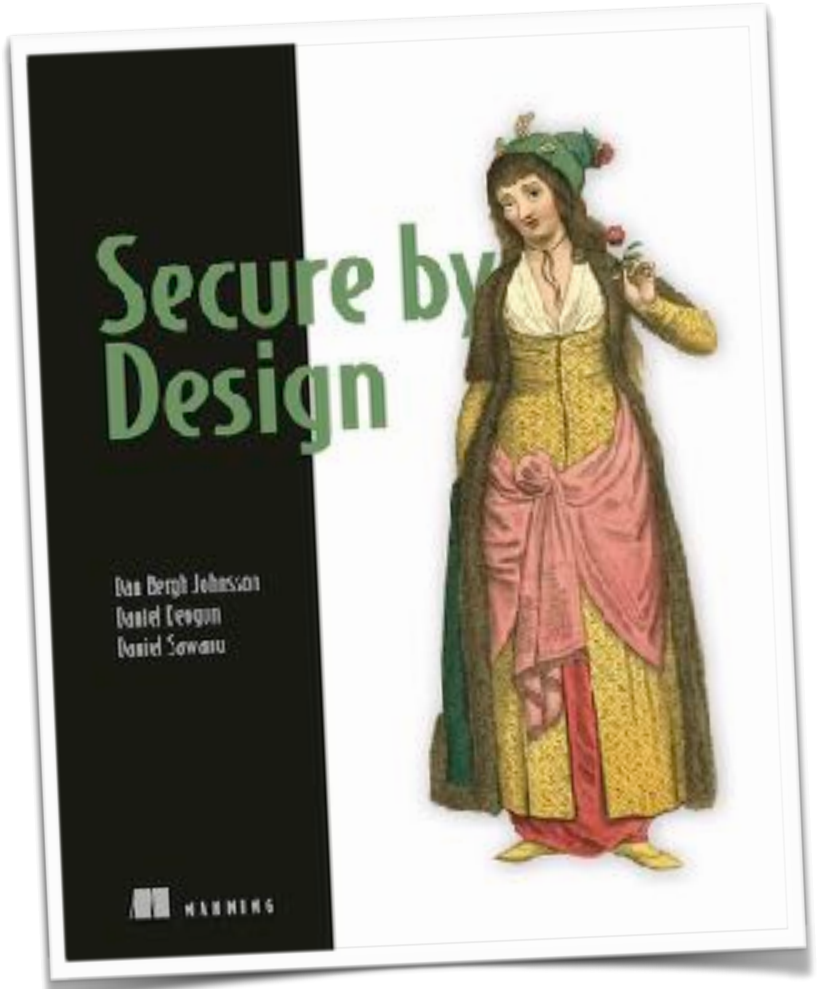
```
"^[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$"
```

What we have covered…

- ✓ Domain Primitives
- ✓ Entity Snapshots
- ✓ Dealing with Legacy Code
- ✓ Security in your Pipelines

Design Patterns

Security & tests



[1]

URL https://manning.com
Discount code: ctwdevdays

Q&A

[2]

Thank you!

@DanielSawano @DanielDeogun

References

[1] Book cover, Secure by Design, Manning Publication

[2] Question mark, https://flic.kr/p/9ksxQa by Damián Navas under license https://creativecommons.org/licenses/by-nc-nd/2.0/

[3] DDos Attack, Secure by Design, Manning Publication

[4] Uber vs Ola, https://www.bloomberg.com/news/articles/2016-03-23/uber-sues-ola-claiming-fake-bookings-as-india-fight-escalates

[5] Lyft vs Uber, http://time.com/3102548/lyft-uber-cancelling-rides/

[6] Boundary, https://flic.kr/p/nEZKMd by Graeme Fowler under license https://creativecommons.org/licenses/by/2.0/

[7] 3d key, https://flic.kr/p/e9qfrf by Yoel Ben-Avraham under license https://creativecommons.org/licenses/by-nd/2.0/

[8] Building blocks, https://flic.kr/p/agPw7C by Tiffany Terry under license https://creativecommons.org/licenses/by/2.0/

[9] Doctors Stock Photo, https://flic.kr/p/HNJUzV, by Sergio Santos under license https://creativecommons.org/licenses/by/2.0/

[10] Anonymous, https://flic.kr/p/a2eniw, by Lio Leiser under license https://creativecommons.org/licenses/by-nd/2.0/

[11] CIA, https://goo.gl/images/DRzRcp

[12] Bilfinger - Natural gas pipeline USA, https://flic.kr/p/nrFHFz by Bilfinger SE under license https://creativecommons.org/licenses/by-nd/2.0/