





# Type-Driven Domain Design

Use the types, Luke!

heise devSec – 2022-10-05

 **Bundesministerium**  
Klimaschutz, Umwelt,  
Energie, Mobilität,  
Innovation und Technologie

 **Bundesministerium**  
Digitalisierung und  
Wirtschaftsstandort



wirtschafts  
agentur  
wien  
Ein Fonds der  
Stadt Wien



**FWF**  
Der Wissenschaftsfonds.

 **netidee**  
OPEN INNOVATIONS

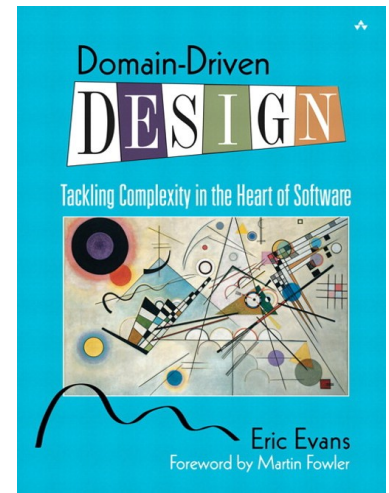
# \$ whoami

- Michael Koppmann
- Penetration tester at SBA Research
- Software engineer
- sec4dev co-founder



# Domain-Driven Design (DDD)

- Primary focus on core domain and business logic
- Iteratively refine concepts by consulting domain experts
- Uses ubiquitous language that everyone in the domain understands
- Popular concepts: Entities, Value Objects, Aggregates, Bounded Contexts, Repositories



# Type-Driven Domain Design

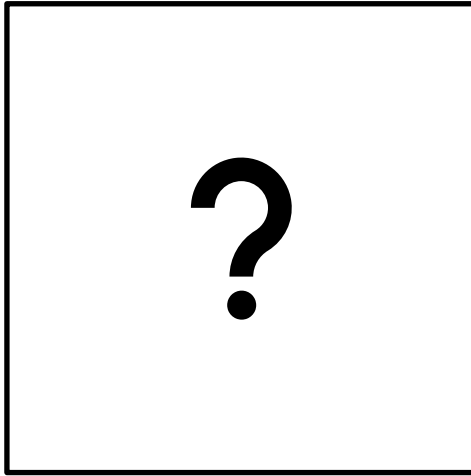
- Encode business rules into types
- Make illegal state unrepresentable
- Prevent security vulnerabilities
- Immutability avoids doing the same checks over and over again
- Offload work to the compiler



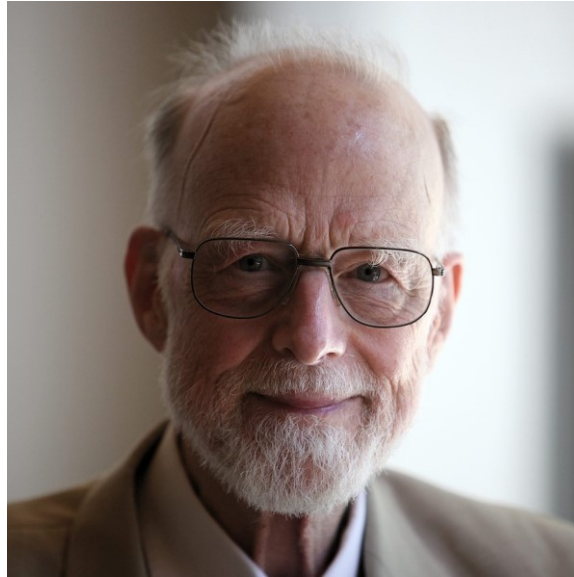
# Language Support

- Any language can be used for DDD
- But some provide more powerful type systems
- Nice to have: Sum Types (Tagged Unions), Pattern Matching
- Examples:
  - Haskell
  - Elm
  - OCaml
  - F#
  - Rust
  - Scala
  - ReScript
  - TypeScript
  - C# (7)
  - Java (17)

# Null



# Null



*"I call it my billion-dollar mistake. It was the invention of the null reference in 1965."*  
Tony Hoare



# Make the Absence of Values Explicit (Java)

```
1. Optional<String> lookup(String key, Map<String, String> map) {  
2.     String value = map.get(key);  
3.     return Optional.ofNullable(value);  
4. }  
  
5. String getContactDescriptionOrDefault(String name,  
6.                                     Map<String, String> phonebook) {  
7.     Optional<String> opt = lookup(name, phonebook);  
8.     return opt.map(number -> name + "'s number is: " + number)  
9.               .orElse("Could not find a number for " + name);  
10. }
```

# The Problem With Basic Data Types

- Relying on them too much is an anti-pattern called “Primitive Obsession”
- No enforcement of constraints
- Any manipulation could invalidate the invariants of the data

# Never Mix Up IDs Anymore (C#)

```
1. public readonly record struct UserId(Guid Id)
2. {
3.     public override string ToString() => Id.ToString();
4.     public static implicit operator Guid(UserId userId) => userId.Id;
5. }

6. void StorePaymentForUser(PaymentId paymentId, UserId userId) { ... }

7. Service.StorePaymentForUser(userId, paymentId); // COMPILER ERROR!
```

# Modeling a Contact Type

Because not everything is a string



```
1. type Contact =  
2.     {  
3.         FirstName: string;  
4.         Initial: string;  
5.         LastName: string;  
  
6.         EmailAddress: string;  
7.         // true if verification mail was confirmed  
8.         IsEmailVerified: bool;  
9.     }
```

```
1.  type PersonalName =
2.      {
3.          FirstName: string;
4.          Initial: string;
5.          LastName: string;
6.      }

7.  type EmailContactInfo =
8.      {
9.          EmailAddress: string;
10.         IsEmailVerified: bool;
11.     }

12. type Contact =
13.     {
14.         Name: PersonalName;
15.         EmailContactInfo: EmailContactInfo;
16.     }
```

```
1. let EmailAddress = private EmailAddress of string

2. module EmailAddress =

3.     let create (input: string) =
4.         if System.Text.RegularExpressions.Regex.IsMatch(input, @"^\S+@\S+\.\S+$")
5.             then Some (EmailAddress input)
6.             else None

7.     let value (EmailAddress address) = address
```

```
1.  type PersonalName =
2.      {
3.          FirstName: String50;
4.          Initial: String1 option;
5.          LastName: String50;
6.      }

7.  type EmailContactInfo =
8.      {
9.          EmailAddress: EmailAddress;
10.         IsEmailVerified: bool;
11.     }

12. type Contact =
13.     {
14.         Name: PersonalName;
15.         EmailContactInfo: EmailContactInfo;
16.     }
```



```
1. type EmailContactInfo =
2.     | Unverified of EmailAddress
3.     | Verified of VerifiedEmailAddress

4. let EmailAddress = EmailAddress of string
5. let VerifiedEmailAddress = private VerifiedEmailAddress of string

6. module EmailVerification =

7.     let storeVerificationCode (contact: EmailContactInfo, code: VerificationCode) =
8.         match contact with
9.         | Verified -> ()
10.        | Unverified (EmailAddress addr) ->
11.            // store verification code in the DB for email address.

12.     let verify (contact: EmailContactInfo, code: VerificationCode) =
13.         match contact with
14.         | Verified -> Some contact
15.         | Unverified (EmailAddress addr) ->
16.             // check if given verification code matches code
17.             // stored in DB for this mail address
18.             Some (VerifiedEmailAddress addr)

19.     let value (VerifiedEmailAddress address) = address
```

```
1. type Contact =  
2.     {  
3.         Name: PersonalName;  
4.         EmailContactInfo: EmailContactInfo;  
5.     }  
  
6. type SendPasswordResetEmail = VerifiedEmailAddress -> ...  
7. type ChangeEmail = String -> ... -> EmailContactInfo  
8.     // creates new email contact with Unverified constructor
```

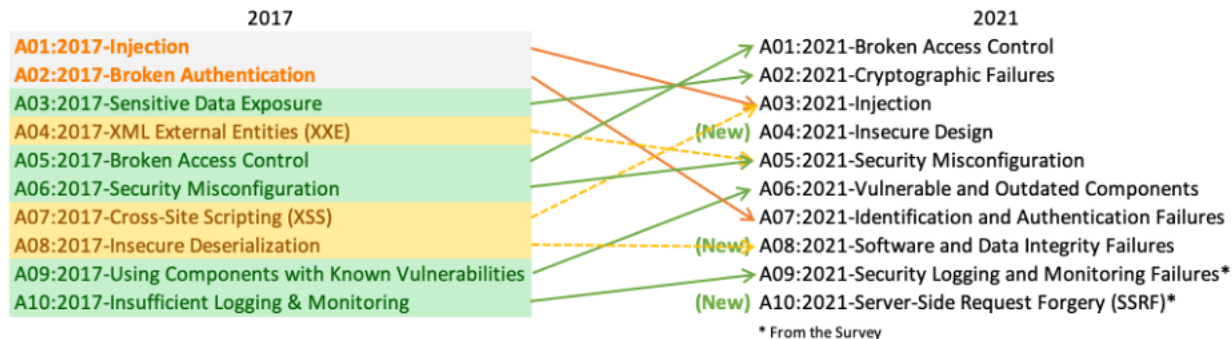
# Tackling the OWASP Top Ten

At least some of them.

Types don't solve all your problems ;)

# OWASP Top Ten (2021)

- A01:2021-Broken Access Control
- A03:2021-Injection
- A04:2021-Insecure Design



# OWASP API Security Top 10 (2019)

- API1:2019 Broken Object Level Authorization
- API3:2019 Excessive Data Exposure
- API5:2019 Broken Function Level Authorization
- API6:2019 Mass Assignment
- API8:2019 Injection

# Types For Authorization



Fighting broken access control attacks

```
1. Web.get "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   if not (AuthZ.canAccessArticle curUser articleId)
5.     then Web.status 401
6.     else do
7.       article <- Service.getArticle articleId
8.       Web.json (toViewModel article)
```

```
1. Web.delete "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   if (AuthZ.canModifyArticle curUser articleId)
5.     then Web.status 401
6.     else do
7.       Service.deleteArticle articleId
8.       Web.status 204
```

```
1. Web.put "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   articleData <- param "data"
5.   if not (AuthZ.canModifyArticle curUser articleId)
6.     then Web.status 401
7.     else do
8.       Service.putArticle articleId articleData
9.       Web.status 204
```

1.

```
canAccessArticle :: User -> Id Article -> Bool
canAccessArticle user articleId = ...

canModifyArticle :: User -> Id Article -> Bool
canModifyArticle user articleId = ...
```

2.

```
newtype AccessArticle = AccessArticle (Id Article)
newtype ModifyArticle = ModifyArticle (Id Article)

newtype AccessToken a = AccessToken a

tokenData :: AccessToken a -> a
tokenData (AccessToken data) = data
```

3.

```
getAccessArticleToken :: User -> Id Article -> Maybe
(AccessToken AccessArticle)
getAccessArticleToken user articleId =
  if canAccessArticle user articleId
  then Just (AccessToken (AccessArticle articleId))
  else Nothing

getModifyArticleToken :: User -> Id Article -> Maybe
(AccessToken ModifyArticle)
getModifyArticleToken user articleId =
  if canModifyArticle user articleId
  then Just (AccessToken (ModifyArticle articleId))
  else Nothing
```

4.

```
module Authorization
  ( AccessToken
  , tokenData
  , AccessArticle (..)
  , ModifyArticle (..)
  , getAccessArticleToken
  , getModifyArticleToken
  )
where
```



```
1. getArticle :: AccessToken AccessArticle -> IO Article
2. getArticle token = do
3.   let (AccessArticle articleId) = tokenData token
4.   Db.fetchArticle articleId

5. putArticle :: AccessToken ModifyArticle -> Article -> IO ()
6. putArticle token articleData = do
7.   let (ModifyArticle articleId) = tokenData token
8.   Db.putArticle articleId articleData

9. deleteArticle :: AccessToken ModifyArticle -> IO ()
10. deleteArticle token = do
11.   let (ModifyArticle articleId) = tokenData token
12.   Db.deleteArticle articleId
```

```
1. Web.get "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   case AuthZ.getAccessArticleToken curUser articleId of
5.     Nothing    -> Web.status 401
6.     Just token -> do
7.       article <- Service.getArticle token
8.       Web.json (toViewModel article)
```

```
1. Web.delete "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   case AuthZ.getModifyArticleToken curUser articleId of
5.     Nothing    -> Web.status 401
6.     Just token -> do
7.       Service.deleteArticle token
8.       Web.status 204
```

```
1. Web.put "/articles/:id" $ do
2.   curUser  <- AuthN.getUser
3.   articleId <- param "id"
4.   articleData <- param "data"
5.   case AuthZ.getModifyArticleToken curUser articleId of
6.     Nothing    -> Web.status 401
7.     Just token -> do
8.       Service.putArticle token articleData
9.       Web.status 204
```

# Fighting Injection Attacks (Haskell)

```
1. import qualified Database.SQLite.Simple as SQL

2. main = SQL.withConnection "products.db" $ \conn -> do
3.     putStrLn "Search by product name:"
4.     pname    <- getLine
5.     products <- getProductsByName conn pname
6.     putStrLn ("Here is the data: " ++ show products)

7. -- SQL.query :: SQL.Connection -> SQL.Query -> args -> IO [result]

8. getProductsByName :: SQL.Connection -> String -> IO [Product]
9. getProductsByName conn pname =
10.     SQL.query conn (SQL.Query "SELECT * FROM products WHERE product_name=?" ) (pname)
```

# Fighting Excessive Data Exposure (Java)

```
1.  public class User {
2.      private final Id<User> id;
3.      private Name50 firstName, lastName;
4.      private Birthdate dateOfBirth;
5.      private PasswordHash passwordHash;
6.      // Constructor, getters, setters, domain logic, etc.
7.  }
8.  public class UserViewModel {
9.      private final String fullName;
10.     private final DateTime dateOfBirth;
11.     // Constructor, getters, mapping from entity to dto and vice versa
12.  }
13.  public List<UserViewModel> getUsers(...) {
14.      List<User> users = userService.getUsers(...);
15.      return users.stream().map(UserViewModel::entityToViewModel).collect(Collectors.toList());
16.  }
```

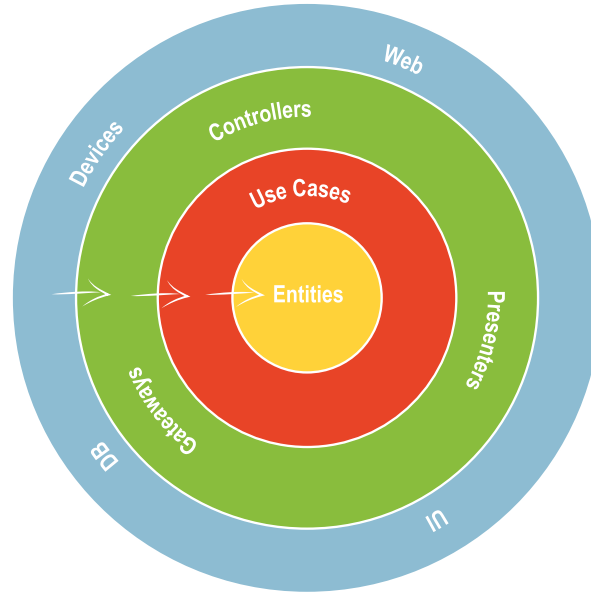
# Fighting XSS Attacks (Elm)

```
1. -- div : List (Attribute msg) -> List (Html msg) -> Html msg
2. -- text : String -> Html msg

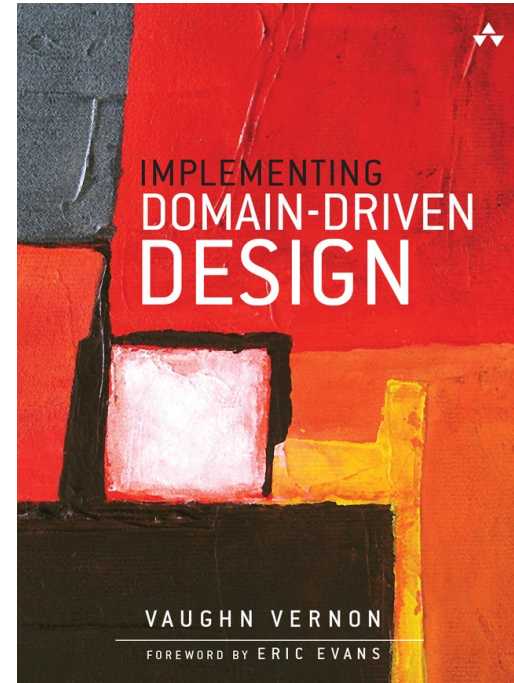
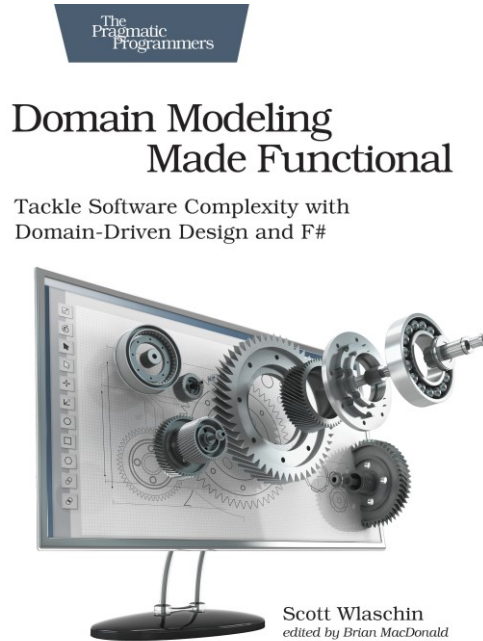
3. userNameComponent : String -> Html msg
4. userNameComponent userName =
5.     div [ class "user-name" ] [ text username ]

6. -- <div class="user-name">Foo</div>
```

# Hexagonal / Onion / Clean Architecture



# Book Recommendations



## Key Takeaways

- **Make illegal state unrepresentable**
- Encode business rules in your types
- Parse, don't validate
- Use the compiler to your advantage
- Eliminate security vulnerabilities by design



# Michael Koppmann

## SBA Research

Floragasse 7, 1040 Vienna, Austria

Email: [mkoppmann@sba-research.org](mailto:mkoppmann@sba-research.org)

Matrix: [@mkoppmann:sba-research.org](https://matrix.org/#/mkoppmann@sba-research.org)