
SBA
Research

Typed Security

Preventing vulnerabilities by design

sec4dev – 2022-09-09

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

Bundesministerium
Digitalisierung und
Wirtschaftsstandort

FFG
Forschung wirkt.

wirtschafts
agentur
wien
Ein Fonds der
Stadt Wien

European
Commission

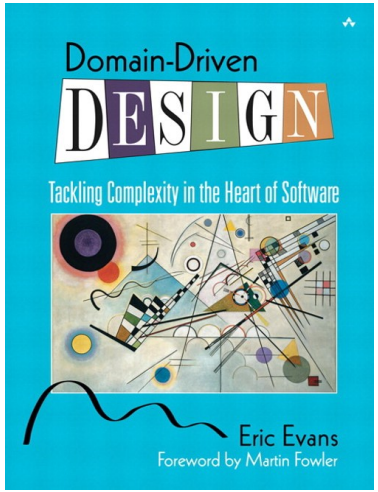
FWF
Der Wissenschaftsfonds.

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

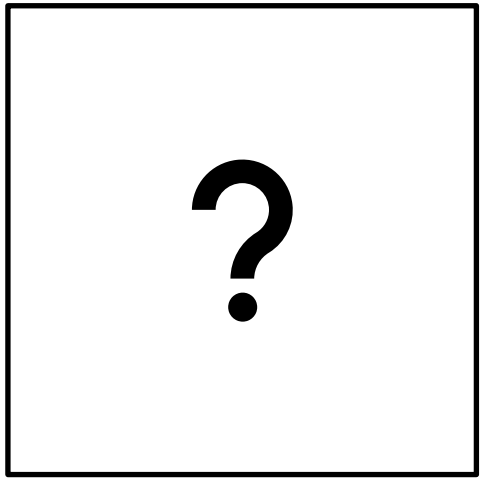
Type-Driven Domain Design

- 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

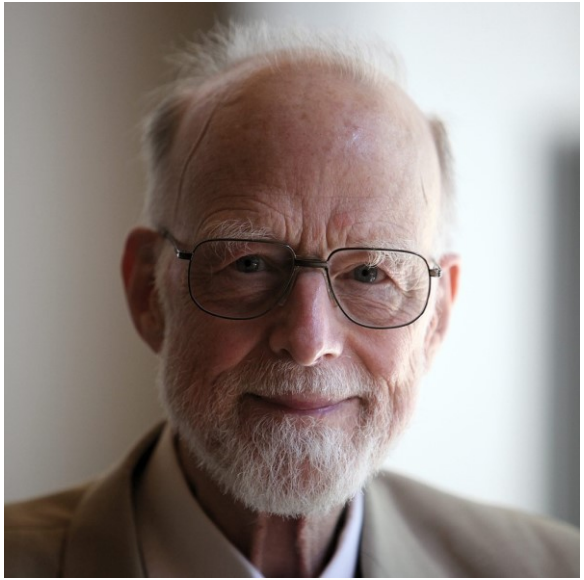
Language Support

- 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

Source: https://commons.wikimedia.org/wiki/File:Sir_Tony_Hoare_IMG_5125.jpg

Make the Absence of Values Explicit (Java)

```java
Optional<String> lookup(String key, Map<String, String> map) {
  String value = map.get(key);
  return Optional.ofNullable(value);
}

String getContactDescriptionOrDefault(String name,
                                      Map<String, String> phonebook) {
  Optional<String> opt = lookup(name, phonebook);
  return opt.map(number -> name + "'s number is: " + number)
            .orElse("Could not find a number for " + name);
}
```

The Problem With Basic Data Types

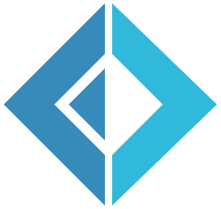
- Relying on them too much is an anti-pattern called “Primitive Obsession”
- No enforcement of constraints
- Any manipulation could invalidate the invariants of the data

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



```
type Contact =
    {
        FirstName: string;
        Initial: string;
        LastName: string;

        EmailAddress: string;
        // true if verification mail was confirmed
        IsEmailVerified: bool;
    }
```

```
type PersonalName =
    {
        FirstName: string;
        Initial: string;
        LastName: string;
    }

type EmailContactInfo =
    {
        EmailAddress: string;
        IsEmailVerified: bool;
    }

type Contact =
    {
        Name: PersonalName;
        EmailContactInfo: EmailContactInfo;
    }
```

```
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
```

```
type PersonalName =
    {
        FirstName: String50;
        Initial: String1 option;
        LastName: String50;
    }

type EmailContactInfo =
    {
        EmailAddress: EmailAddress;
        IsEmailVerified: bool;
    }

type Contact =
    {
        Name: PersonalName;
        EmailContactInfo: EmailContactInfo;
    }
```

```
type EmailContactInfo =
    | Unverified of EmailAddress
    | Verified of VerifiedEmailAddress

let EmailAddress = EmailAddress of string
let VerifiedEmailAddress = private VerifiedEmailAddress of string

module EmailVerification =

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

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

    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
```

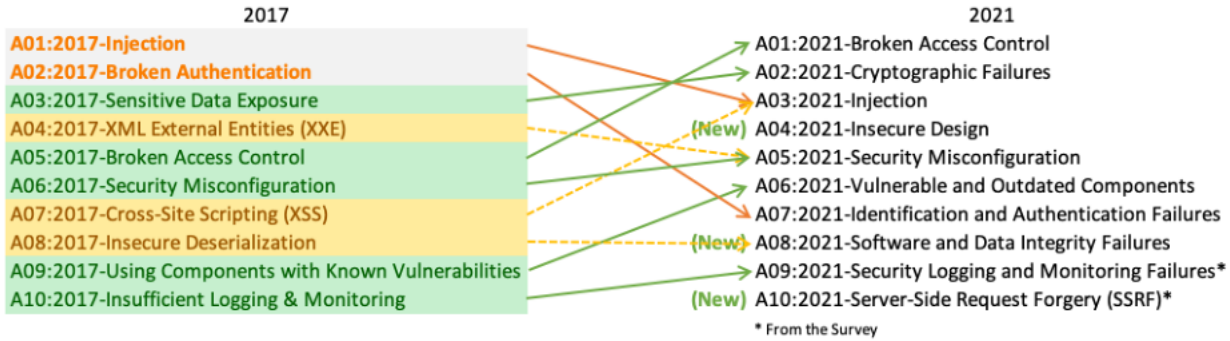
Tackling the OWASP Top Ten

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
```

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

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

deleteArticle :: AccessToken ModifyArticle -> IO ()
deleteArticle token = do
  let (ModifyArticle articleId) = tokenData token
  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)

```java
public class User {
    private final Id<User> id;
    private Name50 firstName, lastName;
    private Birthdate dateOfBirth;
    private PasswordHash passwordHash;
    // Constructor, getters, setters, domain logic, etc.
}
public class UserViewModel {
    private final String fullName;
    private final DateTime dateOfBirth;
    // Constructor, getters, mapping from entity to dto and vice versa
}
public List<UserViewModel> getUsers(...) {
    List<User> users = userService.getUsers(...);
    return users.stream().map(UserViewModel::entityToViewModel).collect(Collectors.toList());
}
```

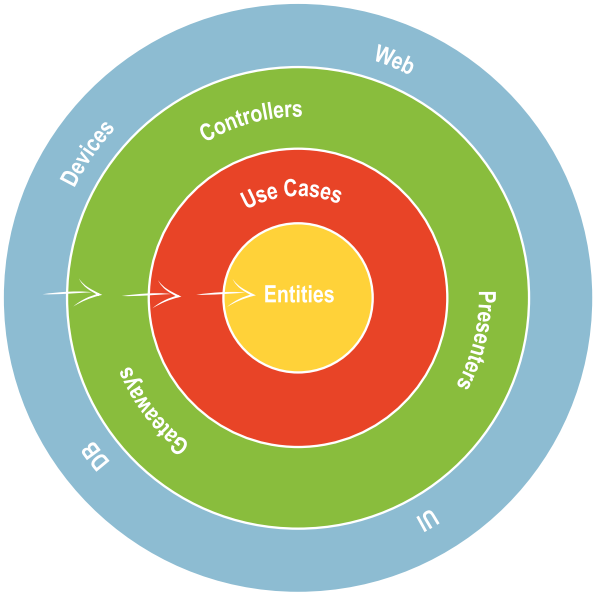
Fighting XSS Attacks (Elm)

```
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>
```

Want to Learn About Trusted Types?

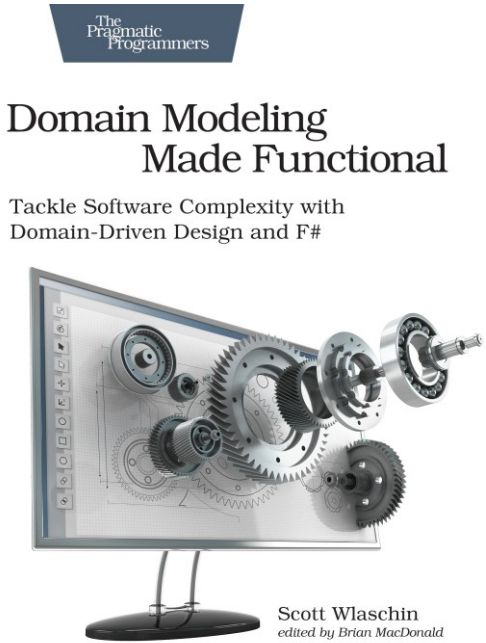
https://www.youtube.com/watch?v=ndk5vFudkMo

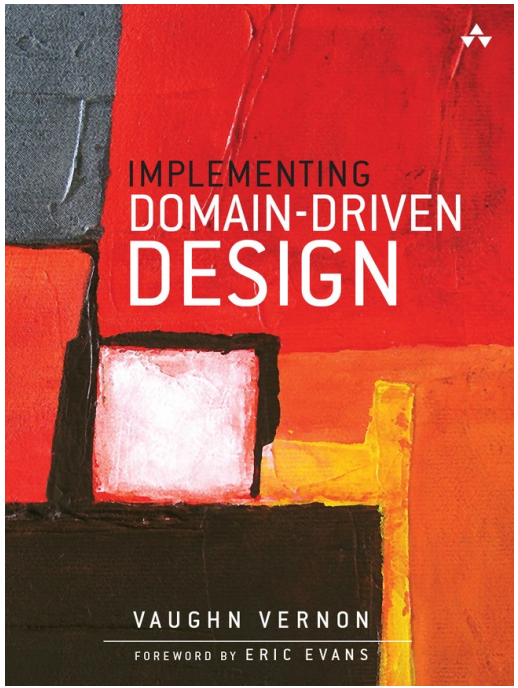
Hexagonal / Onion / Clean Architecture



Source: https://www.4soft.de/blog/2019/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

mkoppmann@sba-research.org

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

Bundesministerium Digitalisierung und Wirtschaftsstandort

FFG Forschung wirkt.

wirtschaftsagentur wien Ein Fonds der Stadt Wien

European Commission

FWF Der Wissenschaftsfonds.

netidee OPEN INNOVATIONS