Web Application Architecture

Jakub Daněk (Yoso Czech s.r.o.)

http://danekja.org (GitHub & LinkedIn links)
http://yoso.fi (yep, in Finnish)

Motivation

- **SW development project contains many unknowns from start**
  - *"… with proper design, the features come cheaply. This approach is arduous, but continues to succeed."*
    - Dennis Ritchie
- **Many developers made this experience**
  - → best practices, paths to follow

Contents

- **Elementary design principles**
  - Code organization
    - Single responsibility, Separation of concerns, Encapsulation, DRY
  - Dependencies and their expression
    - Inversion of control, Dependency injection
- **Web Application Architectures**
  - Monolithic application, microservices
  - Application layers, common patterns
  - Cross-cutting concerns, Aspect-oriented programming

Elementary Design Principles

Separation of Concerns

- **Code dealing with different work should be kept separate**

- **Increases**
  - Testability
  - Reusability
  - Readability

Example: for user registration, implementation of input validation is irrelevant

Example files: soc_bad.example, soc_good.example

Same principle applies on class, package, module, universe level!

Single Responsibility Principle

- **Specifically for object-oriented design:**
  A class should have one, and only one, reason to change.

  - If class has multiple responsibilities
    - → each responsibility has own specifications
    - → change of one responsibility specification (= reason to change the class) may impact ability of the class to fulfil the others
      - → fragile design

Src: Robert C. Martin, The Single Responsibility Principle

Single Responsibility Principle

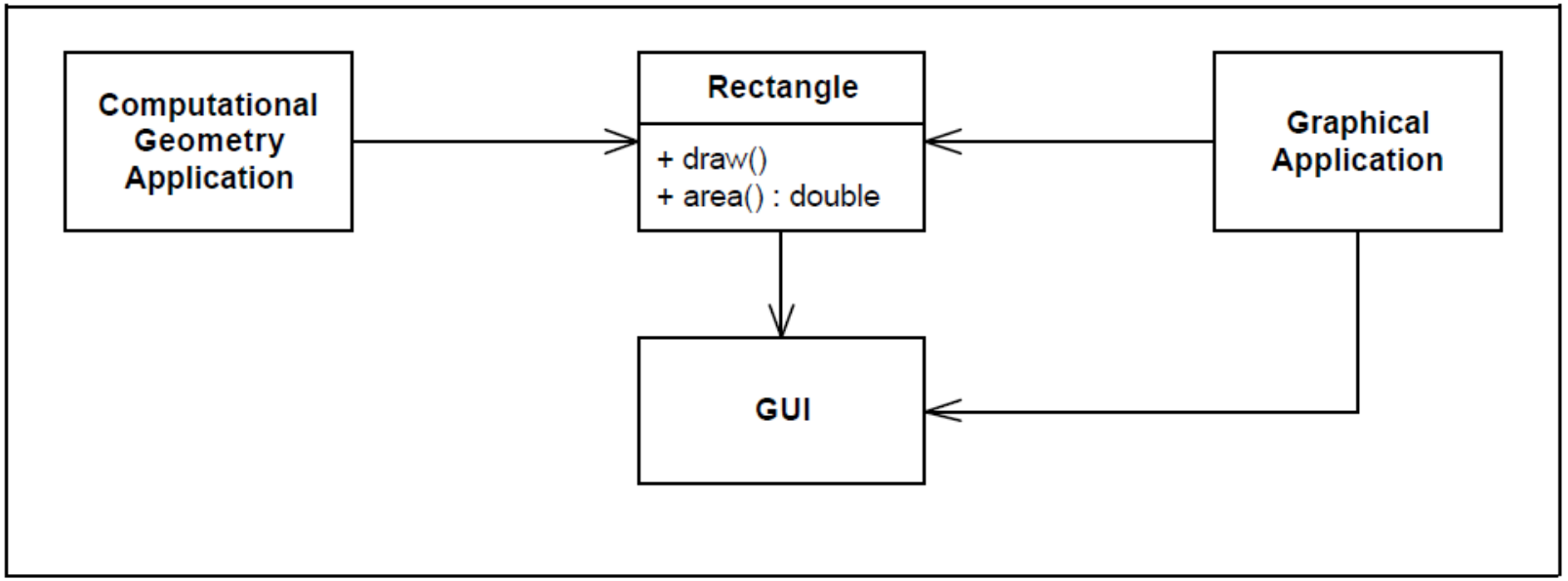


Figure 8-1
More than one responsibility

Src: Robert C. Martin, The Single Responsibility Principle

Single Responsibility Principle

Problems with this design:

- Computational application depends on GUI library through Rectangle
- If Graphical application requires Rectangle to change, it may force rebuild (test, deployment) of the Computational application

Src: Robert C. Martin, The Single Responsibility Principle

Single Responsibility Principle

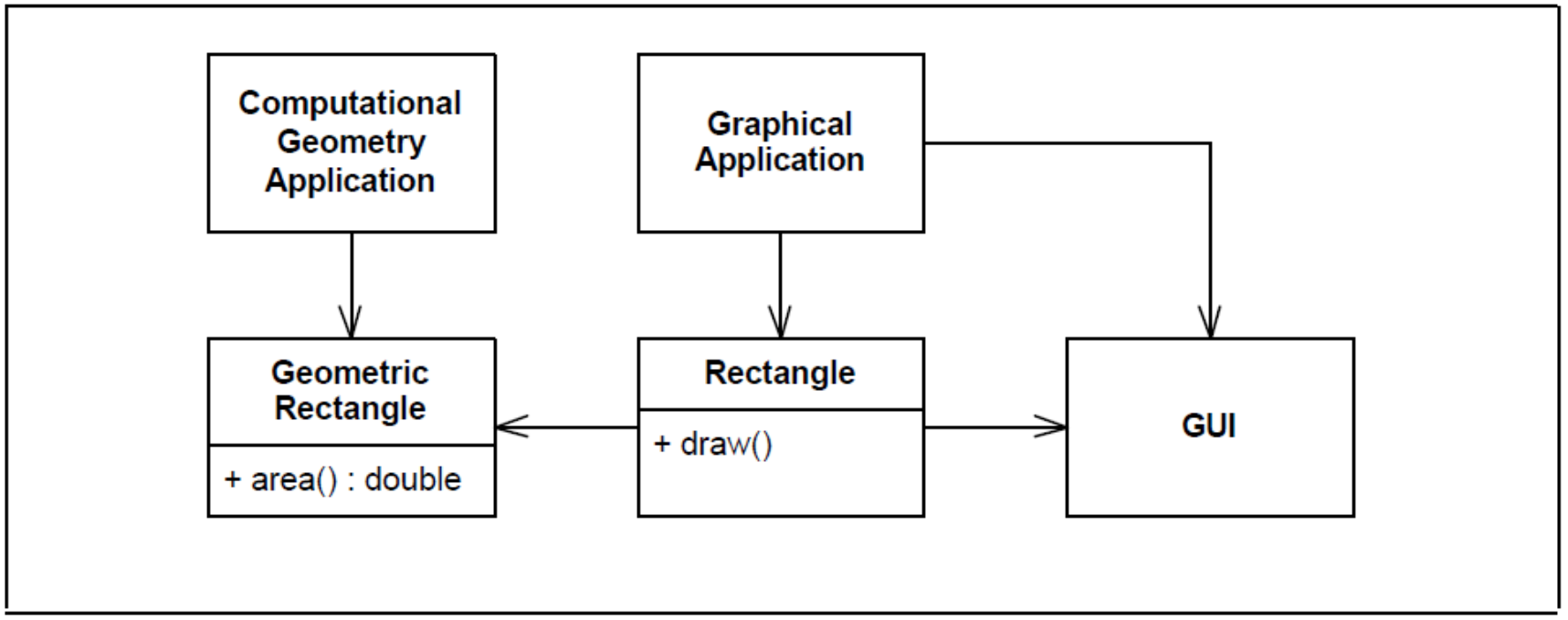


Figure 8-2
Separated Responsibilities

Src: Robert C. Martin, The Single Responsibility Principle

Encapsulation

- **Prevent direct access to internal state of object/module**
  - Note encapsulation is not an OOD-related principle only
- **State read/change should happen only through well-defined public interfaces**
  - Class/module owns its state and should control how it is changed
  - Auto-generating getters and setters breaks encapsulation quite often!

Examples: enc_bad.example, enc_good.example

Don´t Repeat Yourself (DRY)

DRY says that every piece of system knowledge should have one authoritative, unambiguous representation.
Dave Thomas

- **System knowledge consists of**
  - Code, database schemas
  - Test cases and scenarios
  - Documentation
  - …

Src: https://www.artima.com/intv/dry.html

Don´t Repeat Yourself (DRY)

- **Multiple representations of single piece of information**
  - → you have to update them all when changes happen
  - → awful lot of work to do
  - → or they end up out of sync with each other

Src: https://www.artima.com/intv/dry.html

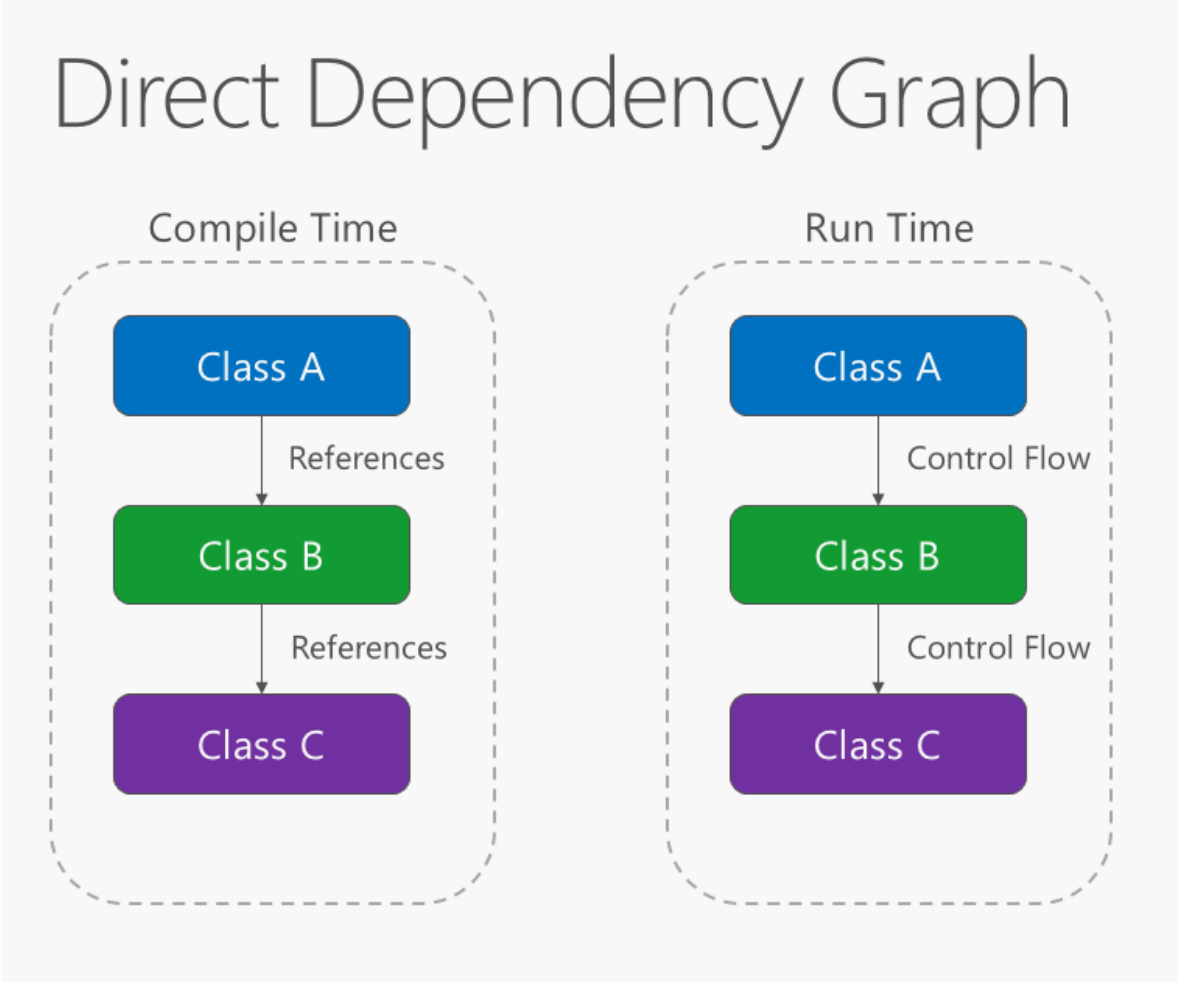
Don´t Repeat Yourself (DRY)

Authoritative representation = change only that and generate the other required representations automatically, if needed

- Example: Database schema
  - Authoritative representation: UML data model
  - Derivative representations generated from the UML model: SQL DDL scripts
  - *Example files: dra_model.pdf, dra_ddl.sql*

Src: https://www.artima.com/intv/dry.html

Dependency Inversion – Direct Dependencies



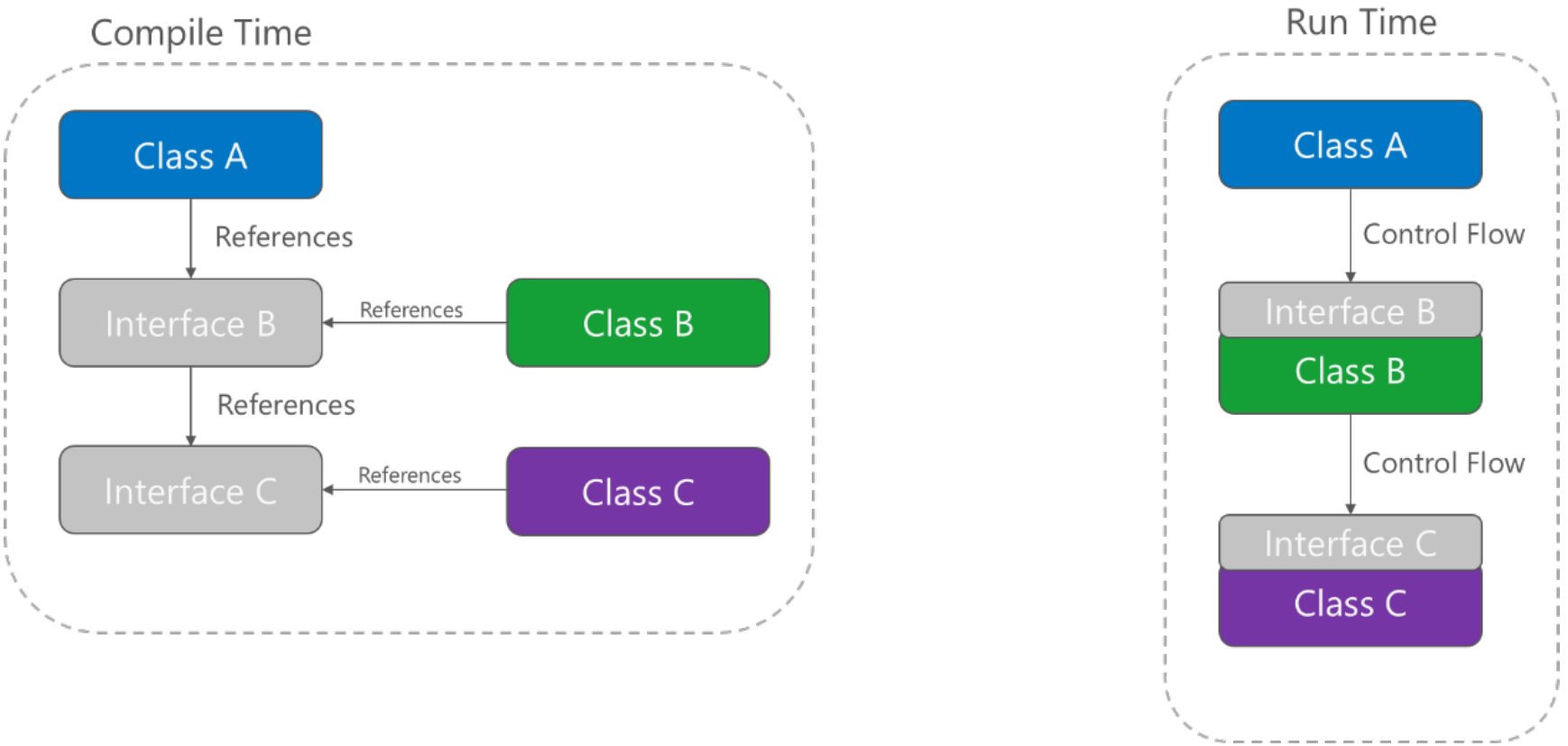
Src:
https://docs.microsoft.com/cs-cz/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles

Dependency Inversion – Direct Dependencies

- **Direct dependencies mean tight coupling between Class A and Class B**
  - It makes difficult to replace one implementation of Class B with another
    - Changes in class A would be necessary
  - → Higher level parts of system depend on the lower parts
- **Higher level abstractions should not depend on implementation details.**

Dependency Inversion

Inverted Dependency Graph



Src:
https://docs.microsoft.com/cs-cz/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles

Dependency Inversion

- **Both high- and low- level modules depend on/implement abstractions (interfaces)**
  - → loose coupling between modules
  - → easy to replace one implementation with another
    - Implementation does not impact the world beyond its interface

Explicit Dependencies

- **Applications/modules/classes should advertise required dependencies for proper functionality**

- **Solutions on several layers:**
  - Installation – e.g. Linux packaging system
  - Build – Maven, Gradle (both Java), npm (node.js), setuptools/pip (python), etc.
  - Code – constructor/function parameters

Explicit Dependencies

- **Explicit code dependencies**
  - Neither functions, classes nor methods should depend on a particular global state
  - Functions should always take all required dependencies as arguments
  - Classes should take all required dependencies as constructor parameters
    - In other words object **must** be in initialized, consistent state after constructor call
    - Optional parameters can be set via setters, but the class code must be functional even if they remain unset!

Example: explicit_dep.example

Sidestep: Inversion of Control

- **Generic term expressing that our code is *not* in control of program flow: some other code (e.g. a framework) is in control**
- **Common use-cases: dependency injection, window GUIs**
- **Common misconception: inversion of control == dependency injection**
  - Dependency injection represents the IoC concept, but not the other way around
  - See the example by Martin Fowler: *ioc.example*

Dependency Injection

- **Main idea: separate code for populating objects with their dependencies**
  - Configuration either in code or text (XML) file
  - This code is commonly provided by so called DI Containers (such as Spring, EJB, PicoContainer)
- **Result: replacing dependencies requires changes at one place (the configuration) instead of all places of use**

Dependency Injection - Constructor

Dependencies are injected via constructor

→ classes have to explicitly declare their dependencies

- Preferred for mandatory dependencies
- *Example: di_constructor.example*

Dependency Injection - Setter

Dependencies are injected via setters

→ classes have to provide setters for their dependencies

- **Preferred for optional dependencies**
- ***Example: di_setter.example***
  - You can see the examples are very similar. The difference is mainly conceptual.

Dependency Injection - Attribute

Dependencies are injected directly to attributes

→ classes don’t have to provide setters for their dependencies

- **Works with some DI containers, but hides the dependencies from their users**
  - → consider it to be an anti-pattern and avoid
  - Sometimes it can be useful though (e.g. when writing unit tests)
- ***Example: di_attribute.example***

Service Lookup

Dependency resolution approach alternative to dependency injection.

Main idea: there is an all-knowing object (service locator) which can retrieve any instance we need.

- We still need to keep reference to the service locator
- *Example with Spring: servicelookup.example*

Dependency Injection vs Service Lookup

- **DI uses Inversion of Control principle, Service Lookup does not**
- **Dependency Injection can be more difficult to debug (more complicated implementation)**
- **Service Lookup forces dependency on the service locator**
  - That is especially a problem when writing a library which others should use
- **Service Lookup is less flexible, but easier to implement**

Sources + Additional Reading

Robert C. Martin: Principles of OOD

http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Microsoft .NET Guide: Architectural Principles

https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles

Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern

https://martinfowler.com/articles/injection.html

Common Web Application Architectures

Important aspects

- **How is the application deployed – 1,2,3,n processes/servers?**
  - Impacts how application can be scaled horizontally
  - Impacts difficulty of deployment updates
- **How the application looks from the outside world**
  - Impacts how applications can be integrated
- **Development constraints**
  - Used technologies, development team allocation

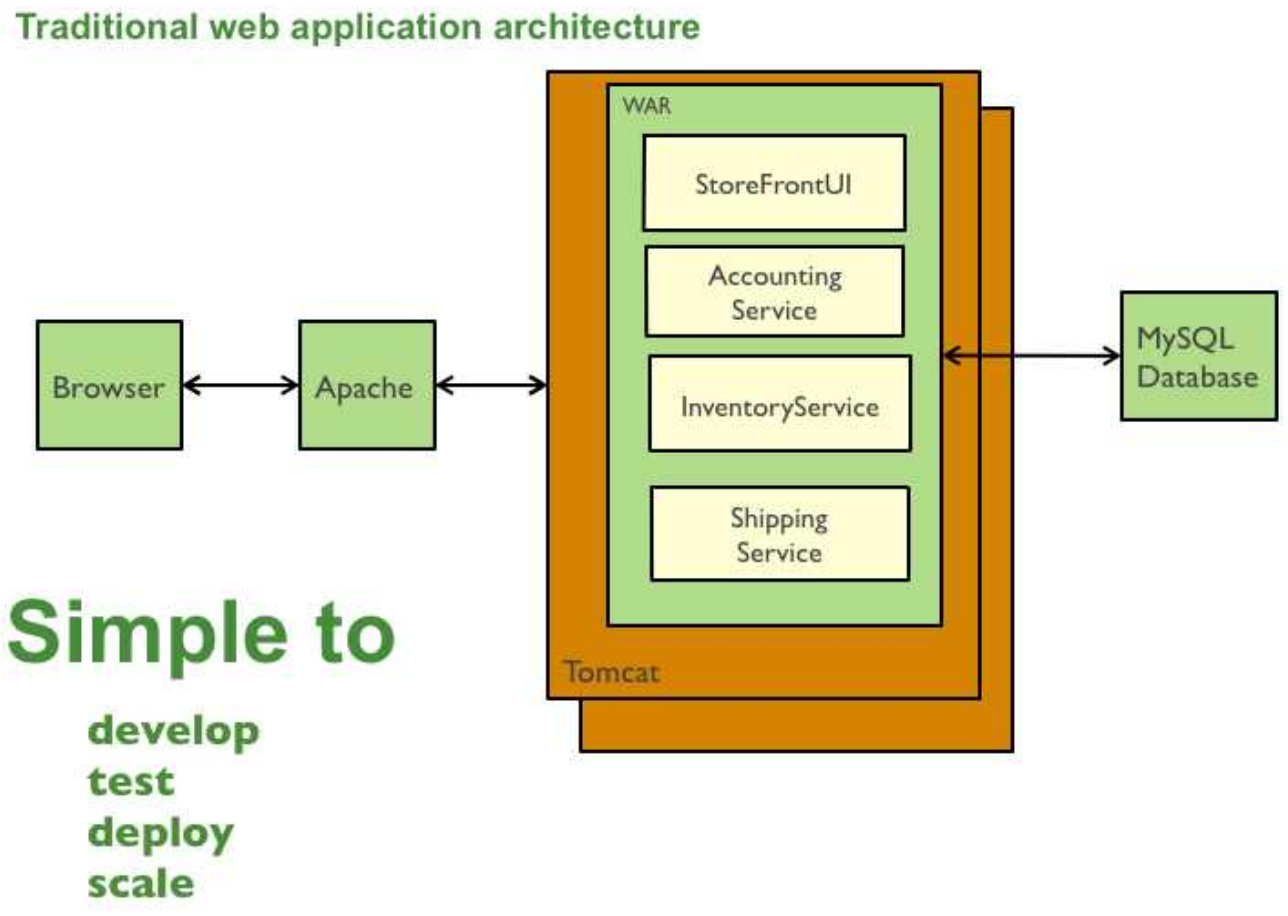
Monolithic Web Application

- **Your typical application:**
  - UI component – e.g. HTML + CSS + JavaScript running in browser
  - Server application with some kind of web-service API
    - Handles all the functionality → monolith (even though it has a web-service API!)
  - Database server for storing data – e.g. MariaDB

Monolithic Web Application

- **Example: Accounting application**
  - The server handles the following actions:
    - User authentication
    - Invoice processing
    - Salaries
    - Document printing
    - Tax returns
    - Company/Customer management
    - Payment orders
    - …

Monolithic Web Application



Src: https://microservices.io/patterns/monolithic.html

Monolithic Web Application

- **Main pros**
  - Easy to load into IDE – just one project
  - Easy to deploy – one archive/folder which is uploaded to the server runtime
  - Simple scaling – running multiple copies
- **Works nicely until the application grows, then...**

Monolithic Web Application

- **Main cons (when the application is large)**
  - Large codebase is difficult to understand – especially for new team members
    - → decline of code quality over time
  - Long start-time for VMs (e.g. Java, .NET) – wasted time during development and problems with deployment up-time
  - Scaling is not so easy anymore
    - Cannot scale just portions which are heavily used
    - Various resource requirements by different parts of the application

Monolithic Web Application

- **Main cons (when the application is large)**
  - Problems with scaling development
    - People influence each other with their work → big coordination costs
  - Fixed technology stack
    - Whole application must be written in the same technology
    - Upgrades to newer framework versions are difficult due to size and amount of people involved
    - Change of framework means rewriting a lot of code

Service-Oriented Architecture

- **Attempt on solving issues with monolithic applications**
- **Or a corporate buzzword…**

- **Nowadays almost useless term:**
  - Many people understand SOA as many different things
    - Google the term and see for yourselves
  - Read: https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

- **So instead of wasting time on SOA, we define what a service is and work with that**

Service-Oriented Architecture

- **Services should:**
  - Be a black box for their consumers
  - Have a clear agreement on how-to communicate with them (client-service contract)
  - Be loosely-coupled – services need to know only contract to communicate with each other → no direct, implementation-specific dependencies
  - Be stateless – service either responds to client with a value or an error message

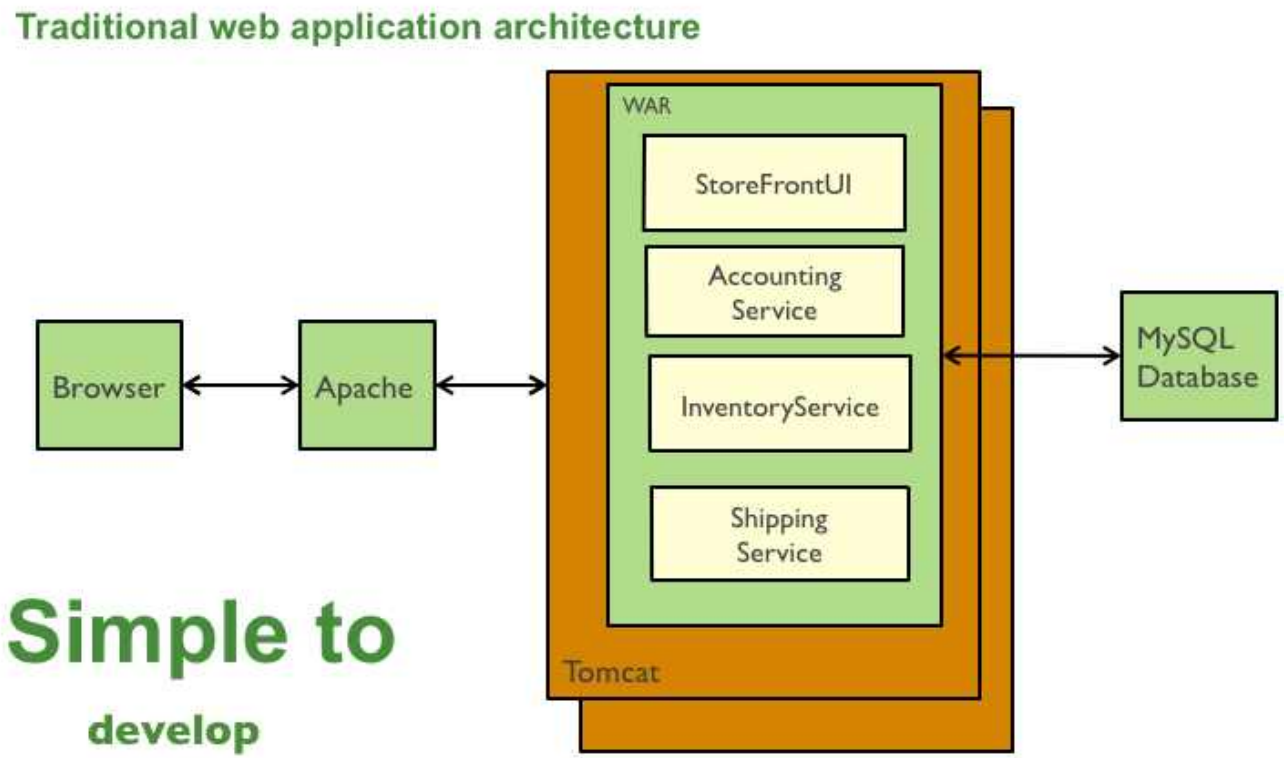
Service-Oriented Architecture

- **Services communicate by sending messages via:**
  - HTTP – web-services, SOAP
  - Message queues
- **Increases complexity due to integrations**
- **→ Monolithic applications integrated via web-service API are not a good SOA**
  - Issues with monolithic architecture remain
  - Additional complexity of service integrations

Microservices

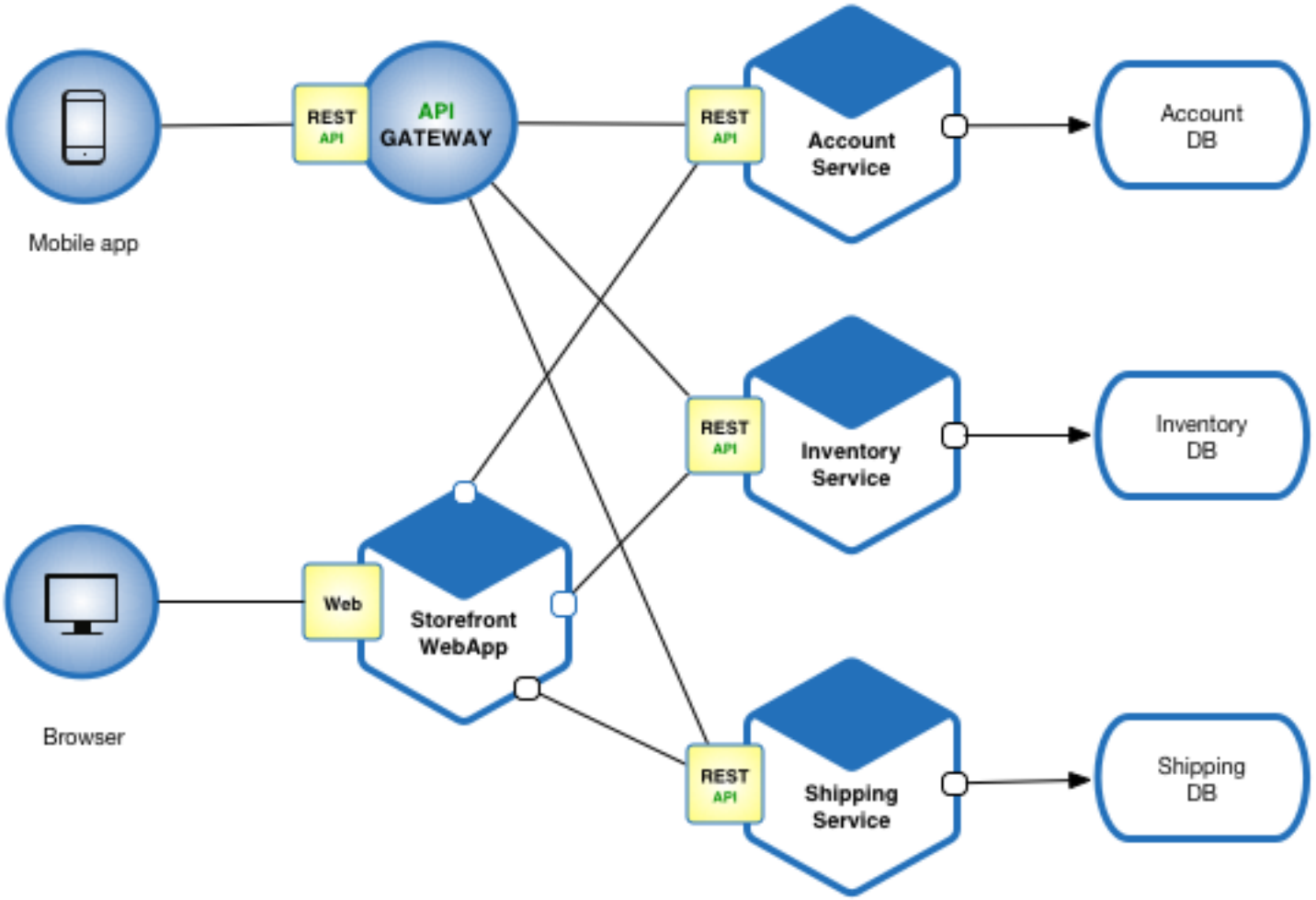
- **One application splitted into several small services**
  - Each runs as separate process on either same, or more often different (virtual) server
- **Based on Unix concept of “Do One Thing and Do It Well”**
- **Each service handles one small well-defined area of functionality**

Example:
Monolithic Web Application



Src: https://microservices.io/patterns/monolithic.html

Example: Microservices



Src: https://microservices.io/patterns/microservices.html

Microservices

- **Main pros**
  - Each service is relatively small
    - Easy to develop (also for new team members)
    - Easy to test individual functionality
  - Fast deployment – small services start faster
  - Flexible horizontal scalability – you scale only those services which you need
  - You don´t need to take down entire application for updates

Microservices

- **Main pros**
  - Easy to scale development
    - Teams can take responsibility of a service → limited interference
  - Each service can be implemented using different technology
  - Relatively easy technology upgrades and refactoring due to limited scope of services

Microservices

- **Main cons**
  - Relatively difficult to deploy due to distributed nature
    - Industry standard development tools are only slowly adapting from monolith applications to microservice pattern
    - Containers (Docker) help a lot with that
    - → increased maintenance costs
  - Integration testing required – and expensive

Microservices

- **Main cons**
  - Use-cases that span across multiple services present a challenge
    - Require coordination of multiple teams
    - Transaction handling is difficult in distributed environment
  - Possible increased resource costs
    - Each service should have own datastore
    - Each service process usually runs in on VM
    - Each service requires own runtime which may have significant overhead (e.g. JVM)

Monolith vs Microservices

- **Each has its uses**
- **Monolith is easier to develop and maintain as long as the application remains relatively small**
- **Complexity of microservice distributed deployment does not pay off for small applications**
- **But for large projects, the microservices allow better scalability, testing, and decrease complexity of individual code packages**
  - And for languages without static type system, microservices are almost a necessity

Sources + Additional Reading

Microservice Architecture:
https://microservices.io/index.html

ASP.NET Guide: Common Application Architectures
https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures

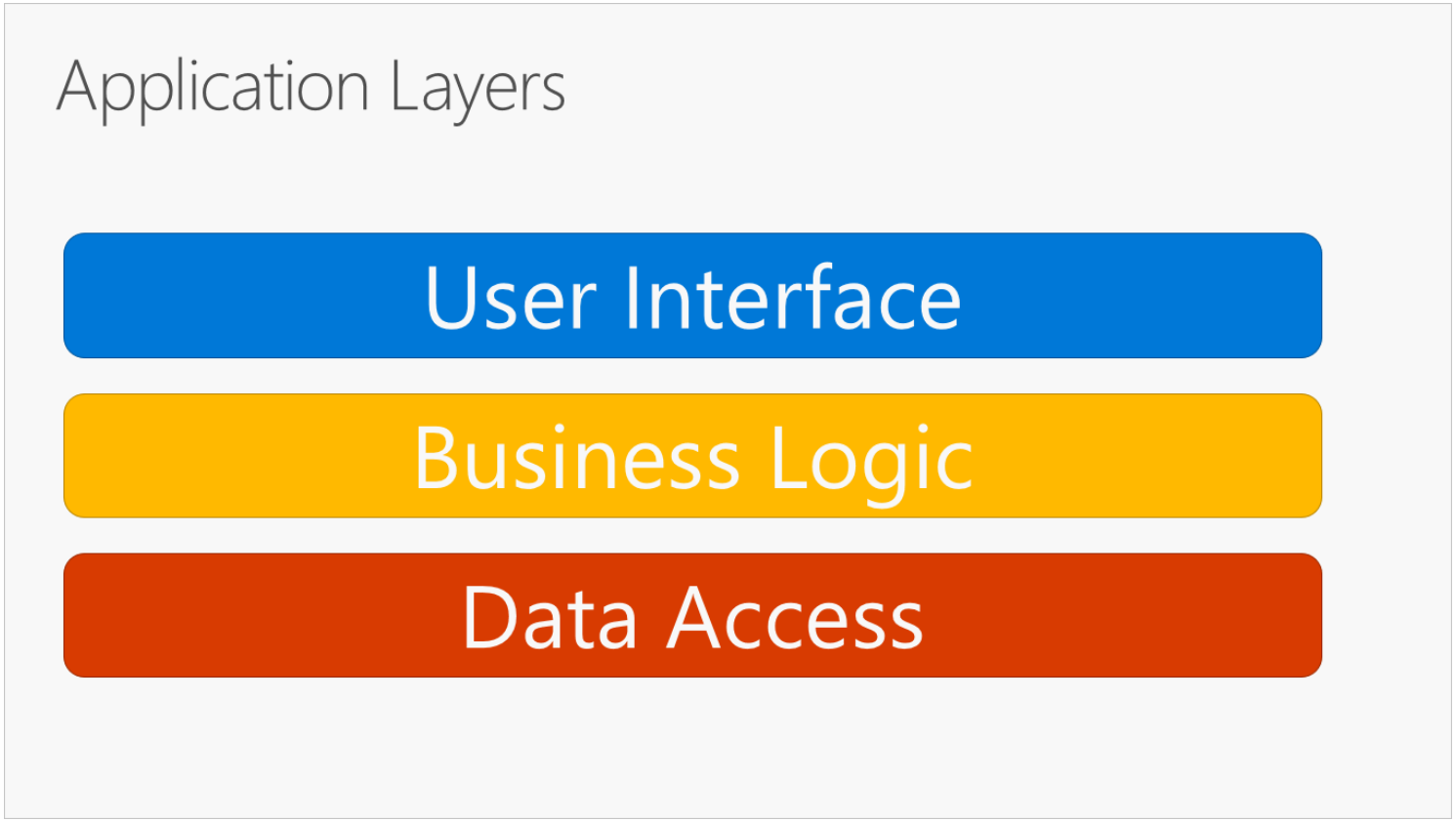
Martin Fowler: Microservices
https://martinfowler.com/articles/microservices.html

Common Web Application Design Principles

"If you think good architecture is expensive, try bad architecture."
- Brian Foote and Joseph Yoder

Application Layers

- Application code is separated into “layers” by its responsibilities (separation of concerns)



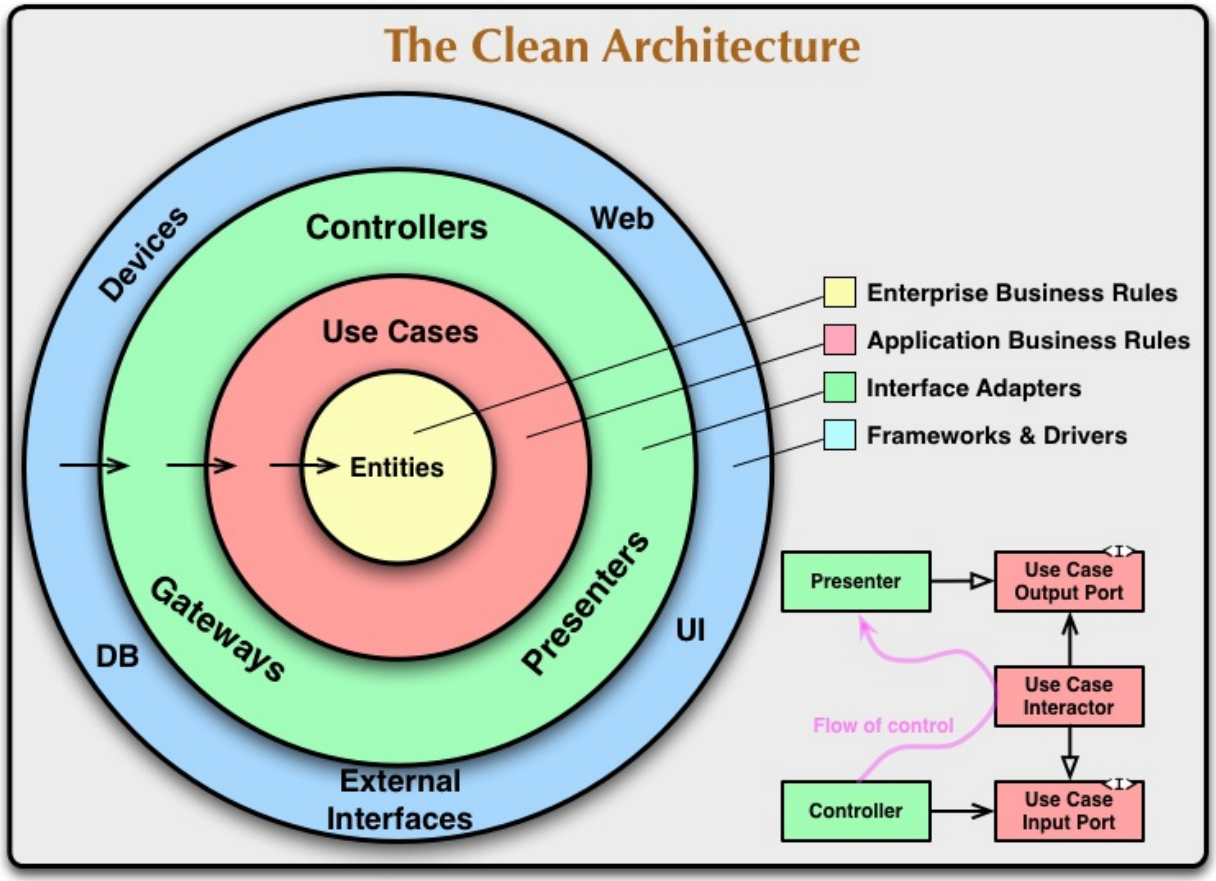
Src: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures

Application Layers

- **Layers encapsulate functionality inside application**
- **Layers communicate via interface**
- **→ easier management and replacement**
  - e.g. MySQL storage replaced by MongoDb
- **→ easier testing**
  - e.g. replace real database access with mocks for unit tests

Application Layers – Clean Architecture

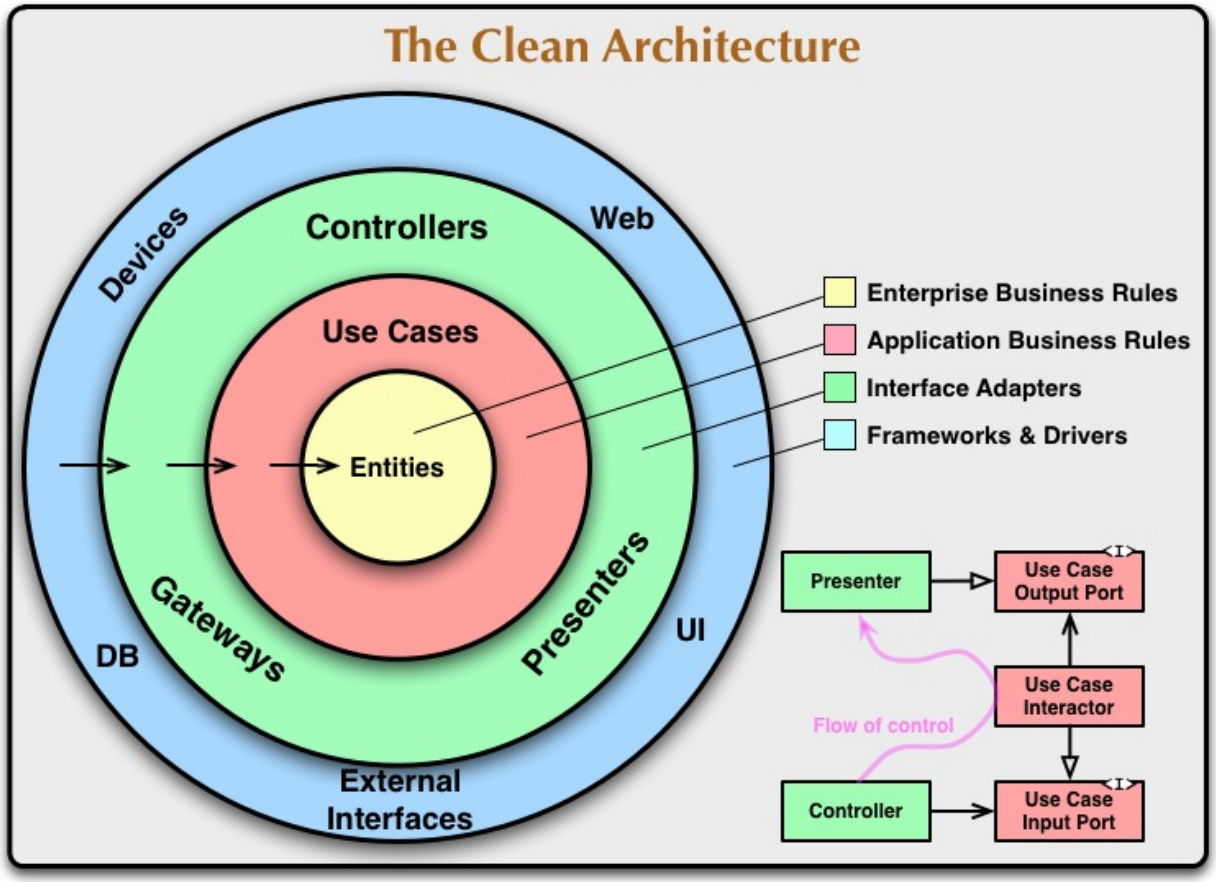
- Different schema, same concept



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

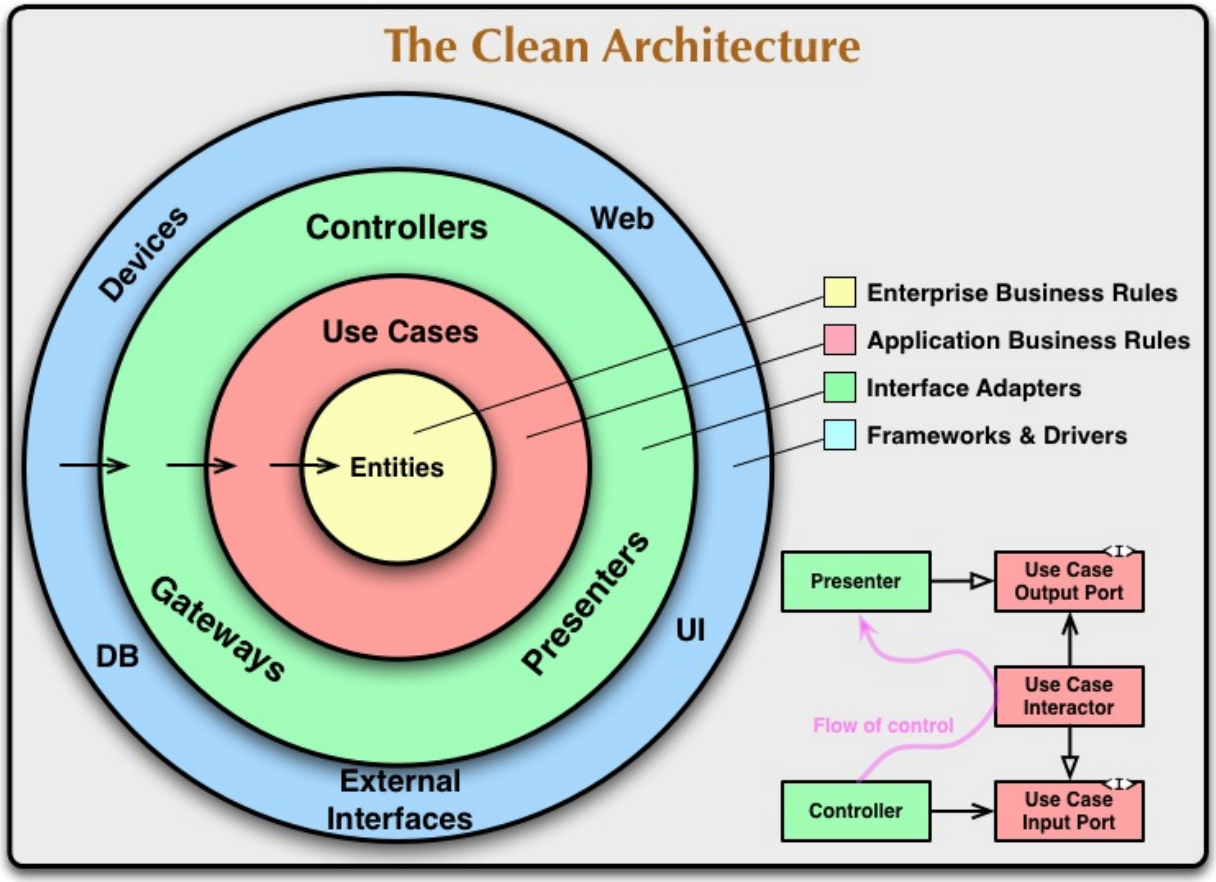
- Main rule: dependencies are always from the outside inwards



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

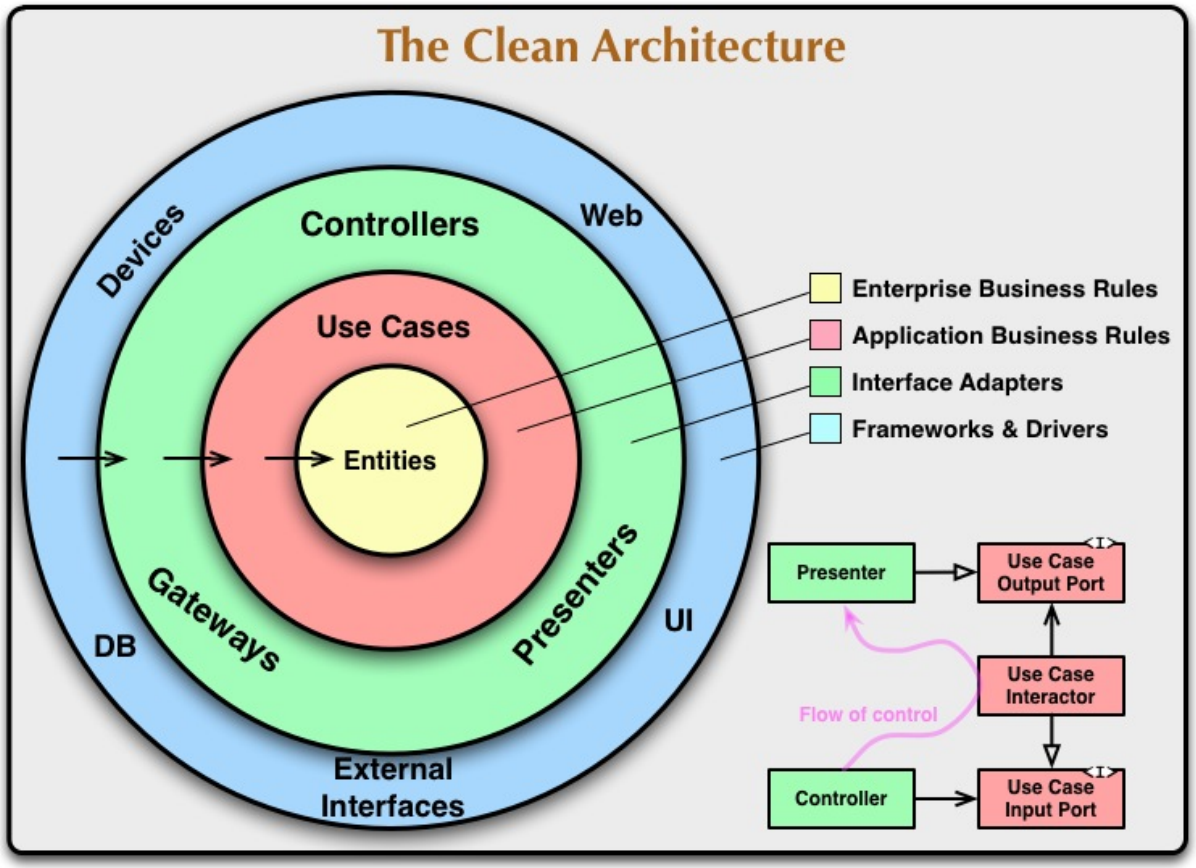
- Entities: Generic rules shared across whole system



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

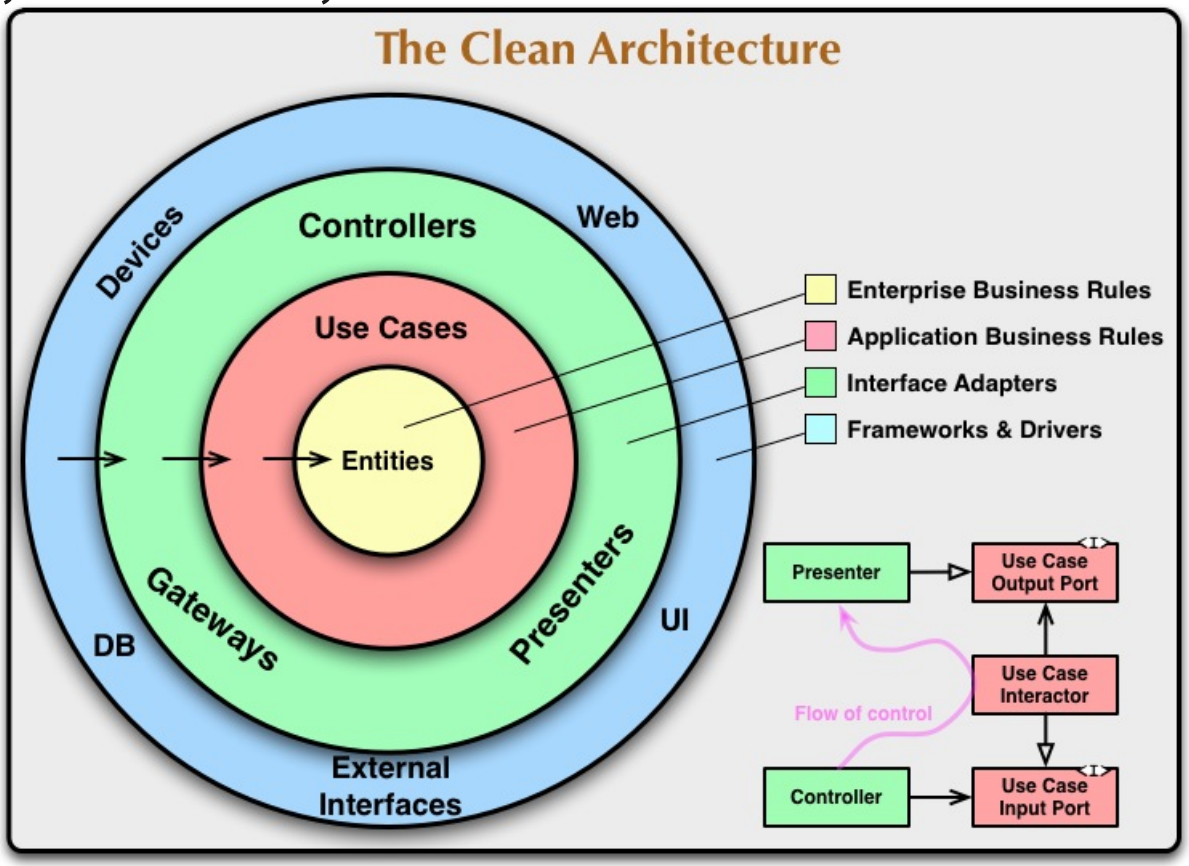
- Use Cases: application-specific rules, control data flow to/from entities and invoke their functions



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

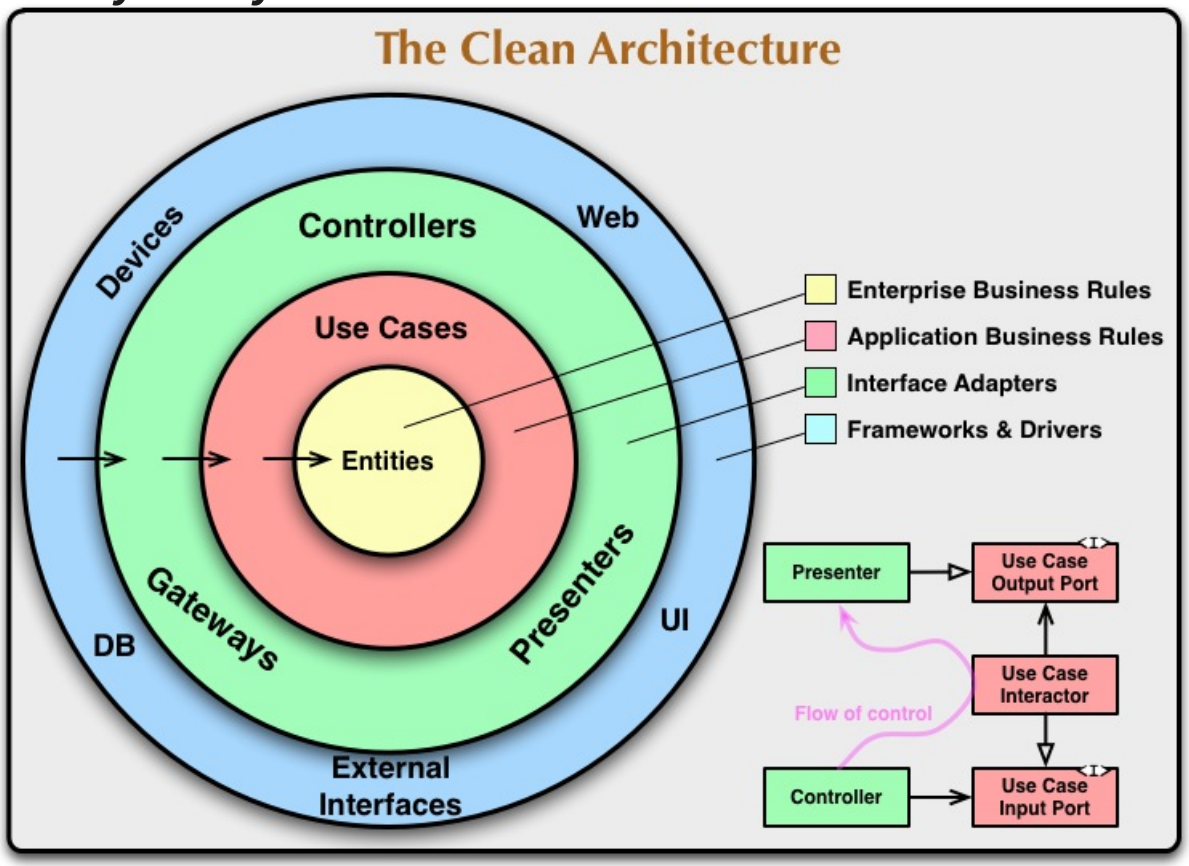
- Interface adapters: convert data from entities/use cases into format for external systems – UI, web services, databases



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

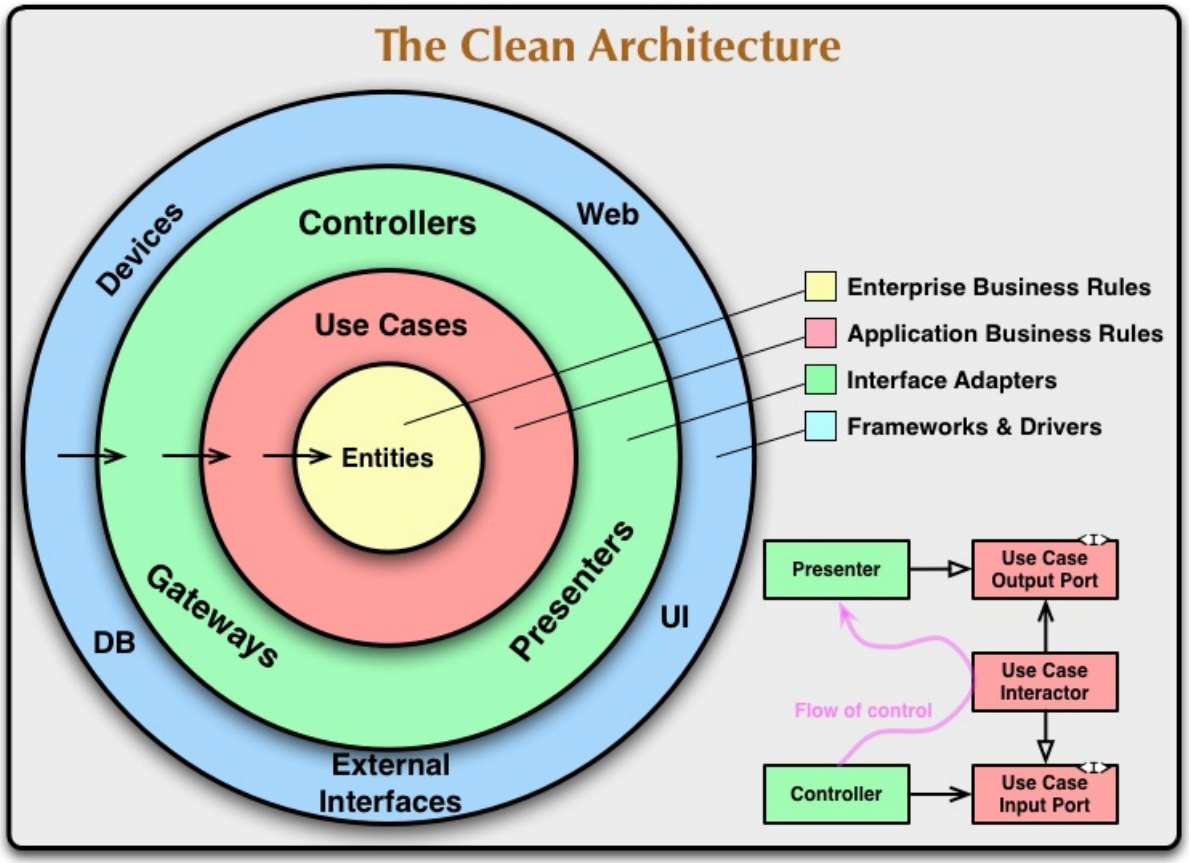
- Frameworks and drivers: external frameworks and tools, databases, etc. You only connect those to your system.



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

- More layers as needed. The concept is what is important here!



Src: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Application Layers – Clean Architecture

Simple Example: Discussment library

https://github.com/danekja/discussment

- **Important things to check:**
  - Inner references in core module (Service classes reference – depend on – the entities)
  - Core module provides interfaces for DAOs (Dependency Inversion)
  - Concrete DAO implementations depend on Core
  - UI module depends on Core

Application Layers

- **Realization of separation of concerns principle and encapsulation on code level**
- **Dependency paths go from upper/outer layers to lower/inner layers**
  - Commonly implemented vie dependency inversion principle
- **Allows reuse of inner parts on multiple outer places**
  - e.g. application logic can be called from:
    - Web UI
    - Smartphone app via web services
    - Test framework during unit tests

Application Layers

- **Common layers**
  - Presentation layer
  - Application logic layer
  - Data Access layer
- **Common cross-cutting concerns**
  - Security
  - Transactions
  - Logging

Data Layer

- **Responsibilities:**
  - HOW the data are stored (format)
    - Relational schema, rules for documents in document databases, file format, etc.
  - Mapping between internal application format (entities) and storage format (SQL)
  - The actual READ/WRITE functionality

Business Layer

- **Responsibilities:**
  - Realize application functionality
  - Change state of entities
  - Trigger data storage access

Presentation Layer

- **Responsibilities:**
  - Accept input from user
  - Call application functions
  - Present result to user

Further reading

- Robert C. Martin, Clean Architecture Overview
  https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html
- Robert C. Martin, Clean Architecture, Prentice Hall; 1 edition (September 20, 2017), ISBN-13: 978-0134494166
  (yep, the book)
- Brian Foote, Joseph Yoder: Big Ball of Mud, Fourth Conference on Patterns Languages of Programs,PloP 97
  http://www.laputan.org/mud/mud.html#BigBallOfMud
- Alistair Cockburn, Hexagonal Architecture
  https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture
- Jeffrey Palermo, Onion Architecture
  https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- ASP.NET Guide: Common Application Architectures
  https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures

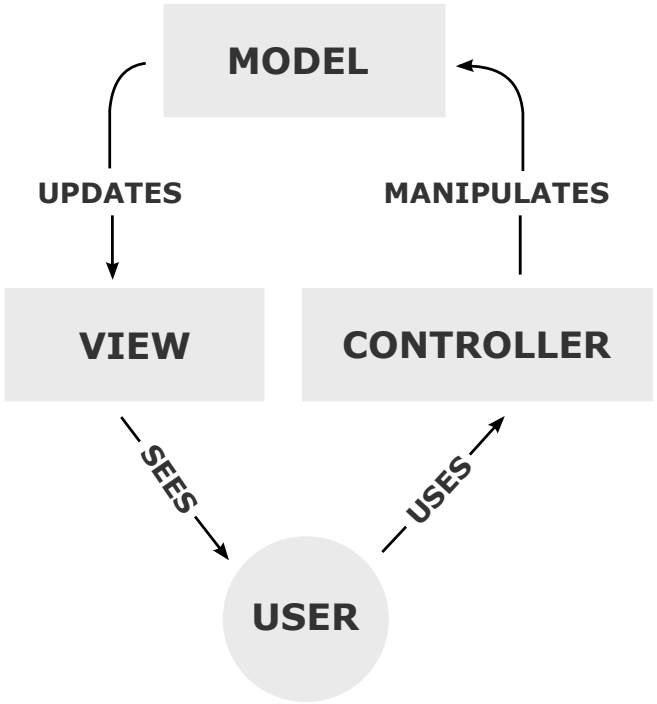
Common Web Application Design Patterns

Common WebApp Design Patterns

- The following sections discuss commonly used design patterns
- Exact definitions of these patterns vary based on source, technology...
- → remember the principle of separating concerns
- → always consider the best approach to solving your goal
- → do not focus on following a pattern just for the sake of it

Model-View-Controller

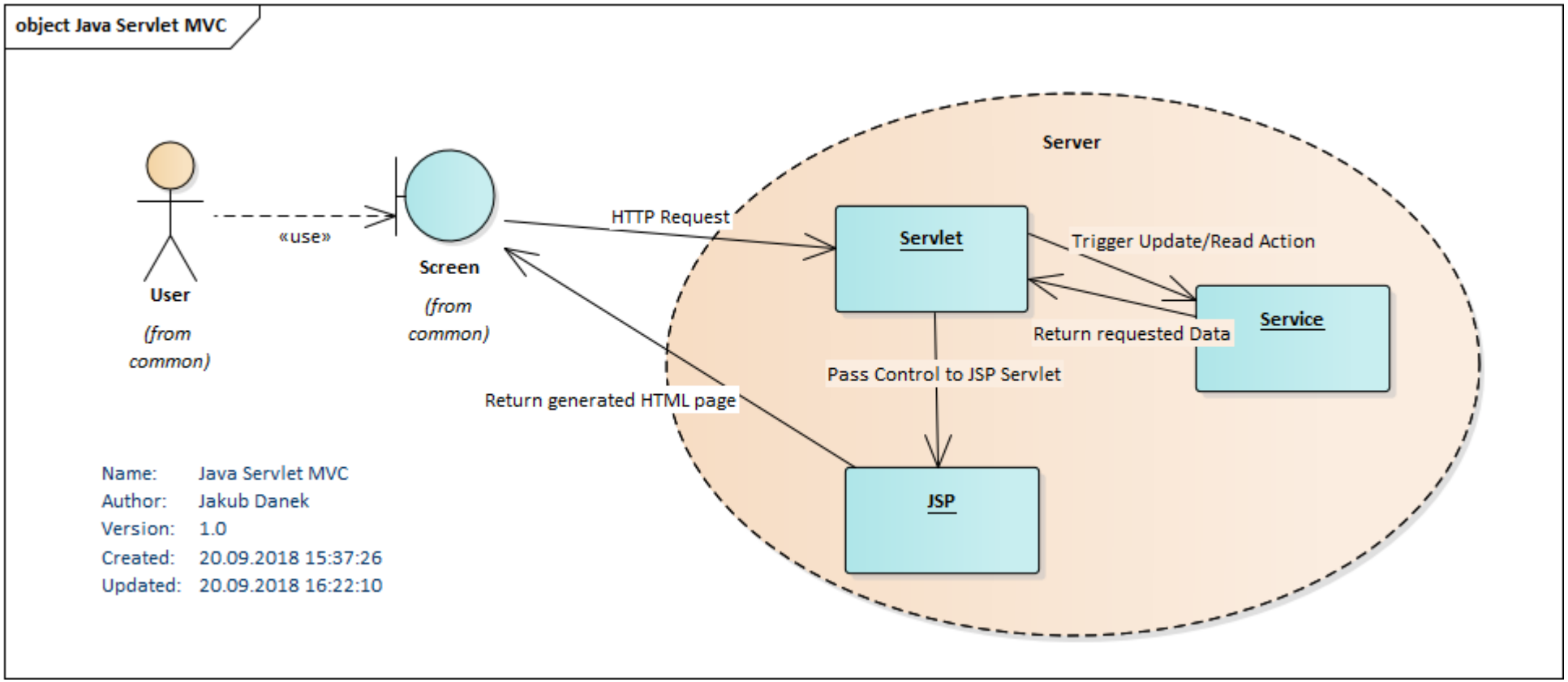
- **Terminology**
  - Model – data and application logic
  - View – displays data from model
  - Controller – handles input events
- **Originally from Smalltalk-80 GUI**
- **Many flavours in web applications**



Img src: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Model-View-Controller

- Java JSP: whole MVC on the server



Model-View-Controller

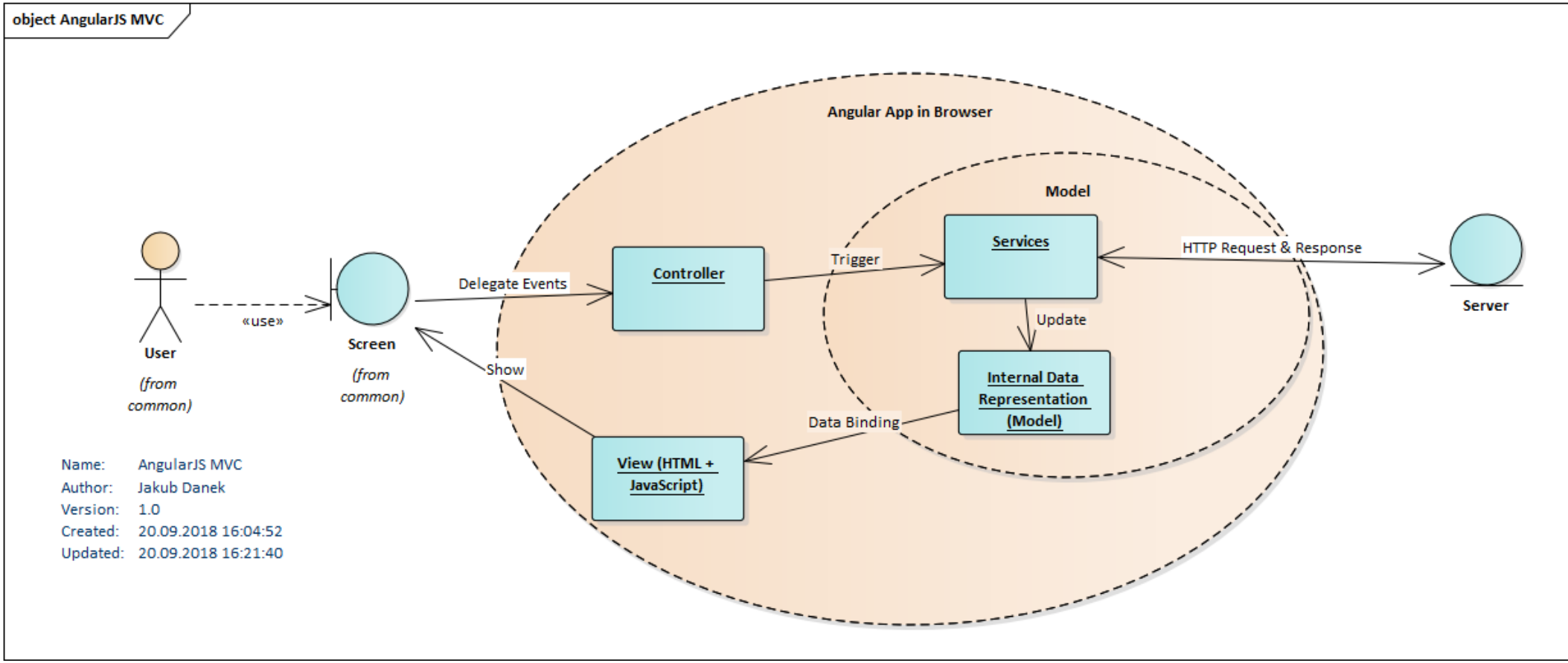
- **Java JSP: whole MVC on the server**
- **Differences to basic MVC schema:**
  - Model (services) do not directly inform view (JSP) about changes – controller does that
    - Instead, new View is generated for each request in form of HTML page
    - → notifications about Model change are not necessary
  - JSP is the actual View, HTML page is its output
  - Note: Both Controller and View are implemented as Servlets (remember what servlet container does with JSPs?)

Model-View-Controller

- **AJAX/SPA: Part/Full application in browser**
- **Terminology: These terms are not limited to MVC!**
  - AJAX = Advanced Javascript and XML
    - Technology which allows updating html directly in browser without the need to get full page HTML from the server
  - SPA = Single Page Application
    - Application which run fully in browser and view changes are handled by Javascript manipulation of DOM
    - Data are received from the server, but HTML is generated in browser

Model-View-Controller

- AJAX/SPA: Part/Full application in browser

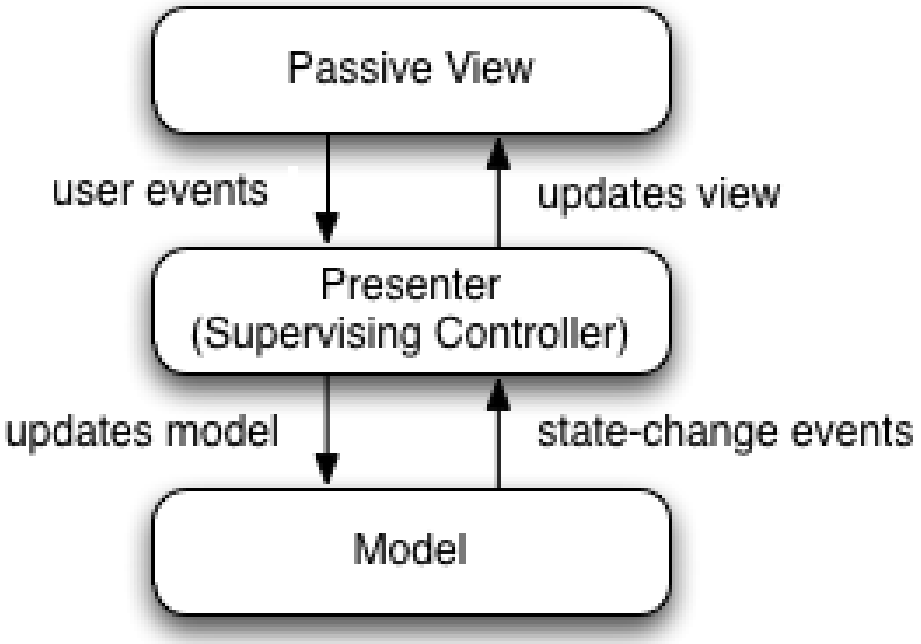


Model-View-Controller

- **AJAX/SPA: Part/Full application in browser**
  - Controller handles input events (link/button click, input changes) and delegates to model for action
  - View observes the model data for change
  - → when data change, view updates itself

Model-View-Presenter

- Breaks connection between Model and View
- Presenter is a Controller which also serves as direct mediator between View and Model



Img src: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter

Model – View - Presenter

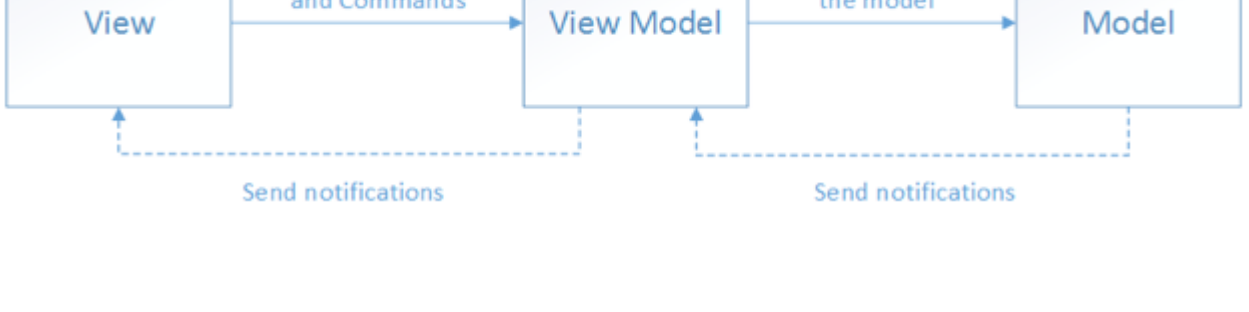
- **View consists of individual Widgets**
  - Widget can be a:
    - Button
    - Input field
    - Set of input fields
    - A form
- **Each Widget has own Presenter which handles its events and sets the data it shows**

Model – View - Presenter

- **Presenters form a hierarchy:**
  - Page Presenter
    - Menu presenter (handles View changes)
    - Form Presenter (handles form submit actions)
      - Input fields presenters
      - Submit button presenters (handles e.g. when the button is enabled)
- **Presenter receives user input event, delegates to Model and then updates the View with result**

Model – View - ViewModel

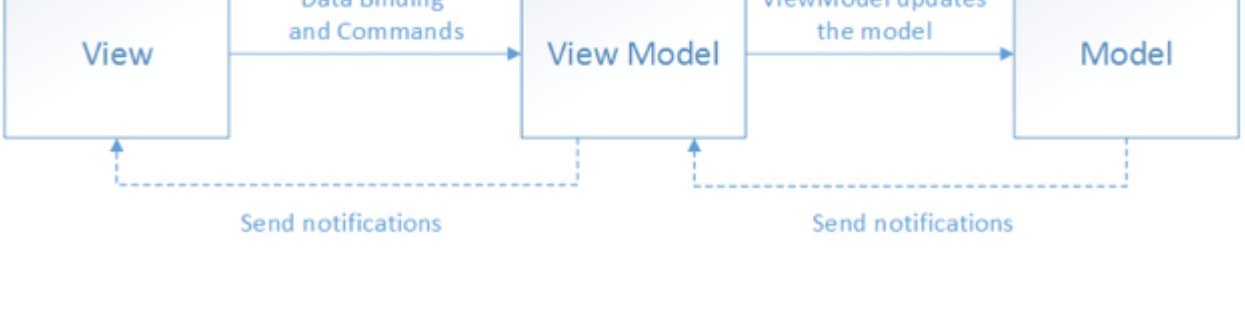
- Another way of breaking coupling between Model and View



- Originally introduced by Microsoft for its ASP.NET/WPF frameworks

Img src:
https://docs.microsoft.com/cs-cz/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

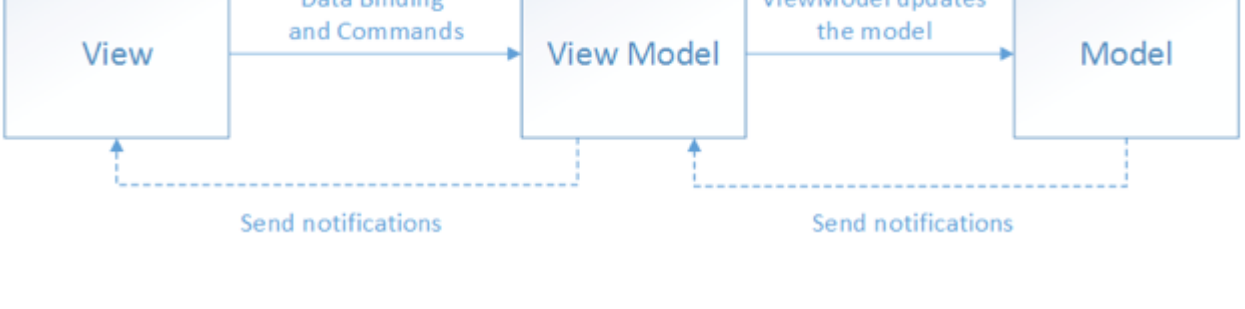
Model – View - ViewModel



- **ViewModel provides**
  - Data for View
    - Converts data from internal Model into format which suitable for display
  - Commands
    - Reactions to user input – commands are mapped to button clicks, input changes etc.

Img src:
https://docs.microsoft.com/cs-cz/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Model – View - ViewModel



- **Communication with View:**
  - Via DataBinding and Notifications – View is notified about data change in ViewModel via events and updates itself
- **Communication with Model**
  - ViewModel calls Model functions to trigger application logic
- **→ ViewModel forms both data and functional interface between View and Model**

Img src:
https://docs.microsoft.com/cs-cz/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Model – View - * Comparison

	MVC	MVP	MVVM
User Input	Controller handles all user input and calls Model/changes View	Widgets pass user input to their respective presenter, which calls Model/changes View	Actions in View mapped to Commands in ViewModel. Commands call Model/trigger View change
View Updates	View observes the model and updates itself when notified	Presenter updates View when Model changes	View observes ViewModel data and updates itself when notified
View – Model communication	Yes, View observes Model	No	No

Model – View - Whatever

- **Many different opinions on exact definition of these patterns**
- **Each application implements them in their own flavour**
- **→ Do not worry about terminology too much**

- **Important things to take away:**
  - Separate your presentation and application logic
  - There are multiple paths for such separation
  - Follow guidelines and best practices for your chosen technology as well as internal team rules
    - → same structure everywhere → easier to navigate through new project

WebApp Design Patterns Sources

- Mike Potel, MVP
  http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
- Martin Fowler, GUI Architectures
  https://martinfowler.com/eaaDev/uiArchs.html
- Martin Fowler, Observer Synchronization
  https://martinfowler.com/eaaDev/MediatedSynchronization.html
- DrDobbs, MVC Paradigm in Smalltalk
  http://www.drdobbs.com/tools/the-mvc-paradigm-in-smalltalkv/184408445
- Microsoft MVVM
  https://docs.microsoft.com/cs-cz/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Cross-Cutting Concerns

Cross Cutting Concerns

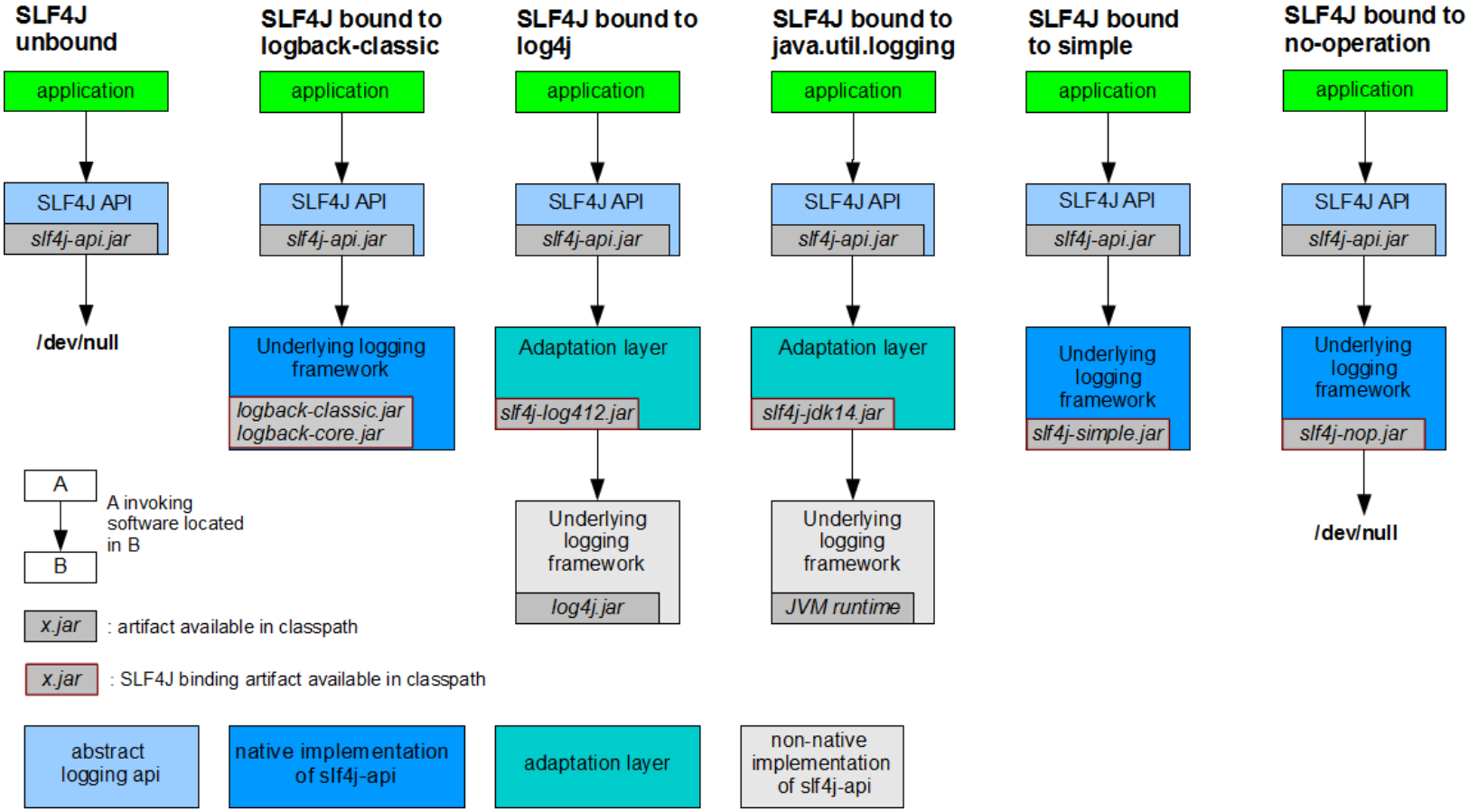
- **Certain functionality is required on all layers**
  - Security – user authorization
    - Presentation layer – to decide if user can see particular page/view
    - Application layer – to decide if user can invoke particular action/read data
  - Logging – technical, debug logs, audit logs
    - Everywhere to log what the application/user are doing
  - Transactions
    - Application and Data layer to ensure atomicity of actions – we will talk more about this in the persistence lecture

Cross Cutting Concerns

- **Certain functionality is required on all layers**
  - naive implementation introduces dependencies into all layers
    - expensive to change
    - Clutters implementation with unrelated code
    - Difficult to test

Cross Cutting Concerns

- Approach 1 – API dependency and configuration



Src: https://www.slf4j.org/manual.html

Cross Cutting Concerns

- **Approach 1 – API dependency and configuration**
  - Commonly used with logging facades
  - Application depends on interface, not implementation
  - Allows to call logging API at any place of your code
  - Changing logger implementation requires only:
    - Replace logger implementation library
    - Provide new configuration
  - Actual application code does not need to change

Cross Cutting Concerns

- **Approach 2 – Aspect-Oriented Programming (AOP)**

- **What is an Aspect?**
  - A module (typically a class) that contains code which would be otherwise called/duplicated among multiple classes
  - Technologies implementing AOP ensure that the Aspect code is executed when particular method is called (and other conditions apply)
    - e.g. logging aspect could for each method call of any class in our application's business layer log: timestamp, user, method and class name, input parameters
    - e.g. security aspect could check that all methods from our business layer are called only if user has been authenticated

Cross Cutting Concerns – AOP Example

```
//no aspect
Class UserManager {
      AuthenticationManager auth;
      User addRoleToUser(User user, Role role) {
          //throws unauthorized exception on fail
          auth.checkCurrentUserHasRole(Roles.ADMIN);
          //attach role to user, save etc
          ...
      }
}
```

Cross Cutting Concerns – AOP Example

```
//with aspect
Class UserManager {
 User addRoleToUser(User user,
                    Role role) {
       //attach role to user, save
         etc
       ...
     }
}
```

```
//with aspect
@Aspect
Class AuthAspect {
   AuthenticationManager auth;
   @Before("execution(* UserManager.add*(..))")
     public void doAdminCheck() {
       auth.checkCurrentUserHasRole(Roles.ADMIN)
     }
}
```

Cross Cutting Concerns - AOP

- **Terminology:**
  - Join Point – place of program execution at which the cross-cutting code is run
    - Commonly is a method execution (but not exclusively)
      - This course focuses on method execution only
  - Pointcut – predicate expressing whether action should be executed at certain join point
    - e.g. `execution(* UserManager.add*(..))`

Cross Cutting Concerns - AOP

- **Terminology:**
  - Advice – action taken by aspect – joins pointcuts and the actual code to be executed
    - e.g. `@Before("execution(* UserManager.add*(..))")`
      `public void doAdminCheck(){…}`
  - Types of Advices
    - The aspect code can be executed at several places in respect to the particular join point:
      - Before
      - After
        - After returning (with the return value)
        - After throwing (with the exception thrown by the method)
      - Around (meaning part of aspect code is executed before and part is executed after the actual join point)

Cross Cutting Concerns - AOP

- **Terminology:**
  - Weaving – process of linking aspects to the advised code
  - Types of Weaving
    - Compile time
    - Load time
    - Runtime

Cross Cutting Concerns - AOP

- **Compile-time weaving**
  - Accomplished by custom compiler
    - Analyses code, searches for aspects and pointcuts
    - Creates modified version of classes which contain the relevant code
  - Issues:
    - Custom compiler required
    - Debugging tools might have issues with figuring out program execution mapping to source code

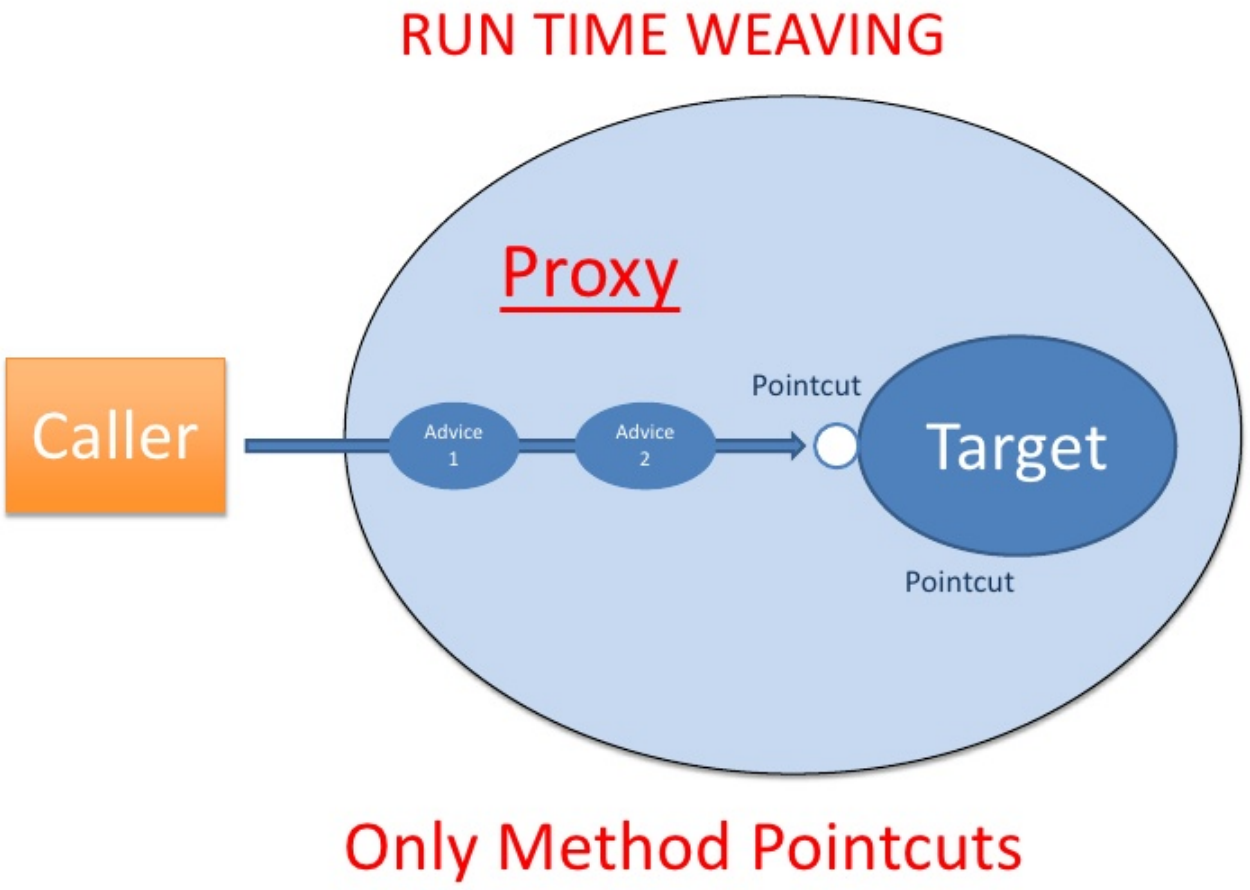
Cross Cutting Concerns - AOP

- **Load-time weaving**
  - Accomplished by custom class loader
    - Analyses binaries, searches for aspects and pointcuts
    - Creates and loads modified version of classes which contain the relevant code
  - Issues:
    - Custom class loader required
    - Debugging tools might have issues with figuring out program execution mapping to source code

Cross Cutting Concerns - AOP

- **Run-time weaving**
  - Accomplished by creating proxies of the classes by subclassing
    - References to proxies are used instead of the actual objects
    - Proxies contain the aspect's code and invoke the original methods
  - Pros:
    - The original classes remain intact at runtime
  - Issues:
    - Bigger memory footprint due to creating subclasses and their instances
    - It is difficult/impossible to use pointcuts pointing at private methods

Cross Cutting Concerns - AOP



Src: https://www.slideshare.net/rohitsghatol/aspect-oriented-prog-with-aspectj-spring-aop/40

Cross Cutting Concerns – Sources & Reading

https://docs.jboss.org/aop/1.1/aspect-framework/userguide/en/html_single/index.html

http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

https://docs.spring.io/spring/docs/5.0.9.RELEASE/spring-framework-reference/core.html#aop

https://www.slideshare.net/rohitsghatol/aspect-oriented-prog-with-aspectj-spring-aop