Software Architecture and Techniques

Architecture Of Components
And Subsystems

Truths (1/2)

Architecture is a **hypothesis**, that needs to be proven by **implementation** and **measurement**.
- Tom Gilb

The only way to go fast, is to go well.
- Robert C. Martin

Attitude and aptitudes – you can always learn the latter, seldom the former
- Marcel Baumann

Truths (2/2)

The goal of software architecture is to **minimize** the human resources required to build and maintain the required system.
- *Robert Martin*

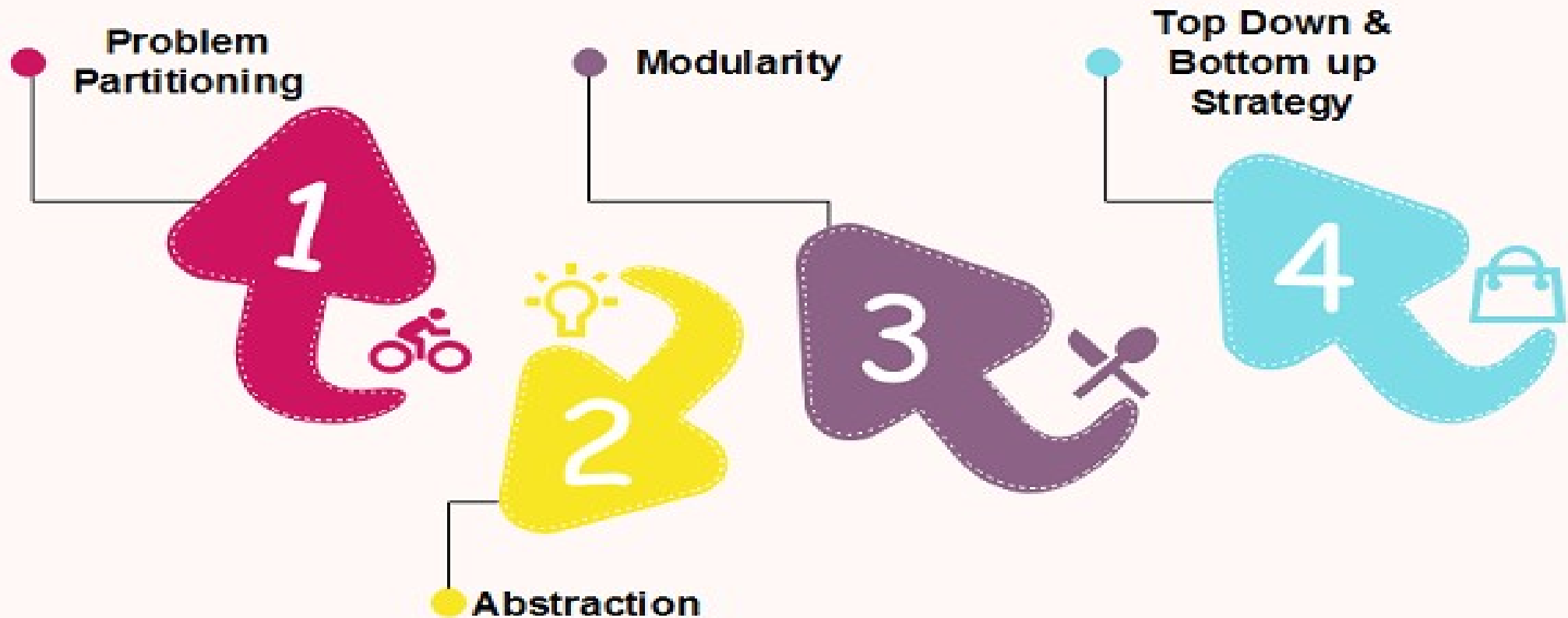Big design up front is dumb, but doing no design up front is even dumber.
- *Dave Thomas*

Approaches

- The system you are building already have siblings

- Open source solutions and articles give you access to huge amount of information

- Copy, mutate, improve
  - Avoid Not Invented Here Syndrome *NIH*

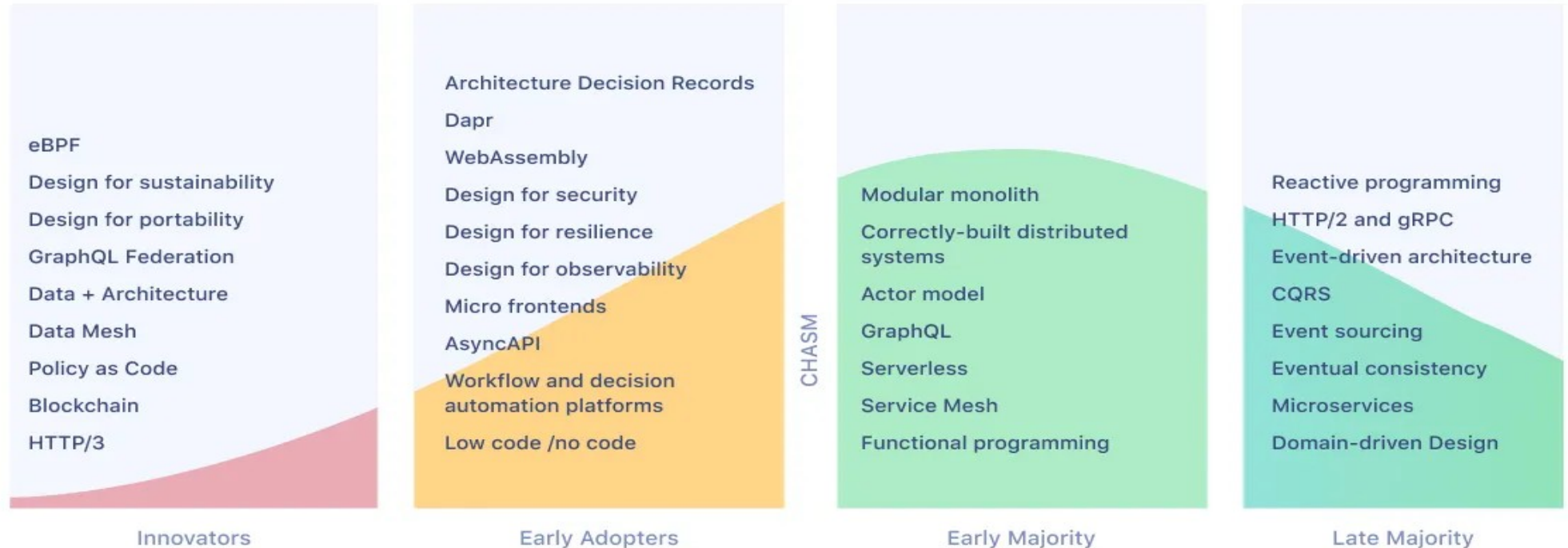# Software Design Principles

Design Approaches

- Divide and Conquer
- Increase Cohesion
- Reduce Coupling
- Increase Abstraction
- Increase Reusability

- Design for Flexibility
- Anticipate Obsolescence
- Design for Portability
- Design for Testability
- Design Defensively

Design Trends

Software Development
Architecture and Design 2022 Graph
http://infoq.link/architecture-trends-2022

InfoQ

Innovators
- eBPF
- Design for sustainability
- Design for portability
- GraphQL Federation
- Data + Architecture
- Data Mesh
- Policy as Code
- Blockchain
- HTTP/3

Early Adopters
- Architecture Decision Records
- Dapr
- WebAssembly
- Design for security
- Design for resilience
- Design for observability
- Micro frontends
- AsyncAPI
- Workflow and decision automation platforms
- Low code /no code

CHASM

Early Majority
- Modular monolith
- Correctly-built distributed systems
- Actor model
- GraphQL
- Serverless
- Service Mesh
- Functional programming

Late Majority
- Reactive programming
- HTTP/2 and gRPC
- Event-driven architecture
- CQRS
- Event sourcing
- Eventual consistency
- Microservices
- Domain-driven Design

SOLID

- S – Single responsibility principle
  high cohesion, only one reason to change

- O – Open/close principle
  open for extension, closed for change

- L – Liskov substitution principle
  *subclasses fulfill superclasses or interfaces role,
  see covariance and contra-variance*

- I – Interface segregation principle
  clients should not be forced to depend on features they do not use

- D – Dependency inversion principle
  *high-level classes should not depend upon low-level classes,
  both should depend on abstraction*

DRY

- Do not Repeat Yourself

- This principle states that each small pieces of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code.

KISS

- Keep it Simple, Stupid!

- This principle states that try to keep each small piece of software simple and unnecessary complexity should be avoided. This helps us to write easy maintainable code.

YAGNI

- You ain't gonna need it

- This principle states that always implement things when you actually need them never implements things before you need them.

Patterns

Creational → blue
Structural → sand
Behavioral → green

THE 23 GANG OF FOUR DESIGN PATTERNS

C — Abstract Factory	S — Facade	S — Proxy
S — Adapter	C — Factory Method	B — Observer
S — Bridge	S — Flyweight	C — Singleton
C — Builder	B — Interpreter	B — State
B — Chain of Responsibility	B — Iterator	B — Strategy
B — Command	B — Mediator	B — Template Method
S — Composite	B — Memento	B — Visitor
S — Decorator	C — Prototype	

Patterns

- Patterns tell **stories** of repeatedly **successful** engineering

- Honest pattern descriptions tell you the **drawbacks** as well as the **benefits**

- Applying patterns is **never** mechanical

- Patterns allow more **conscious** and efficient engineering by discussing **alternatives**

- Patterns give a **common vocabulary** which makes communication about design more efficient

Builder Pattern Example (1/3)

```java
static Letter of(String returnAddress, String insideAddress,
    LocalDate date, String salutation, String body, String closing) {
        return new Letter(returnAddress, insideAddress, date, salutation,
            body, closing);
}
```

Builder Pattern Example (2/3)

```java
// All interfaces are SAM — Single Abstract Method

public static class Builder {
    public static ReturnAddress builder() {
        return returnAddress -> insideAddress -> dateOfLetter -> salutation -> body -> closing ->
            new Letter(returnAddress, insideAddress, dateOfLetter, salutation, body, closing);
    }

    public interface ReturnAddress {
        InsideAddress withReturnAddress(String returnAddress);
    }

    public interface InsideAddress {
        DateOfLetter withInsideAddress(String insideAddress);
    }

    public interface DateOfLetter {
        Salutation withDateOfLetter(LocalDate dateOfLetter);
    }

    public interface Salutation {
        Body withSalutation(String salutation);
    }

    public interface Body {
        Closing withBody(String body);
    }

    public interface Closing {
        Letter withClosing(String closing);
    }
}
```

Builder Pattern Example (3/3)

```java
// create a letter with the traditional factory method builder

var letter = Letter.of(returnAddress, insideAddress, date, salutation,
                       body, closing);
```

```java
// create a letter with the functional builder

var letter = Letter.Builder.builder().
        .withReturnAddress(returnAddress)
        .withInsideAddress(insideAddress)
        .withDateOfLetter(date)
        .withSalutation(salutation)
        .withBody(body)
        .withClosing(closing);
```

Layered Architecture

This point is somewhat redundant and maybe theoretical but is worth mentioning. The Layered Architecture breaks almost all rules and idioms of object-orientation. Here are just a few:

- **Encapsulation**: Encapsulation does not survive crossing layers, because the interfaces between layers are defined in terms of data.

- **Abstraction**: There is very little to no abstraction because every layer has to understand all concepts nearly equally.

- **Cohesion and Coupling**: Cohesive parts of the same "thing" are broken up because of the potentially differing technologies involved. So it makes the code less cohesive and more coupled.

- **Law of Demeter**: Access to data, using DTOs, for example, almost always leads to violations.

- **Tell don't ask**: Objects don't get told what to do in the Layered Architecture; they are asked for data, and then, things happen with that data somewhere else out of the control of the object producing or holding the data.
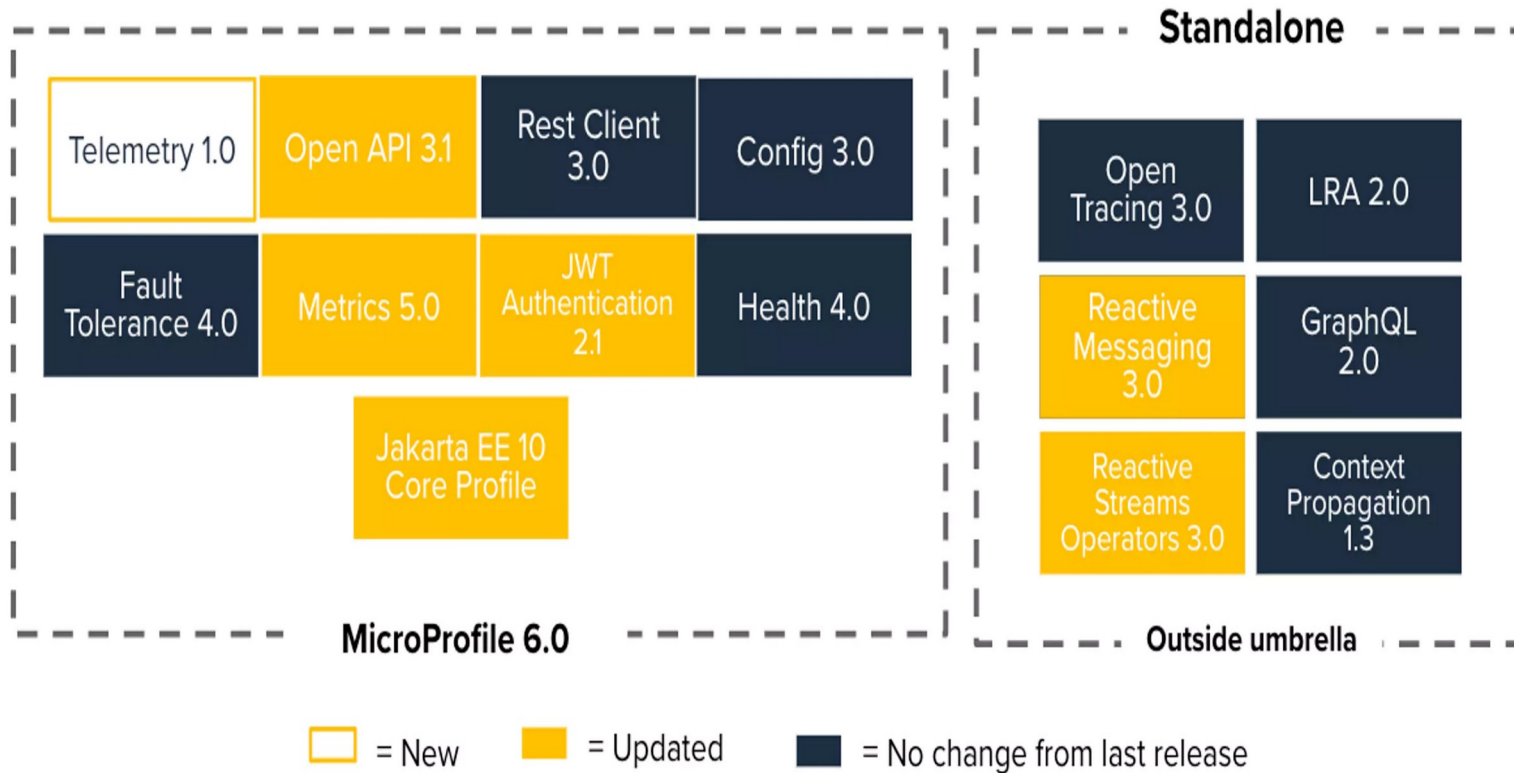
Hexagon Architecture

- It promotes mocking of connectors
  - Improves testability
  - Simplify integration
- It promotes domain models
- It could promote event based approaches

Java Ecosystem (1/3)

- Patterns in Java

- Streams in Java
  - e.g. filtering, composing of collectors

- Functional programming
  - e.g. strategy pattern, function composing, complex predicate expressions

- Reactive programming

Java Ecosystem (2/3)

MicroProfile 6.0

Telemetry 1.0	Open API 3.1	Rest Client 3.0	Config 3.0
Fault Tolerance 4.0	Metrics 5.0	JWT Authentication 2.1	Health 4.0
	Jakarta EE 10 Core Profile		

Standalone

Open Tracing 3.0	LRA 2.0
Reactive Messaging 3.0	GraphQL 2.0
Reactive Streams Operators 3.0	Context Propagation 1.3

Outside umbrella

□ = New   ▮ = Updated   ▮ = No change from last release

SLF4J
Simple Logging
Facade for Java

LOGBack™
The Generic, Reliable
Fast & Flexible
Logging Framework

Java Ecosystem (3/3)

- Exception Handling – prefer runtime exceptions -
- Multi-Threading *java.util.concurrent*
- Patterns in API
- Immutability in API (*see also **record***)
- Java Trends
 - functional programming, immutability, reification, memory-efficiency, heterogeneous processors, fibers

Clean Code

- You shall only produce clean code

- If you inherit dirty components, you have to clean them

  - *Boy scout rule*: Each time you change a code segment, leave it cleaner than you found it

  - It is similar to improve hot code – meaning most valuable or most updated -
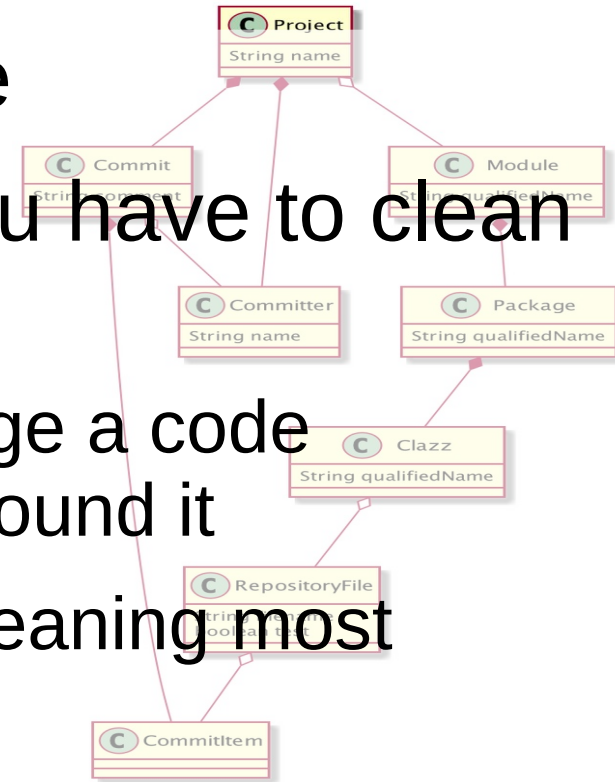
Clean Code Examples

- Remove *java.util.Date*, use *java.time.LocalDate*
- Remove *java.io.File*, use *java.nio.file.Path*
- Use *Stream.toList()*
- Use try with resources
- Remove checked exception, use runtime exception
- Remove XML, use JSON
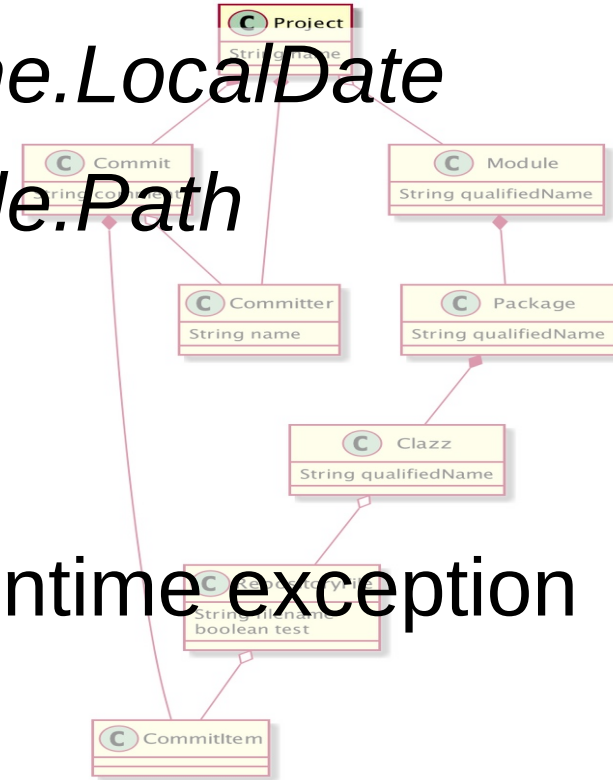
Refactor

- Aggressively refactor your code
- Aggressively refactor your design
- Remember the cone of uncertainty
- Developing a product means learning
- Agile means improving

OOP Anti-Pattern Examples (1/2)

- Singletons are **evil**

- Never return a **null value**

- Returning modifiable collections is evil

- Anemic domain classes are worthless

- DTO *Data Transfer Objects* are waste

OOP Anti-Pattern Examples (2/2)

- Class casting is an object-oriented design error
  - *instanceof* operator is a crime (*see pattern matching for reasonable use of instanceof in Java 14+)*
- Public static methods are often suspect
- Abuse of utility classes is procedural design
- God classes shall be forbidden

Links

- Blog *Agile Component Design*

- Patterns used in Java API (Stackoverflow article) and in a blog article

  All the patterns you are using daily
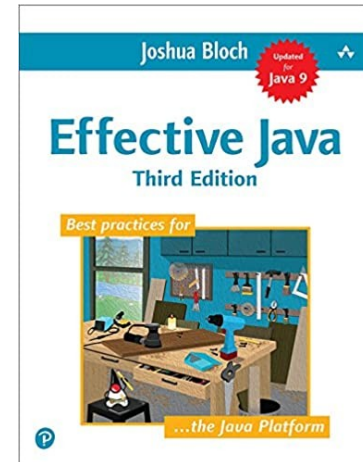
Exercises (1/2)

- Analyse your Java packages and refactor them to fulfill SOLID

- Analyse your Java packages and identify the used patterns

- Can you improve your code with Java idioms and patterns?

- How do you handle errors and exceptions?

Exercises (2/2)

- Read the optional paper on Java Patterns
  - Understand Builder, Facade, Strategy, Factory method patterns and how to use lambda expressions to implement them
  - *RAII* pattern *Resource Acquisition Is Initialization* and Java try with resources
  - Iterator pattern as implemented in Java
- Coding Dojo - Code examples of students
  - Replace custom methods and classes with standard API methods and classes
  - Implement a Java Pattern