SINGLE RESPONSIBILITY PRINCIPLE

Robert Bräutigam

MATHEMA Software GmbH.

GOAL

TO DEFINE THE *SRP* IN A PRAGMATICAL* AND OBJECTIVE* WAY, ALLOWING IT TO BE USED DIRECTLY IN OUR DAY-TO-DAY PROGRAMMING.

DEFINITIONS

Each software module should have one and only one responsibility.

"Each software module should have one and only one reason to change"

"Gather together the things that change for the same reasons. Separate those things that change for different reasons."

"Same reason" means it originates from the same business person. *(Really???)*

RATIONALE

- We want maintainability!
- Minimize work for a given amount of change.
- Minimize amount of code to read/write.

So:

- Small classes.
- High probability of localized change.
- Low probability of change propagating to other classes.

SINGLE RESPONSIBILITY PRINCIPLE

==

MAXIMIZE COHESION

MINIMIZE COUPLING

COHESION & COUPLING

- It's about dependencies.
- Dependencies inside an object make it more cohesive.
- Dependencies between objects make them more coupled.

What are dependencies?

- Physical relationships, like a method calling another one, or method referencing a variable.
- Semantic relationships.

PHYSICAL DEPENDENCIES

```
public final class Amount {
    private final int cents;
    private final Currency currency;

    public Amount(int cents, Currency currency) {
        this.cents = cents;
        this.currency = currency;
    }

    public Amount subtract(Amount other) {
        if (currency != other.currency) { ...error...}
        return new Amount(cents - other.cents, currency);
    }

    public boolean isEqualTo(Amount other) {
        return subtract(other).cents == 0;
    }
}
```

This class is *cohesive*.

PHYSICAL DEPENDENCIES

```
public final class Account {
    private Amount balance;
    ...
    public void debit(Amount amount) {
        ...
        balance = balance.subtract(amount);
    }
}
```

Class has *coupling* to `Amount`.

SEMANTIC DEPENDENCIES

```
public final class CurrencyConverter {
    public Amount convert(Amount amount, Currency currency) {
        return new Amount(
            amount.getValue() *
                exchange.getDailyRate(
                    amount.getCurrency(), currency),
            currency);
    }
}
```

Class knows too much about `Amount`. It is semantically coupled. High probability *they change together*.

SEMANTIC COUPLING

- It is always a code smell.
- It is also a design smell.
- Much worse than physical coupling, since it is invisible (to the compiler).
- Changes are very likely to propagate through semantic couplings, sometimes in subtle and unexpected ways.
- Very often facilitated by *getters*.

DESIGNING A UI: METERS

METER

These are *Water* or *Gas* Meters, a part of an IoT network, in which we're writing the server code, with following requirements:

- A meter may receive requests for making a readout of current values.
- A meter may at any time update its capabilities.
- These capabilities have to be displayed on the web interface.

METER DESIGN

```java
public final class Meter {
   private boolean gzipSupported;
   private Encryption encryptionSupported;
   private X509Certificate encryptionCertificate;
   ...

   public void receiveReadoutRequest() {
       ... using capabilities ...
   }

   public void updateCapabilities(...) {
       ... update capabilities ...
   }

   public String displayHtml() {
       return " ... html code ... ";
   }
}
```

COMMON INTERPRETATION OF SRP

```java
public final class Meter {
   private boolean gzipSupported;
   private Encryption encryptionSupported;
   private X509Certificate encryptionCertificate;
   ...
   ...getters, setters...
}

public final class MeterView {
   public String displayMeter(Meter meter) {
      return "...html with meter.getGzipSupported(),
         meter.getEncryptionSupported(), meter.getX()...";
   }
}
```

Heavy semantic and physical coupling, very unmaintainable. Violation of SRP!

MORE PRAGMATIC INTERPRETATION

```java
public final class Meter {
   private boolean gzipSupported;
   private Encryption encryptionSupported;
   private X509Certificate encryptionCertificate;
   ...
   public Component display() {
      return new Tags(
         gzipSupported?new Tag("GZIP"),
         encryptionSupported?new Tag("ENC"),
         encryptionSupported.display(),
         ...);
   }
}
```

No more HTML, no details of design, at the same time `Tags` do not know `Meter`.

UGH, UI IN THE DOMAIN, I FEEL DIRTY!

Maybe that is just a culture of discrimination. Why shouldn't the UI be part of the business?

UI IN THE DOMAIN: CONS

- It's just wrong
- UI is not important!
- I don't want to change business logic because of *colors*.
- What if I want to change the Web UI to Swing?

UI IN THE DOMAIN: PROS

- UI is by it's nature tightly coupled/cohesive to the domain.
- UI is usually an important part of an application.
- UI is actually part of the requirements!
- Business people actually talk about the UI, it is part of the common understanding and vocabulary!
- *Details* of the UI don't have to be in the Domain!

COMPOSITION AND SRP: USER REGISTRATION

USER REGISTRATION

Users for our system with following requirements:

- User may register with username and password.
- User may authenticate herself with given password.
- At the registration an email should be sent as confirmation.

"TRADITIONAL" DESIGN

```
public class UserManager {
   public boolean authenticate(String username,
         String password) {
      String passwordHash = sql.select("from user ...", ...);
      return HashUtils.match(password, passwordHash);
   }

   public void register(String username, String password,
         String emailAddress) {
      sql.insert("into user ...", ...);
      new SmtpClient().send(emailAddress,
         "Hello "+username+", welcome to Application");
   }
}
```

"Don't mix SQL with SMTP"

ADDING "TRADITIONAL" SRP

```java
public class User {
   private String username;
   private String passwordHash;
   private String emailAddress;

   ...getters, setters...
}

public class EmailNotificationService {
   private SmtpClient smtpClient;
   ...
   public void sendNotification(User user) {
      smtpClient.send(user.getEmailAddress(),
         "Hello "+user.getUsername()
         +", welcome to Application");
   }
}
```

ADDING "TRADITIONAL" SRP

```java
public class UserRepository {
    ...
    public void insert(User user) {
        sql.insert("into user ...",
            user.getUsername(),
            HashUtils.hash(user.getPassword()),
            user.getEmailAddress());
    }

    public User select(String username) {
        return sql.select(...);
    }
}
```

This is still too much.

ADDING "TRADITIONAL" SRP

```
public class InsertUserCommand {
    public void execute(User user) {
        sql.insert("into user ...",
            user.getUsername(),
            HashUtils.hash(user.getPassword()),
            user.getEmailAddress());
    }
}

public class SelectUserCommand {
    public User execute(String username) {
        return sql.select(...);
    }
}
```

ADDING "TRADITIONAL" SRP: RESULT

```java
public class UserManager {
   private InsertUserCommand insertUserCommand;
   private SelectUserCommand selectUserCommand;
   private EmailNotificationService notificationService;

   public boolean authenticate(String username,
         String password) {
      User user = selectUserCommand.execute(username);
      return HashUtils.match(password,
         user.getPasswordHash());
   }

   public void register(String username, String password,
         String emailAddress) {
      User user = new User(username, password, emailAddress);
      insertUserCommand.execute(user);
      notificationService.sendNotification(user);
   }
}
```

ADDING "TRADITIONAL" SRP: RESULT

- Familiar to most people
- *Seems* clean enough
- It is the beginning of the end!
- Leads to high fragmentation (one method per class designs).
- Leads to injection/dependency/testing hell.

ADDING "TRADITIONAL" SRP: ANALYSIS

- Instead of *decoupling*, we actually have very tight coupling. Semantic as well as physical.
- No cohesion, or very little cohesion.
- Has very little to do Object-Orientation, this may or may not be a problem for some.
- Look at the vocabulary: `User`, `UserManager`, `InsertUserCommand`, `SelectUserCommand`, `EmailNotificationService`

SIDENOTE: DON'T DO `Utils`

```java
public final class PasswordHash {
    ...
    public static PasswordHash compute(String clearText) {
        byte[] randomSalt = ...;
        return compute(randomSalt, clearText);
    }
    public static PasswordHash compute(byte[] salt,
            String clearText) {
        byte[] calculatedHash = ...;
        return new PasswordHash(salt, calculatedHash);
    }
    public boolean matches(String clearText) {
        return this.equals(new PasswordHash(salt, clearText));
    }
    public boolean equals(Object o) {
        ...
        return Arrays.equals(this.hash, o.hash);
    }
}
```

ALTERNATIVE DESIGN WITH SRP

```
public interface User {
   boolean authenticate(String password);

   void register();
}
```

The "core" is an interface on which all functionalities are implemented on.

ALTERNATIVE DESIGN WITH SRP

```
public final class SqlUser implements User {
    ...

    @Override
    public boolean authenticate(String password) {
        return new PasswordHash(
                sql.select("passwordhash from user...", ...))
            .matches(password);
    }

    @Override
    public void register() {
        sql.insert("into user...", ...);
    }
}
```

Only SQL code, cleanly separated.

ALTERNATIVE DESIGN WITH SRP

```java
public class DelegatingUser implements User {
    private final User delegate;

    public DelegatingUser(User delegate) {
        this.delegate = delegate;
    }

    @Override
    public boolean authenticate(String password) {
        return delegate.authenticate(password);
    }
    @Override
    public void register() {
        delegate.register();
    }
}
```

Unfortunately delegation is not part of the language (like inheritance). Yet.

ALTERNATIVE DESIGN WITH SRP

```
public final class NotifiedUser extends DelegatingUser {
   ...
   public NotifiedUser(String username, String emailAddress,
         User delegate) {
      super(delegate);
      ...
   }

   @Override
   public void register() {
      super.register();
      smtpClient.send(emailAddress,
         "Hello "+username+", welcome to Application");
   }
}
```

ALTERNATIVE DESIGN WITH SRP: RESULT

```
public User createUser(String username, String password,
      String emailAddress) {
   return new NotifiedUser(username, emailAddress,
      new SqlUser(username, password, emailAddress));
}
```

ALTERNATIVE DESIGN WITH SRP: ANALYSIS

- Instead of separating per technology, it separates based on vertical features. Slight but crucial difference.
- Objects are cohesive and truly decoupled. I.e. SRP.
- Each feature is testable.
- It is OO. No data is pulled out of objects.

PRELIMINARY SUMMARY

- I don't think it means what I was thaught it means.
- (If it is, it shouldn't...)
- Single Responsibility Principle == Cohesion & Coupling

So...

WHAT YOU SHOULD DO FOR YOUR CODE:

1. Make sure your class is physically cohesive. Methods and fields refer to each other. Don't forget the Constructor!
2. Make sure the physical coupling is not stronger than the physical cohesion.
3. Make sure you *have* your data, and don't need to *get* your data! This avoids semantic coupling.
4. Don't be dogmatic, UI (HTML, JSON/HTTP, SOAP) is functionality too!

SOME (UNEXPECTED?) CONSEQUENCES

MVC

"TRADITIONAL" MVC

```
public class Person { // Model
   private String name;
   private int age;
   private String address;
   ...setters, getters...
}

public class PersonController {
   public void greet(Person person) { ... }
   public void add(Person person) { ... }
}
```

View is JSF or plain HTML with substitutions

"TRADITIONAL" MVC

- Despite claims that this is for *decoupling*, it actually **increases** coupling.
- Everything is strongly coupled to the data object, both controller and view.
- The whole thing must change if `Person` changes.
- Only allows limited "view" changes to be localized.
- Makes code unmaintainable.

ALTERNATIVE MVC

```
public final class Person { // Model
    private final String name;
    private final int age;
    private final String address;
    ...

    public Component displaySummary() {
        return new InfoPanel()
            .addInfo("Name", name)
            .addInfo("Age", age)
            .addInfo("Address", address);
    }
}
```

- Model = The "business" object itself
- Controller = Abstract UI Component
- View = HTML the Component reads

ALTERNATIVE MVC WITH INPUT

```java
public final class Person { // Model
    private final String name;
    private final int age;
    private final String address;

    public Person(String name, int age, String address) {
        ...
    }
    ...
    public InputComponent<Person> displayInput() {
        return new InputGroup()
            .add(new TextInput("Name", name))
            .add(new NumberInput("Age", age, ...))
            .add(new TextInput("Address", address))
            .map(Person::new);
    }
}
```

This is what we should mean by cohesion!

ALTERNATIVE MVC

- Responsibilities cleanly separated. `Person` doesn't know details of `View`, `View` doesn't know details of `Person`.
- Composable!

ALTERNATIVE MVC: COMPOSABILITY

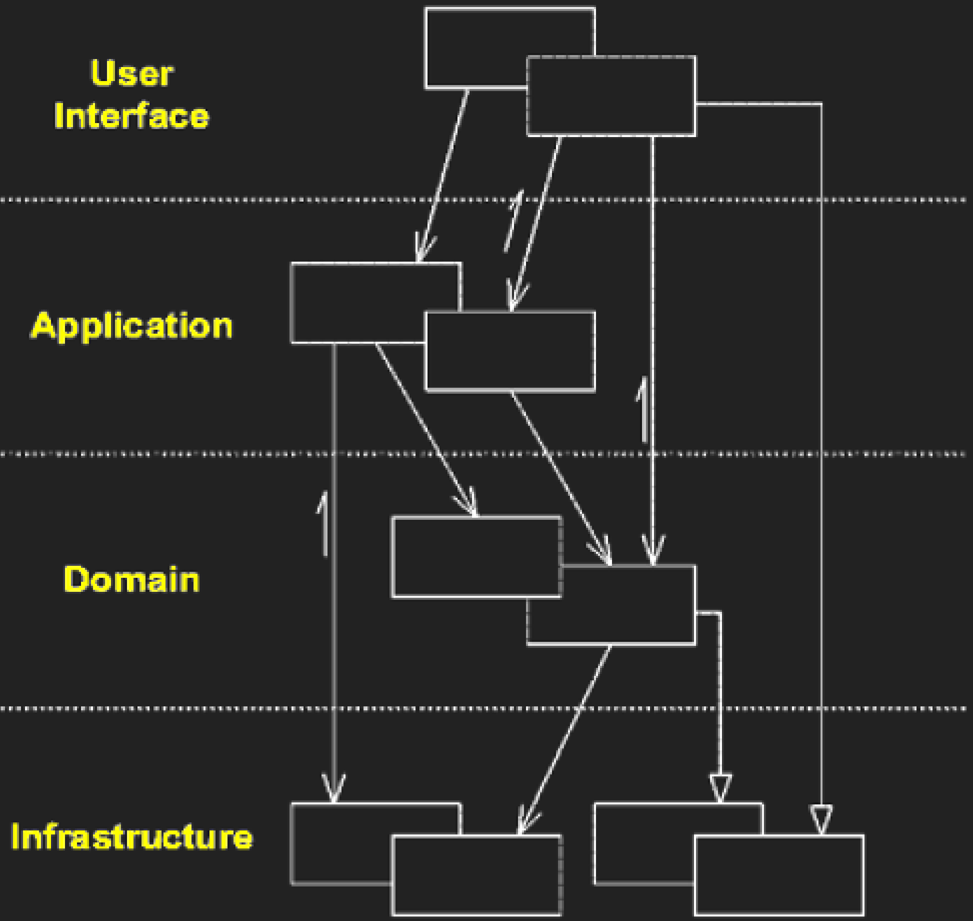
```java
public final class Person { // Model
   private final String name;
   private final int age;
   private final Address address;

   public Person(String name, int age, Address address) {
      ...
   }
   ...
   public InputComponent<Person> displayInput() {
      return new InputGroup()
         .add(new TextInput("Name", name))
         .add(new NumberInput("Age", age, ...))
         .add(address.displayInput())
         .map(Person::new);
   }
}
```

SOME (UNEXPECTED?) CONSEQUENCES

LAYERED ARCHITECTURES

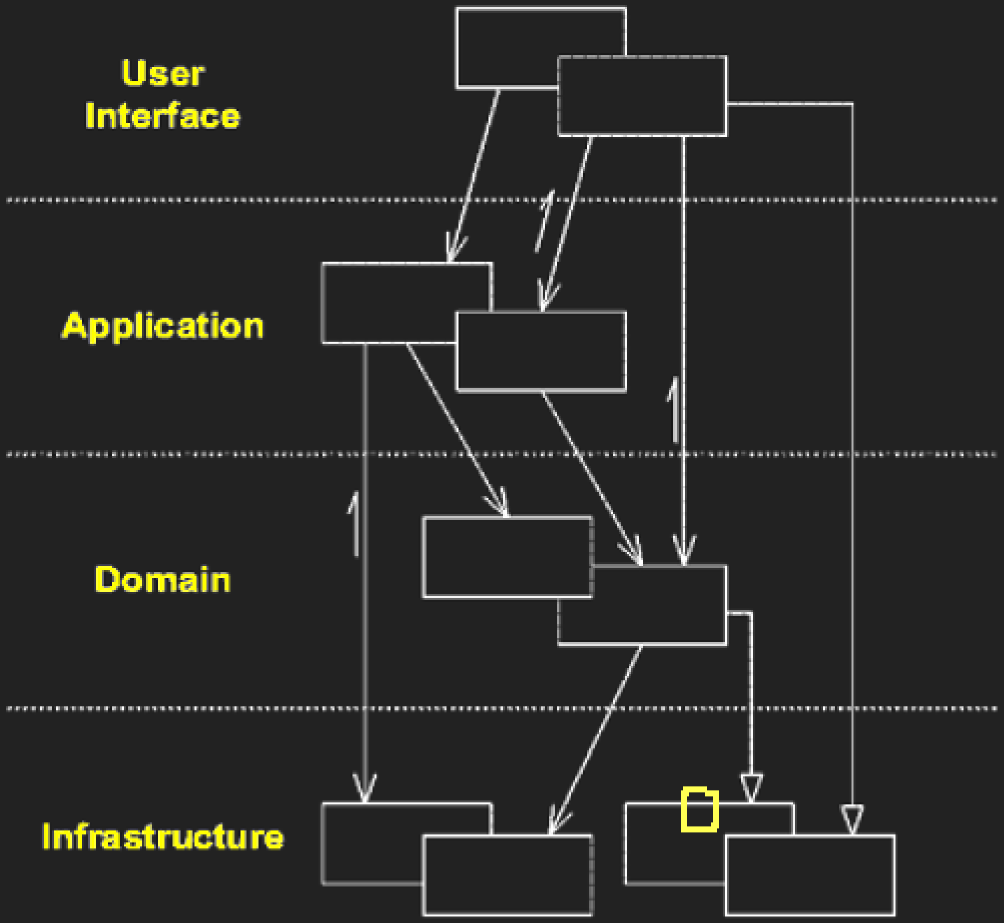
LAYERED ARCHITECTURE



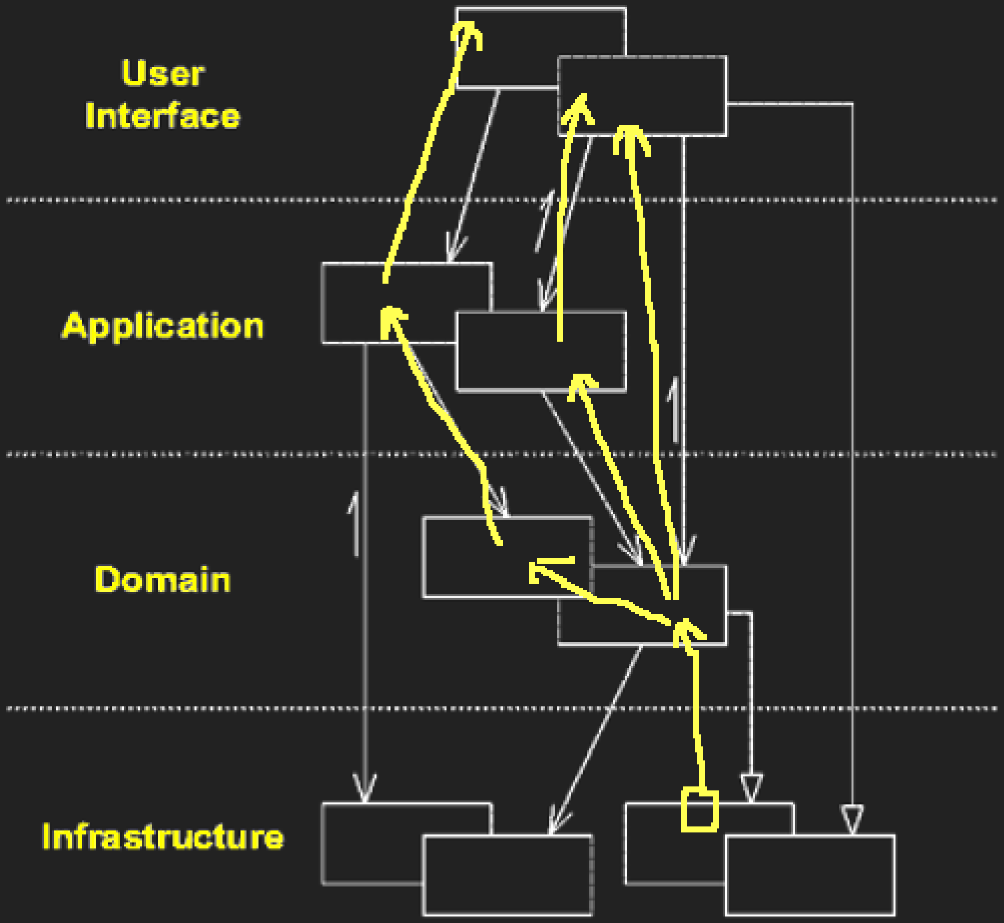
PROBLEMS WITH LAYERS

- The "Domain" is only 1/4 of the Application
- Is a technical design, business-agnostic.
- Layers usually leak data upwards and create coupling (DTOs)
- Layers therefore tightly coupled to layers below.
- ⇒ Changes escalate and expand upwards!

LAYERED ARCHITECTURE



LAYERED ARCHITECTURE

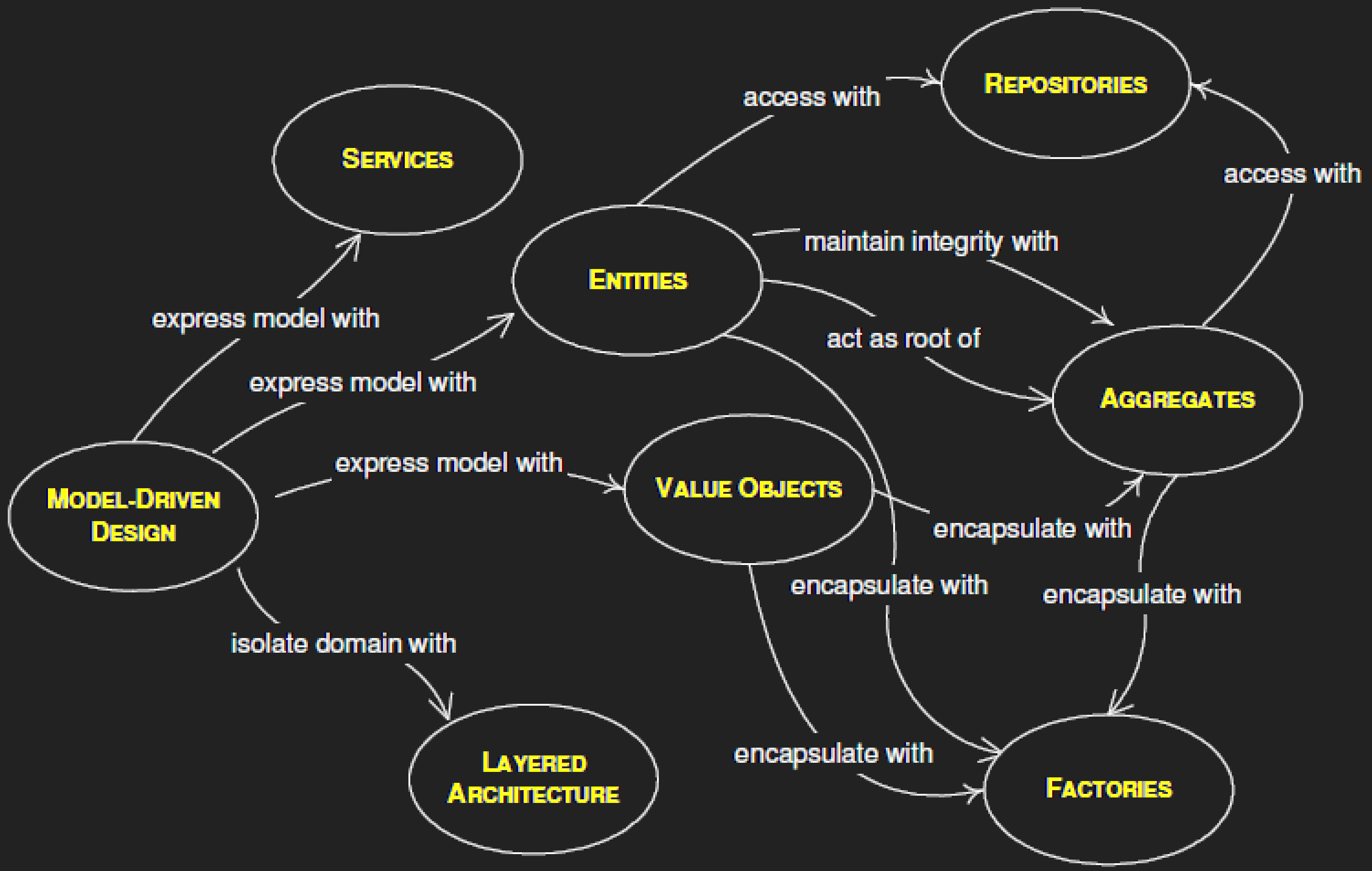


LAYERED ARCHITECTURE

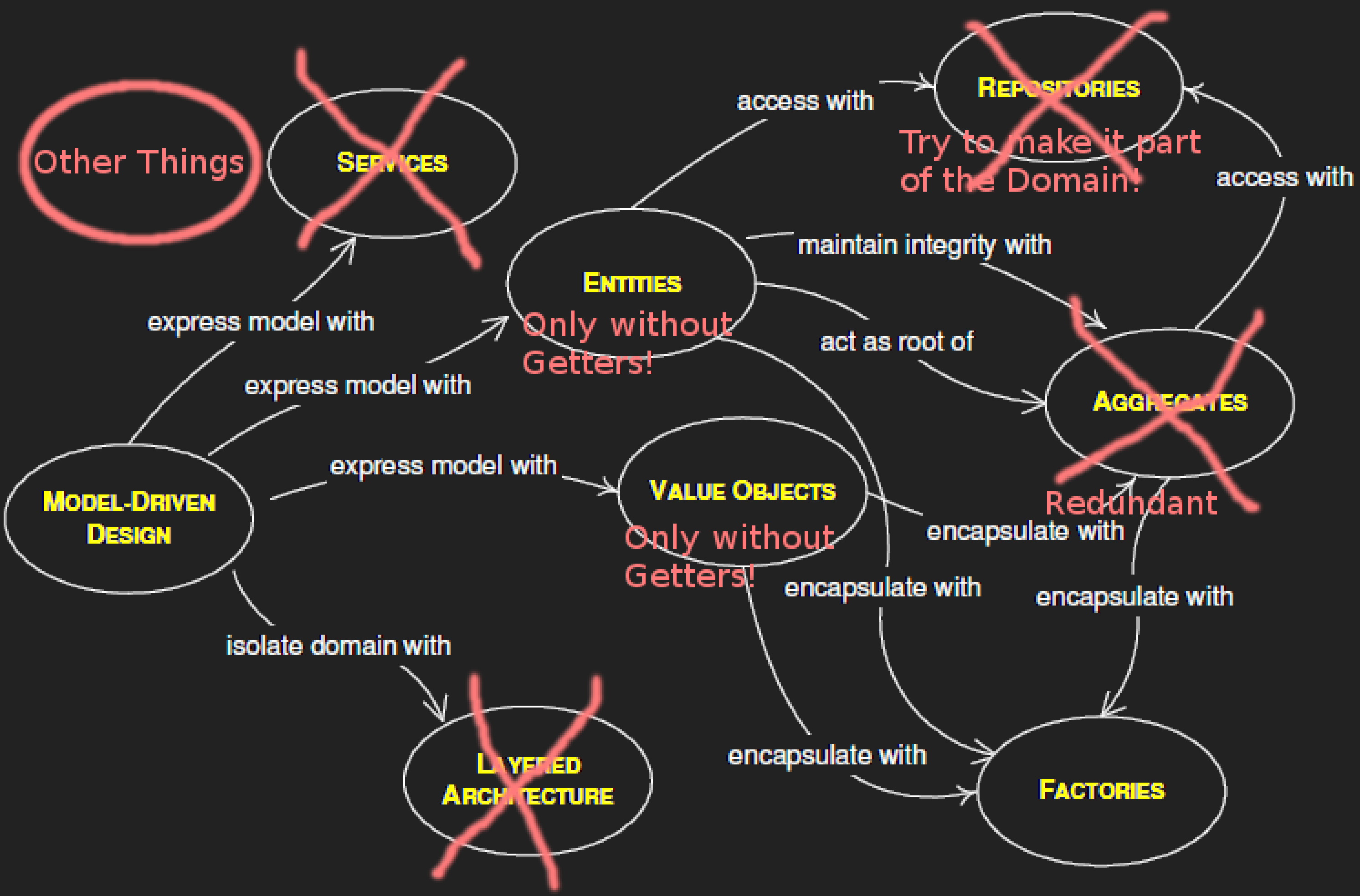
- How many classes would you have to change if a domain object changed slightly? *Is it more than 1?*
- Almost automatically violates *SRP*. Changes are not localized.

DOMAIN-DRIVEN DESIGN + SRP

BUILDING BLOCKS OF DDD



BUILDING BLOCKS OF DDD



DDD + SRP

- DDD makes the case for the ubiquitous language.
- The same language in code as between people.
- ⇒ Responsibilities can not be arbitrary, have to come from the business as well.
- Among other things: *Persistence*, *Json/XML Formatting*, *Validation*, *Rules*, *Commands*, etc., are therefore usually not valid responsibilities.

SUMMARY

- We got a *useful* definition of *SRP*, based on Cohesion and Coupling
- DDD implies responsibilities are not arbitrary, have to come from the requirements.
- Using *SRP* to increase Maintainability leads to a different design than most are familiar with.
- Among others: *UI*, *MVC* and *Layered Architectures* have superior alternative interpretations for most cases.

THANKS

QUESTIONS?

robert@mathema.de

https://javadevguy.wordpress.com/

@robertbrautigam