Some buzzwords and acronyms for today

- Software architecture
- Design pattern
- Separation of concerns
- Single responsibility principle
- Keep it simple, stupid (KISS)
- Don't repeat yourself (DRY)
- Don't talk to strangers (Demeter's law)
- You aren't gonna need it (YAGNI)
- Inversion of Control (IoC)
- Dependency injection (DI)
- Data Access Object (DAO)
- Model View Controller (MVC)
- Hollywood principle
- Encapsulation
- High cohesion, loose coupling

1 Why?

Why should we think about architecture/design?

- Development
  - Adding new features into a mess is more difficult
  - Accommodating new requirements is easier for well-designed application
- Maintenance
  - More resources are spent on maintenance than development

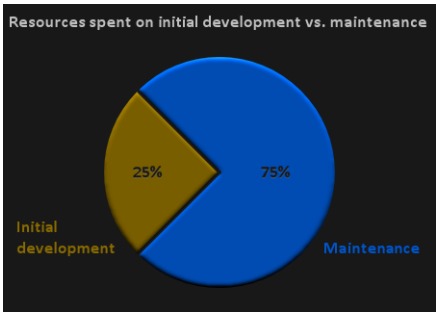


Figure 1: Resource: `http://clarityincode.com/software-maintenance/`

Why should we think about architecture/design?

- Documentation
  - Developers tend to change jobs often (1.5 - 3 years [1])
  - Newcomers need to get up to speed quickly
- Efficiency
  - Clean code is usually more efficient than messy code
- Error prevention
  - Clean code is less prone to bugs

Why architectural styles/design patterns?

- Proven best practice solutions
- Means of communication
  - Documentation
  - Communication between developers
- Improve code structure

2 Software Architecture

What is a software architecture?

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation– are not architectural.

- Bass, Clements, and Kazman *Software Architecture in Practice (2nd edition)*

Architecture	**Component design**
Changes slowly	Rapid change through refactoring
Influences the whole system	Specific for the component
Architectural styles	*Design patterns*

Software architecture

Architecture describes the overall structure of a software system. Good architecture enables smooth evolution of the system.

It must take into account things like

- Deployment environment
- Platform and technology specifics
- Expected system scope

Architecture design principles

Standard design principles also apply to system-wide architecture

- *Separation of concerns*
- *Single responsibility principle*
- *Law of Demeter*
- *Don't repeat yourself*

Before you design the system architecture, you need to

- Determine application type
- Determine deployment strategy and environment
- Determine technologies to use
- Determine quality attributes
- Determine cross-cutting concerns

Architecture example

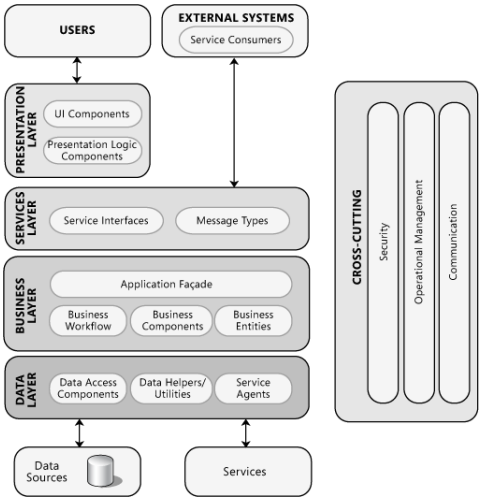


Figure 2: System architecture example. Source: `https://msdn.microsoft.com/en-us/library/ee658124.aspx`

System architecture

- Usually consists of multiple architectural styles
- Should be well understood by the team
- Should be documented (diagrams, pictures, notes)
- Should clearly expose system structure, while hiding implementation details
  - I.e. show where stuff happens, but not how
- Address all user scenarios (eventually)
- Should handle both functional and non-functional requirements
- Evolves as the software grows

3 Architectural Styles

Architectural styles

- There exist plenty of architectural styles
- They are usually combined in an application
- Different styles are suitable for different scenarios
- Various ways of architectural style classification

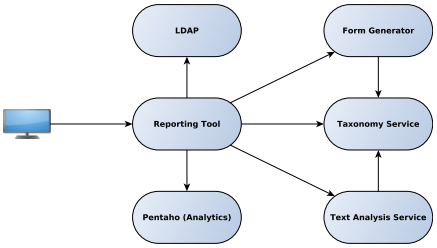


Figure 3: SOA system example.

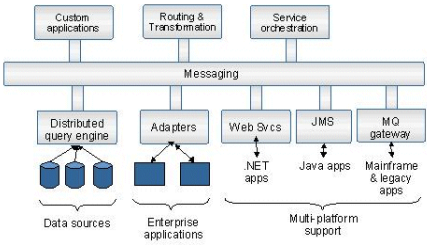


Figure 4: ESB architecture. Source: `https://docs.oracle.com/cd/E23943_01/doc.1111/e15020/img/esb_architecture.gif`

Architectural styles - Communication

Service-Oriented Architecture

- Distributed applications provide services for each other
- Using standard protocols and data formats (REST – HTTP and JSON/XML)
- Loose coupling, easy implementation switch
- *Microservices*

Architectural styles - Communication II

Message Bus

- Central message queue handles message distribution
- Asynchronous messages between clients
- Loose coupling, scalability
- *Enterprise Service Bus* – provided by Oracle, IBM etc.

Architectural styles - Deployment

Client/Server

- Client sends requests, server responds
- Web applications use this pattern
- Server – possible single point of failure and scalability issue

N(3)-tier

- Independent tiers providing functionality
- Easier scaling
- E.g. load balancing, company firewall

Architectural styles - Domain

Domain-driven Design

- Business components represent domain entities
- Suitable for modelling complex domains
- Common language and model for developers and domain experts

Architectural styles - Structure

Object-oriented

- Objects consist of both behaviour and data
- Natural representation of real world
- Encapsulation of the implementation details

Layered

More on layers later...

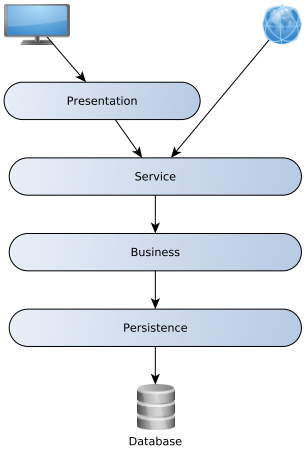


Figure 5: Layered system architecture.

Architectural styles - Structure II

Component-based

- System decomposed into logical or functional components
- Components provide public interfaces
- Supports separation of concerns and encapsulation
- Components can be managed by architecture provider
  - *Dependency injection* and *Service locator* used to managed dependencies
- Components can be distributed
- Higher level than OOP

3.1 Layered Architecture

Layered architecture

- Layers of related functionality
- Typical for web applications
- Behaviour encapsulation, clear separation of concerns, high cohesion, loose coupling
- Testability

Layered architecture II

- In contrast to *N-tier architecture*, the layers are usually in one process (e.g. application server)
- Each component communicates only with other components within the same layer or in the layer(s) below it

  Strict interaction Layer communicates only with the layer directly below

  Loose interaction Layer can communicate also with layers deeper below
- Cross-cutting concerns stem across all layers (e.g. security, logging)

4 Design Patterns

Design patterns

Design patterns represent generally applicable solutions to commonly occurring problems.

Patterns mostly consist of (this was cemented by the GoF):

Pattern name Simple identification useful in communication

Problem Description of the problem and its *context*

Solution Solution of the problem (good practice)

Consequences Possible trade-offs of applying the pattern

4.1 GoF Design Patterns

Gang of Four Patterns

Based on the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

- Bible of design patterns
- Patterns applicable to all kinds of object-oriented software

Creational Patterns

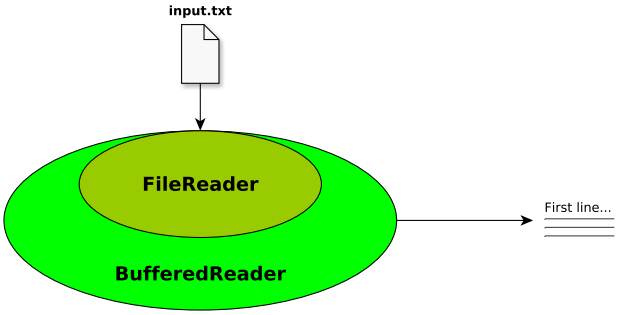
Abstract Factory Interface for creating families of related objects

Builder Instance construction process in a separate object

Factory Method Subclasses decide which class to instantiate

Prototype Build instances based on a prototype

Singleton Only one instance of the class



Structural Patterns

Adapter Convert interface of one class to a different interface using an adapter (e.g. for legacy classes)

Bridge Decouple abstraction from implementation

Composite Build tree-like structure of objects

Decorator Add or alter behaviour of another object by wrapping it in a class with the same interface (e.g. Java I/O streams)

Facade Provide a unified interface to a set of interfaces

Flyweight Use sharing to support a large number of fine-grained objects

Proxy Provide a placeholder for another object to control access to it (e.g. Spring bean proxies)

Decorator

Decorator in Java I/O

```
BufferedReader in = new BufferedReader(new FileReader(new File("input.txt")));
```

Behavioral Patterns

Chain of Responsibility Multiple objects in a chain can handle a request (e.g. request filters)

Command Encapsulate a request in an object (e.g. *undo* functionality)

Interpreter Interpreter for a language and its grammar

Iterator Provide a way to access elements of an aggregate object (e.g. Java collections)

```
Iterator<String> it = set.iterator();
```

Mediator An object that encapsulates how a set of objects interact

Memento Capture object's state so that it can be restored to this state later

Behavioral Patterns II

Observer Decoupled notification of changes of object's state

State Allows object's behaviour to change based on its internal state

Strategy A family of algorithms, which can be interchanged independently of the client

Template method Define a skeleton of an algorithm and let subclasses fill in the details

Visitor Represent an operation to be performed on the elements of an object structure

4.2 Enterprise Design Patterns

Enterprise Design Patterns

Mostly based on the book *Patterns of Enterprise Application Architecture*.

- Design patterns used especially in enterprise software
- Similarly to GoF design patterns, they originate from best practice solutions to common problems, but this time in enterprise application development

PEAA

Data Transfer Object (DTO)

- Object that carries data between processes in order to reduce the number of calls
- Useful e.g. when JPA entities are not the best way of carrying data between REST interfaces

Lazy Load

- Object does not contain all of its data initially, but knows how to load it
- Useful for objects holding large amounts of data (e.g., binary data)
- Often overused as a way of premature optimization

PEAA II

Model View Controller (MVC)

- Splits user interface interaction into three distinct roles
- Decouples UI rendering from data and UI logic
- UI implementation interchangeable

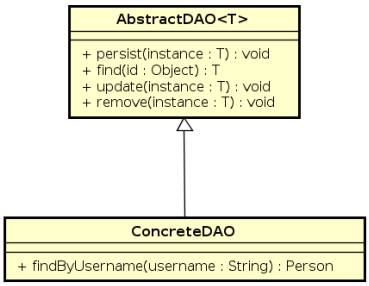


Figure 6: Common Data access object hierarchy.

Unit Of Work

- Maintains objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems
- Common in JPA implementations (e.g. Eclipselink)

4.3 Other Useful Patterns

Data Access Object (DAO)

- Data access object encapsulates all access to the data source
- Abstract interface hides all the details of data source access (data source can be a RDBMS, an external service, a linked data repository)

Inversion of Control (IoC)

- Most common when working with frameworks
- The framework takes control of what and when gets instantiated and called
- The framework embodies some abstract design and we provide behaviour in various places
- Especially important in applications which react to some client's actions
  - Be it a different application
  - or a client using your application's UI
- aka *The Hollywood Principle* – "Don't call us. We'll call you."

IoC II

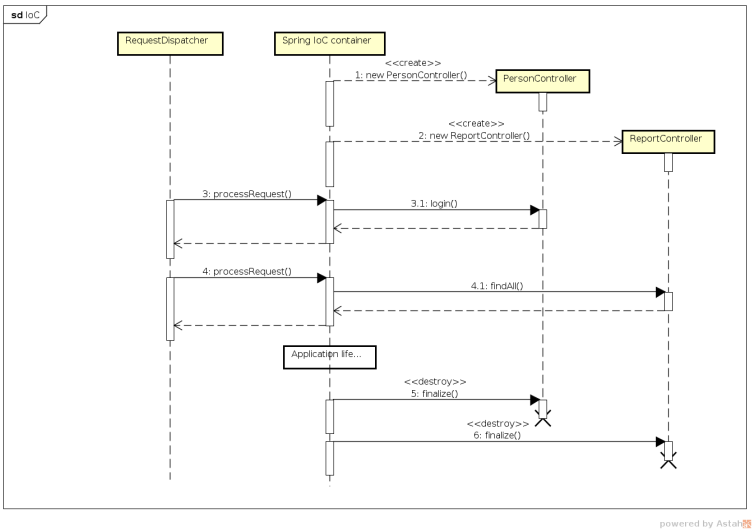


Figure 7: Inversion of Control in a Spring application.

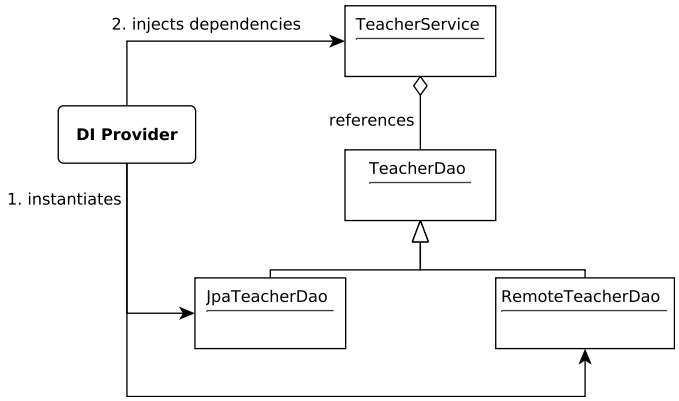


Figure 8: Dependency injection principle.

Dependency Injection

- An assembler takes care of populating a field in a class with an appropriate implementation for the target interface
- Enables the application to use loosely coupled components with interchangeable implementations

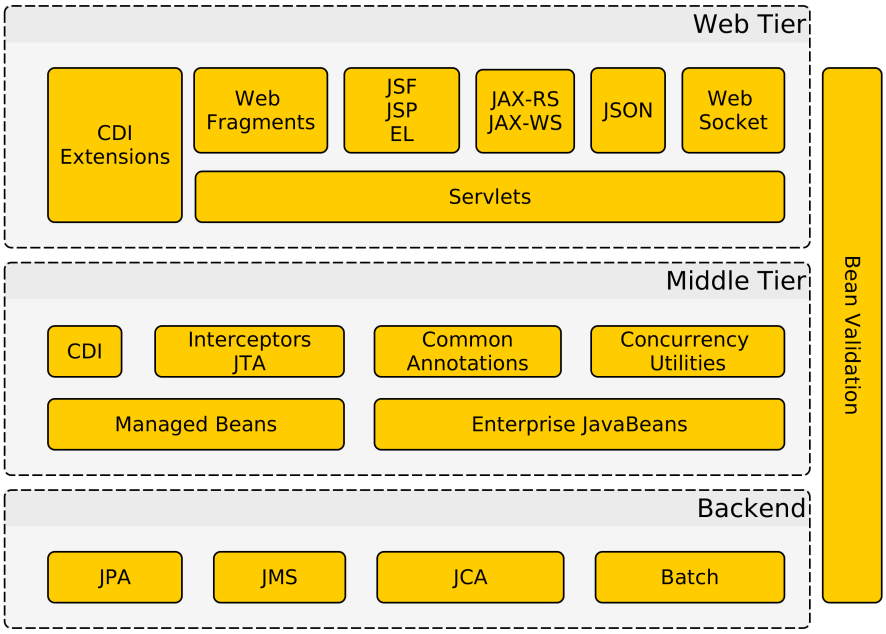
Dependency Injection II

5 Conclusions

Conclusions

- Application design **does** matter
- Architecture consists of multiple architectural styles
- Design patterns are more fine grained than architectural styles
- Web applications usually follow the layered style

Java EE = Java Enterprise Edition



The End

Thank You

Resources

- `https://hackerlife.co/blog/san-francisco-large-corporation-employee-tenure`
- E. Gamma, R. Johnson, R. Helm, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software
- M. Fowler: Patterns of Enterprise Application Architecture
- E. Evans: Domain Driven Design: Tackling Complexity in the Heart of Software
- Lectures of Tomáš Černý – A7B36ASS
- `https://msdn.microsoft.com/en-us/library/ee658098.aspx`
- `https://www.petrikainulainen.net/software-development/design/understanding-spring-web-app`