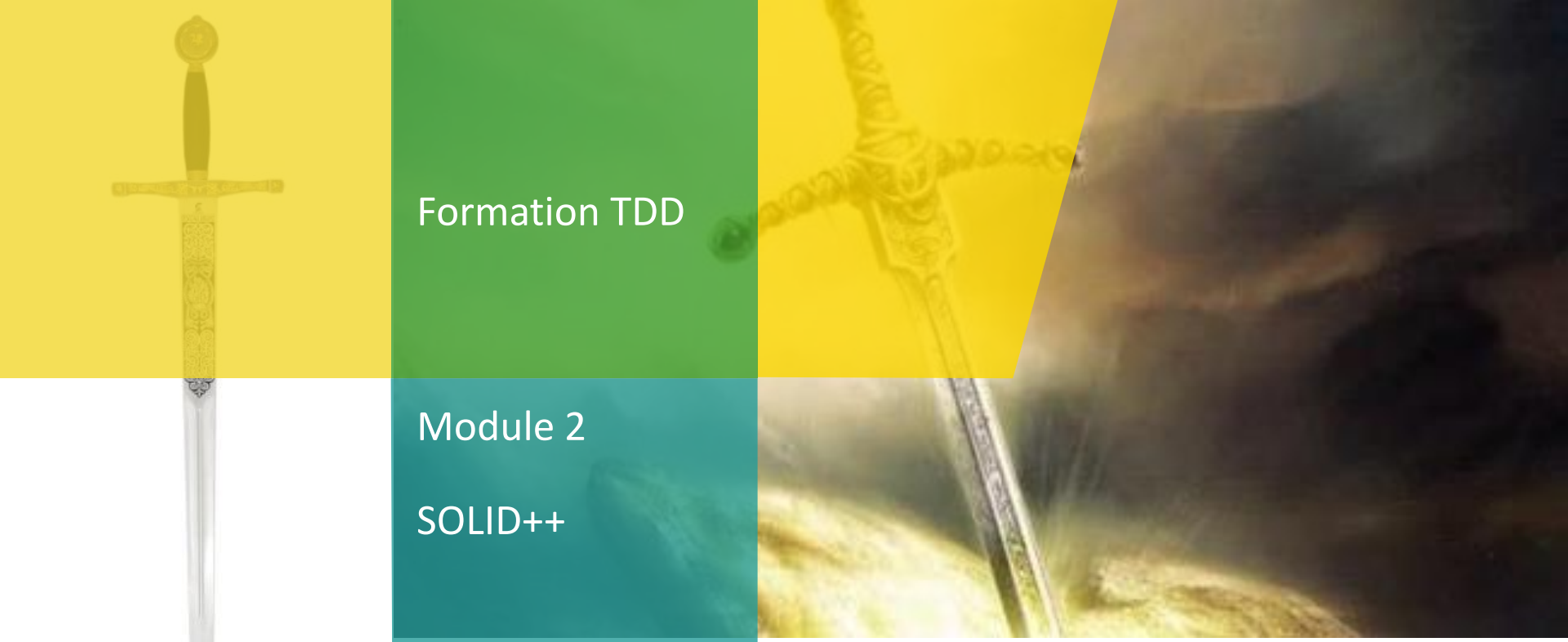

Formation TDD

Module 2

SOLID++

Trianon SA

2021

TRIANON
Compensation
& Benefits Management

Introduction

Origine and definition

- First introduced by Robert C. Martin (a.k.a. Uncle Bob) in his 2000 paper "*Design Principles and Design Patterns*"
- Acronym introduced later by Michael Feathers in 2004
- Mnemonic device for 5 design principles of **object-oriented programs**
  - **S**ingle-responsibility principle
  - **O**pen-closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle
  - **D**ependency inversion principle

TRIANON

Goals

Advantages and benefits

- Helps produce readable, adaptable, and scalable code
- Helps eliminate design smells
- Focuses on maintainability, testability, flexibility and scalability
- Adopted in both agile development and adaptive software development

“ Write better Oriented Object code ”

TRIANON

SOLID++

Single-responsibility principle

- Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices*, 2003

"A class should have only one reason to change."

- Every module, class or function should have responsibility over a single part the program functionality, and should fully encapsulate that part
- A responsibility is defined as a reason to change, meaning that a module, class or function should have one, and only one, reason to be changed
- For example, a class should contain only variables and methods relevant to its functionality
- Relates to cohesion and robustness
- At an architecture level, can be applied using layers
- In DDD, can be applied using contexts, entites, value objects, business rules, …

SOLID++

Single-responsibility principle

```
public class RegisterService
{
    public void RegisterUser(string username)
    {
        if (username == "admin")
            throw new InvalidOperationException();

        SqlConnection connection = new SqlConnection();
        connection.Open();
        SqlCommand command = new SqlCommand("INSERT INTO [...]");

        SmtpClient client = new SmtpClient("smtp.myhost.com");
        client.Send(new MailMessage());
    }
}
```

SOLID++

Single-responsibility principle

```
public void RegisterUser(string username)
{
    if (username == "admin")
        throw new InvalidOperationException();

    _userRepository.Insert(...);

    _emailService.Send(...);
}
```

SOLID++

Open–closed principle

- Bertrand Meyer in *Object-Oriented Software ConstructionDomain Driven Design*, 1988

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

- An entity can allow its behaviour to be extended without modifying its source code
- Minimizes risks when you add new features
- Different approaches, not always practical

TRIANON

SOLID++

Open–closed principle

- A module will be said to be open if it is still available for extension
  - For example, it should be possible to add fields to the data structures it contains or new elements to the set of functions it performs
- A module will be said to be closed if is available for use by other modules
  - The interface of the module has been given a well-defined, stable description
- For classes, specializations are added through inheritance and polymorphism
  - The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface

SOLID++

Open–closed principle

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width * Height;
    }
}
```

SOLID++

Open–closed principle

```
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius * Radius * Math.PI;
    }
}
```

SOLID++

Liskov substitution principle

- Introduced by Barbara Liskov in 1988 and formalized later with Jeannette Wing in 1994

"Derived class objects must be substitutable for the base class objects. That means objects of the derived class must behave in a manner consistent with the promises made in the base class contract."

- LSP can also be described as a counter-example of Duck Test

"If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction ."

SOLID++

Liskov substitution principle

- Any class must be directly replaceable by any of its subclasses without error
  - Each subclass must maintain all **behavior** from the base class along with any new behaviors unique to the subclass
  - The child class must be able to process all the same requests and complete all the same tasks as its parent class
- Imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages (contravariance and covariance)
- Some critics argue that this principle is not consistent with all program types since abstract supertypes that have no implementation cannot be replaced by subclasses

SOLID++

Liskov substitution principle

```
public interface IDuck
{
    void Swim();
    // contract says that IsSwimming should be true if Swim has been called.
    bool IsSwimming { get; }
}

public class OrganicDuck : IDuck
{
    public void Swim()
    {
        //do something to swim
    }

    bool IsSwimming { get { /* return if the duck is swimming */ } }
}
```

SOLID++

Liskov substitution principle

```
public class ElectricDuck : IDuck
{
    bool _isSwimming;

    public void Swim()
    {
        if (!IsTurnedOn)
            throw new Exception();

        _isSwimming = true;
        //swim logic
    }

    bool IsSwimming { get { return _isSwimming; } }
}
```

SOLID++

Interface segregation principle

"Clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of functions, each one serving one submodule."

- Splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them
- Intended to keep a system decoupled and thus easier to refactor, change, and redeploy
- An interface should be more closely related to the code that uses it than code that implements it
  - Methods on the interface are defined by which methods the client code needs rather than which methods the class implements
  - Clients should not be forced to depend upon interfaces that they don't use

SOLID++

Interface segregation principle

```
interface AtmInterface
{
    void Deposit(int money);
    void Withdraw(int money);
}
```

SOLID++

Dependency inversion principle

- Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices*, 2003

"High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces)."

"Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions."

- Depending on a concept instead of on an implementation reduces the need for change at call sites
- Dependency Inversion Principle is primarily about reversing the conventional direction of dependencies from "higher level" components to "lower level"

SOLID++

Dependency inversion principle

```
public interface IAssureRepository : IRepository<Assure>
{
    Task AddAsync(IEnumerable<Assure> effectif, CancellationToken cancellationToken);
    Task<List<Assure>> GetByRepriseIdAsync(RepriseId repriseId, CancellationToken cancellationToken);
}
```

SOLID++

Balanced Abstraction

"All code constructs grouped by a higher-level construct should be at the same level of abstraction."

- All instructions inside a method should be at the same level of abstraction
- All public methods inside a class should be at the same level of abstraction
- All classes should be inside a package/namespace
- All sibling packages/namespace should be inside a parent package/namespace

SOLID++

Balanced Abstraction

```
public class Car
{
    public int CurrentMileage() { ...}
    public void TravelTo(Location location) { ...}
    public void Save() { ...}
}
```

SOLID++

Least Astonishment

- Introduced by Mike Cowlishaw in *The design of the REXX language*, 1984

"If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature."

- And by Frans Kaashoek in *Principles of computer system design: an introduction*, 2009

"People are part of the system. The design should match the user's experience, expectations, and mental models."

- A component of a system should behave in a way that most users will expect it to behave

SOLID++

Least Astonishment

```
int Multiply(int a, int b)
{
    return a + b;
}
```