Multiuser Applications

Module 223

Developing multiuser applications in an object - oriented manner

Document information

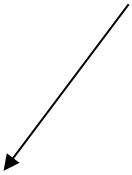
Title:	Course material
Topic:	Module 223 – Developing multiuser applications in an object oriented manner
Filename:	kursunterlagen_223_v2.docx
Storage date:	27.09.2019 11:34
Author:	Yves Kaufmann

Contents

1 Introduction to Course

1.1 Clarification of Content

Analysis and Design of object oriented applications? What about service oriented or component based?

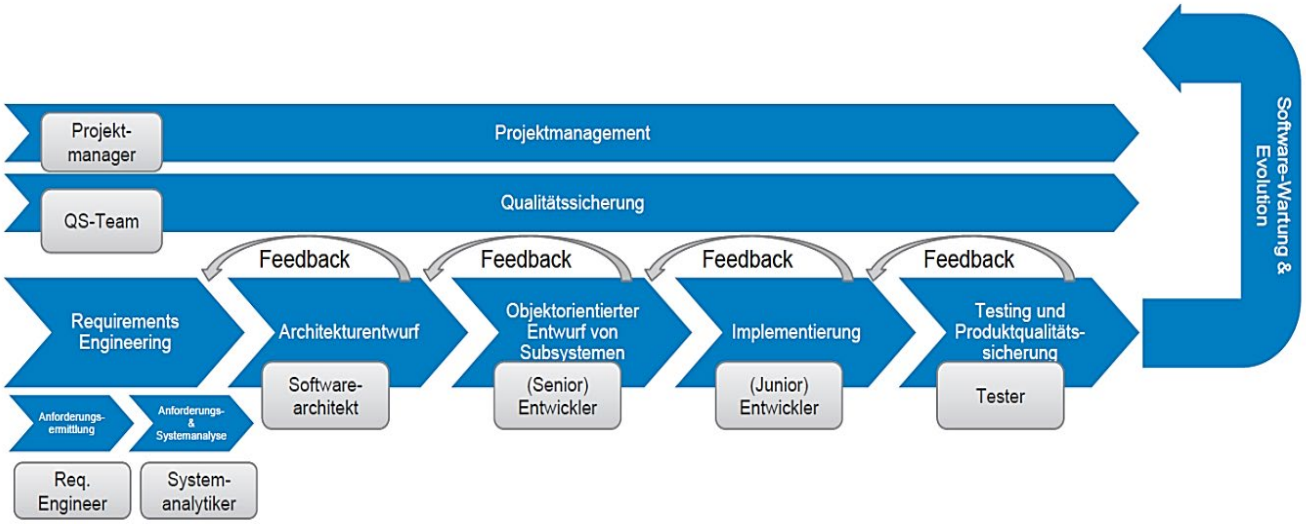


Multiuser Applications

Developing **multiuser applications in an** ***object-oriented*** **manner**

Paradigm? Imperative or declarative? Other languages? Functional? Procedural?

1.3 Overview of covered topics



1.4 Tools

IntelliJ IDEA
https://www.jetbrains.com/idea/

Spring Boot 2.0
https://spring.io/guides/gs/spring-boot/

OpenJDK 11
https://jdk.java.net/11/

Gradle 7.3
https://gradle.org/install/

Swagger
https://swagger.io/

PostgreSQL 11
https://www.postgresql.org/download/

Flyway
https://flywaydb.org/getstarted/how

git
https://git-scm.com/

Postman
https://www.getpostman.com/apps

GitKraken
https://www.gitkraken.com/download/mac

1.5 Script Versions

Version	Changes	Author(s)
1.0	Initial release	Yves Kaufmann, Santiago Gabriel Vollmar
2.0	Update	Gianluca Daffré

2 Requirements Engineering

“The hardest single part of building software is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later” (Fred Brooks 1987)

2.1 Chapter Objectives

The objectives for this chapter are as follows:

- You are familiar with the definition of terms and activities involved in requirements engineering
- You elaborated on how requirements are classified
- You are conscious of issues that might appear during the process and you are able to outline attempted solutions

2.2 Definition of Terms

Requirements engineering represents the initial phase of software engineering.

[1]Requirement analysis, or formerly known requirement engineering, describes the process of collecting, defining, documenting, validating and managing user expectations and demands for a software being built or altered with. It promotes resolution of conflict or ambiguity in requirements and ensures that the final product conforms to the client’s expectations. The analysis functions as a construct for further choices to be made during development.

The term “software requirement” implies a capability needed by a user to solve a given problem or achieve a certain objective. A requirement is essentially a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation.
A requirement ought to be clear, correct, consistent, coherent, comprehensible, verifiable, prioritized, unambiguous, traceable and credible.

[2]For the topics mentioned in the upcoming chapters, it is essential to understand what the term stakeholder implies. According to statements made by IEEE, a stakeholder is an individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations. Stakeholders include, but are not limited to end users, end user organizations, supporters, developers, producers, trainers, maintainers, disposers, acquirers, customers, operators, supplier organizations and regulatory bodies.

[1] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.727.1444&rep=rep1&type=pdf 03.03.2019
[2] https://www.gaudisite.nl/SOSE2017_Aasheim_Zhao_RequirementsDefinition.pdf 03.03.2019

2.3 Activities involved

Figure 1 Activities in requirements engineering

[3]The activities involved in requirements engineering vary widely, depending on the type of system being developed and the specific practices of the organization involved.
The individual steps outlined below are often presented as chronological stages although, in practice, there is considerable interleaving of these activities. Agile engineering for instance, supports the definition of requirements in an agile manner rather than the requirement freeze upfront which could potentially posit risks from invalidated requirements or their alternations during the development process.

- **Requirements Inception / Elicitation / Gathering**

  [4]Before requirements can be analyzed, modeled or specified they are to be gathered and discussed upon through an elicitation process. It is essential for all involving parties to have a common or similar understanding of the given domain. Most often than not, requirements have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during sessions. These cross-functional implications can be elicited by conducting interviews and workshops in a controlled environment. It is advised to nominate someone to act as a facilitator between all attendees.

- **Requirements Elaboration / Modeling / Negotation**

  Requirements are further identified and conflicts with or between stakeholders are ultimately solved. Ambiguities and uncertainties that might still exist are outlined. Issues and topics that were decided upon are usually graphically visualized and further explained by using various models or notations such as UML.

- **Requirements Specification**

  Requirements are now to be documented in a formal artefact called a requirements specification (RS). The specification may include functional and non-functional requirements and various models that describe defined user interactions with the software to be developed. A requirement specification establishes the basis for an agreement between customers, contractors and other involved parties on how the software product should function. Software requirements specification is a rigorous assessment of requirements before the more specific system design stages, and its goal is to reduce later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules. Used appropriately, software requirements specifications can help prevent software project failure.

[3] https://en.wikipedia.org/wiki/Requirements_engineering 03.03.2019
[4] https://pdfs.semanticscholar.org/6d1b/aeb3cfde1ecb8bf1b75e212e44e98edec2c5.pdf 03.03.2019

- **Requirements Validation**

  Checking that the documented requirements and models are consistent and meet the needs of the stakeholder. Only if the final draft passes the validation process, the RS becomes official. If not, the RS has to be adjusted and then examined again.

- **Requirements Management**

  Requirements management involves communication between the project team members and stakeholders and takes care of adjustments and changes to requirements throughout the course of the project. To prevent one class of requirements from overriding another, constant communication among members of the development team is critical.
  Traceability of requirements is used in managing requirements to report back fulfilment of company and stakeholder interests in terms of compliance, completeness, coverage, and consistency. Traceability also support change management as part of requirements management in understanding the impacts of changes through requirements or other related elements, and facilitating introducing these changes.

2.4 Classification of Requirements

In order to succeed in a project, it is of great importance to both the stakeholders and the project delivery teams to clearly define and agree on a project's scope and requirements. In this chapter, we elaborate on the various types of requirements and provide examples highlighting the fundamental differences. Requirements generally fall into the following types:

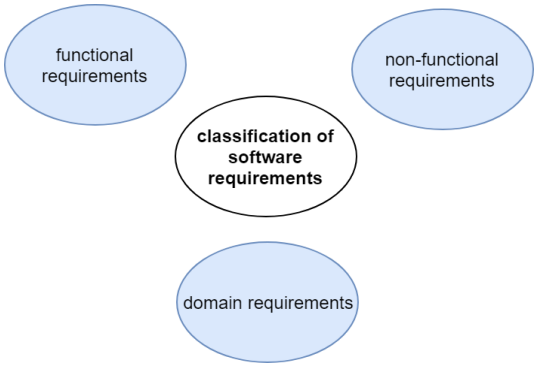


Figure 2 Classification of requirements

Functional Requirement
Often defined as an issue that specifies what a system should do, how it interacts with its environment, how it responds to inputs or how it ought to behave under certain conditions.
E.g. Data must be entered before a request can be approved.

Non-Functional Requirement
Defines an issue that specifies how the system should perform a certain task. States quality attributes and constraints of a system.
E.g. The response time for a search transaction under a peak load of 10,000 users must not exceed 2 seconds.

Domain Requirements
Reflects the environment in which the system should be embedded in. Domain requirements are essential as they often reflect fundamentals of the given domain.
E.g. The system safety shall be assured according to standard IEC 60601-1: Medical Electrical Equipment.

2.5 Potential Issues and Attempted Solutions

[5]There are various issues that might appear during the process of requirements engineering. Defining a system scope, creating a common and shared understanding among the different stakeholders affected by the development of a given system or dealing with the volatile nature of requirements might all lead to poor requirements and potential cancellations. It might even be judged unsatisfactory or unacceptable, has large maintenance costs or undergoes frequent changes. By improving requirements elicitation, the process can be improved, resulting in enhanced system requirements and potentially a significantly better system.

[6]Issues that might occur can be decomposed in the following topics:

Problem of scope

The boundaries of the system are ill-defined. Unnecessary design information may be given.

Problem of understanding

Stakeholders have incomplete or poor understanding of the given domain
Stakeholders are technically unsophisticated
Stakeholders often do not participate in reviews or are incapable of doing so
Stakeholders do not understand the development process
Stakeholders do not know about present technology
Stakeholders and technical personnel may have different vocabularies
Communication with Stakeholders is slow

Problem of volatility

Stakeholders will not commit to a set of written requirements
Stakeholders insist on new requirements after the cost and schedule have been fixed
Stakeholders do not understand what they want, or Stakeholders don't have a clear idea of their requirements

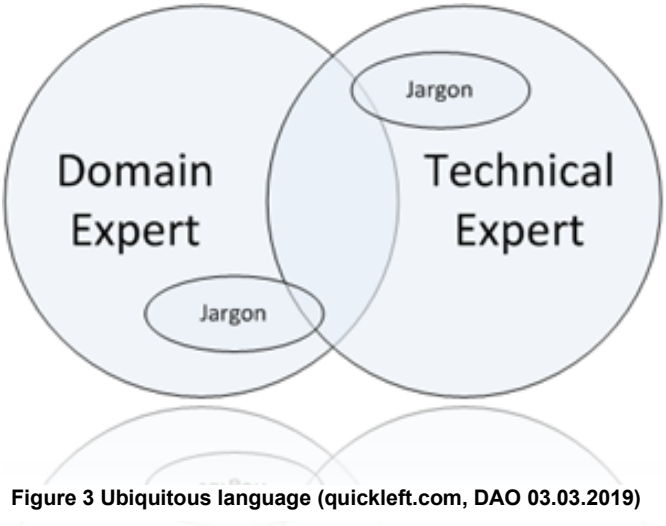


Figure 3 Ubiquitous language (quickleft.com, DAO 03.03.2019)

[5] https://www.computer.org/csdl/proceedings/hicss/2010/3869/00/09-08-06.pdf 03.03.2019
[6] https://www.erpublication.org/published_paper/IJETR022720.pdf 03.03.2019

This may lead to the situation where user requirements keep changing even when system or product development has been started. To be able to better understand and interact with the given stakeholders, Suzanne Robertson classifies requirements in her book “Mastering The Requirements Process” as follows:

Conscious Requirement
Defined as a requirement that the client is consciously aware of.

Unconscious Requirement
An unconscious requirement is a specific issue that the client desires, but for some reason did not explicitly express or forgot to elicit, assuming that it will be taken care of. Those requirements are often challenging as they tend to come to the fore after the application has been developed.

Undreamt Requirement
Usually a topic that clients generally not mention since they assume that such requirements are impossible to achieve. Undreamt requirements are often difficult to capture and take notice of. Most often than not, requirements engineers are the ones that bring such topics to the surface.

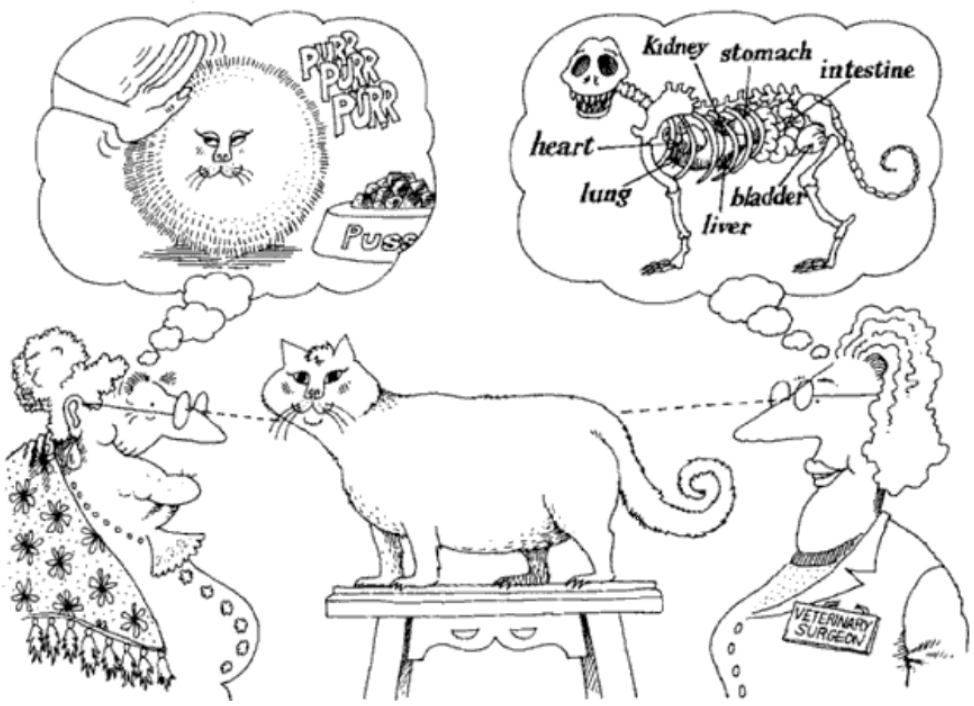


Figure 4 The issue of perception (Object-Oriented Analysis and Design with Applications Gary Booch)

3 Object Oriented Analysis and Design OOAD

“The analysis model will not be a reflection of what the problem domain looks like. The reason is simply to get a more maintainable structure where changes will be local and thus manageable. We do not model reality as it is, as object orientation is often said to do, but we model the reality as we want to see it and to highlight what is important in our application.” (Ivar Jacobson 2006)

3.1 Chapter Objectives

The objectives for this chapter are as follows:

- You are familiar with the definition of terms in OOAD
- You made acquaintance with the three amigos
- You elaborated on the modeling language UML and know how its diagrams are used in OOA as well as OOD

3.2 Definition of Terms

[7]OOAD represents a technical approach to analyze and design software. It uses visual modeling throughout the development life cycles to grant better communication and shared knowledge among all parties involved in the process. OOAD is best conducted with iterative and incremental methodologies. Models constructed in OOA or OOD respectively will be refined and evolve continuously over the course of the project.

[8]The purpose of object-oriented analysis OOA is to create a model visualizing mostly functional requirements independent of any implementation constraints. The OOA approach organizes requirements around objects that the system interacts and communicates with.
Object-oriented analysis usually takes place in the elaboration and modelling phase of requirements engineering. Common visualizations and techniques used in OOA are use cases, domain models, prototypes or other conceptual approaches.
Grady Booch has defined OOA as, “Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.”

During the process of OOD, implementation constraints are applied to the conceptual models previously created in OOA. Implementation independent concepts of OOA are now mapped onto classes and interfaces resulting in a model of the solution domain. It defines how the system is to be built on concrete technologies. Object-oriented design takes place after requirements engineering. Grady Booch has cited object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.”

OOM, which stands short for object-oriented modeling, is a common approach in the object-oriented paradigm to model applications, systems and business domains throughout the development life cycles. OOM is a technique heavily used by both OOD and OOA. The Unified Modeling Language (UML) is one of the most popular international standard languages used in OOM. It promotes efficient and effective communication and helps to share knowledge among all parties involved in development.

[7] https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design#Overview 03.03.2019
[8] https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.htm 03.03.2019

3.3 The Three Amigos – Founding Fathers of UML

[9]The company Rational Machines was founded by Paul Levy and Mike Devlin in the year of 1981 to provide tools to expand the use of modern software engineering practices, particularly explicit modular architecture and iterative development.
Grady Booch was employed as a chief scientist at Rational, mainly working on graphical notations. Rational Software Corporation then hired James Rumbaugh from General Electric in 1994. The company essentially became the source for the two most popular object-oriented modeling approaches of the day, namely OMT and Booch method. Together Rumbaugh and Booch attempted to reconcile their two approaches and started work on a unified method. Last but not least Rational acquired Ivan Jacobson's Objectory company. The three methodologists were collectively referred to as the three amigos, since they were well known to argue frequently with each other regarding methodological preferences. Rational was sold for $2.1 Billion to IBM on February 20, 2003. Eventually OMG took ownership.

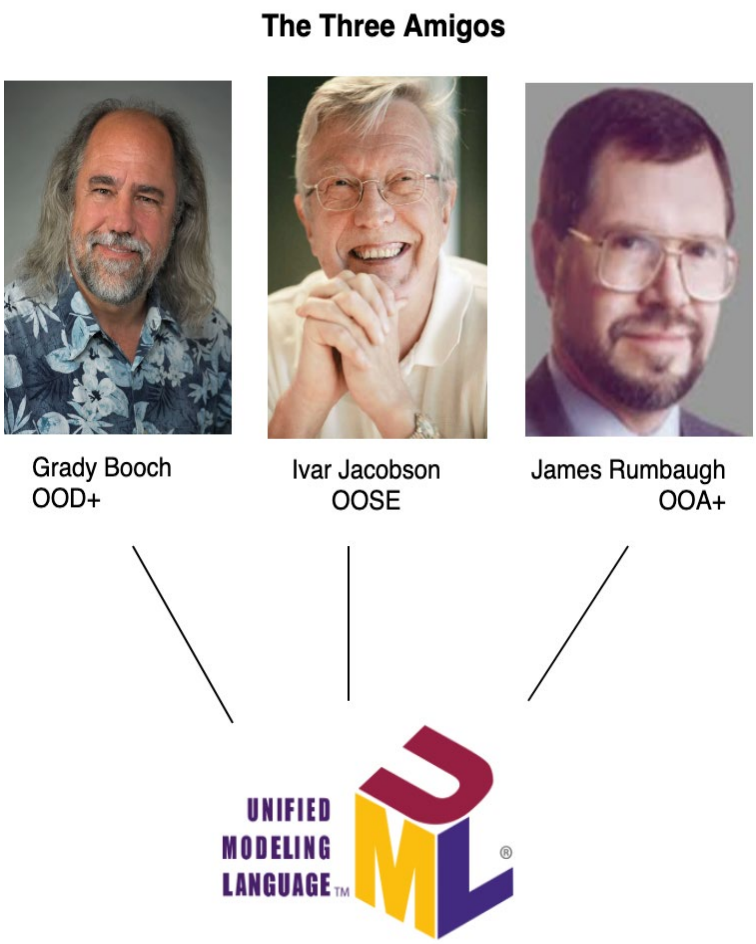


Figure 5 Founders of UML

Recommended lectures:

Grady Booch: Booch method – suited towards OOD
Object-Oriented Analysis and Design with Applications, 2nd ed. ISBN 0-8053-5340-2

James Rumbaugh: OMT – suited towards OOA
Object-Oriented Modeling and Design. ISBN 0-13-629841-9

Ivar Jacobson: OOSE method – first object-oriented design methodology with use cases
Object-Oriented Software Engineering: A Use Case Driven Approach. ISBN 0-20-154435-0

[9] https://en.wikipedia.org/wiki/Rational_Software 03.03.2019

3.4 Classification of UML diagrams

[10]The unified modelling language is a general-purpose, conceptual and developmental modeling language in the field of software engineering. It functions as a graphical representation of a model of a system under design or facilitates the process of visualizing concerns during the actual implementation.

[11]UML diagrams can be decomposed into two different views:

Dynamic (or behavioural) view:
Emphasizes the behaviour, interaction and collaborations among objects within a system being modelled. Extensively used to describe and visualize functionality. Interaction diagrams, a subset of behaviour diagrams, focus on the flow of control and data within the system.
Sequence, use case or activity diagrams are only a few dynamic models to be mentioned.

Static (or structural) view:
Structural modeling captures and emphasizes the static features of a system. Hence, class diagrams or object diagrams are static models.

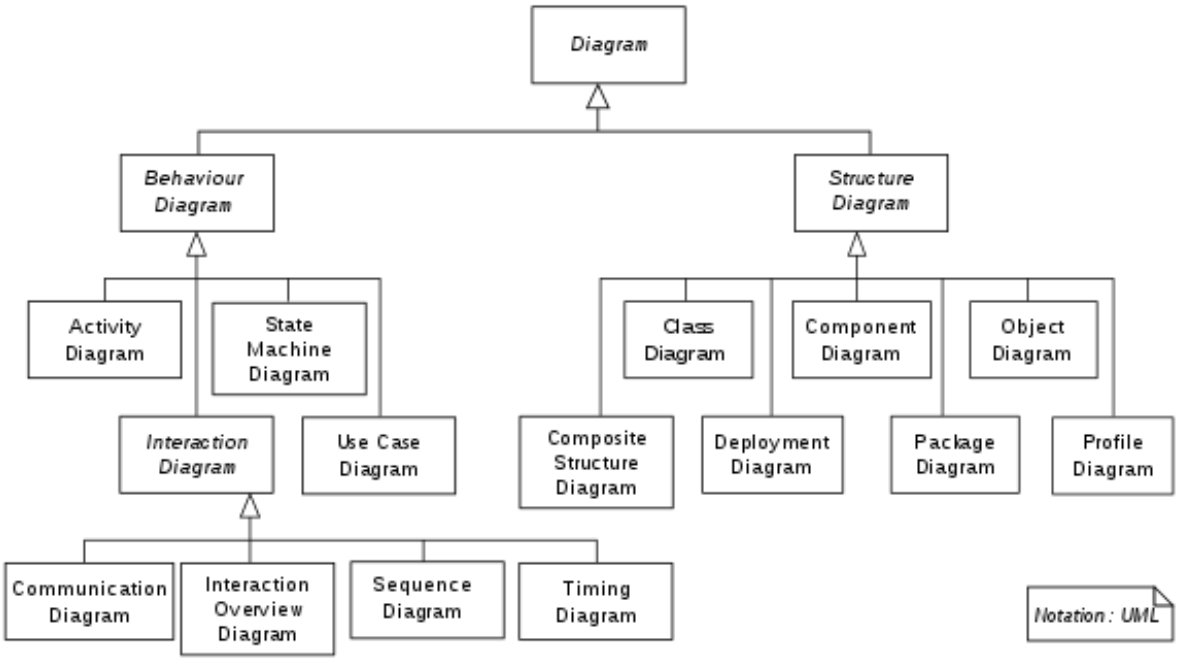


Figure 6 Classification of UML diagrams (www.wikipedia.com, DAO 04.03.2019)

[10] https://en.wikipedia.org/wiki/Unified_Modeling_Language#cite_note-:1-1 04.03.2019
[11] https://www.tutorialspoint.com/uml/uml_modeling_types.htm 04.03.2019

3.5 UML Class Diagram

The unified modelling language can help you model systems in various ways. You might be familiar with the class diagram. It is fairly popular in OOAD processes to illustrate the functionality, attributes and relationships (including association, inheritance, aggregation and composition) of classes. The perspective determines the amount of detail necessary and the relationships worth presenting. Conceptual class diagrams often function as domain models in OOA. As OOD focuses on the actual implementation, the third scope is appropriate.

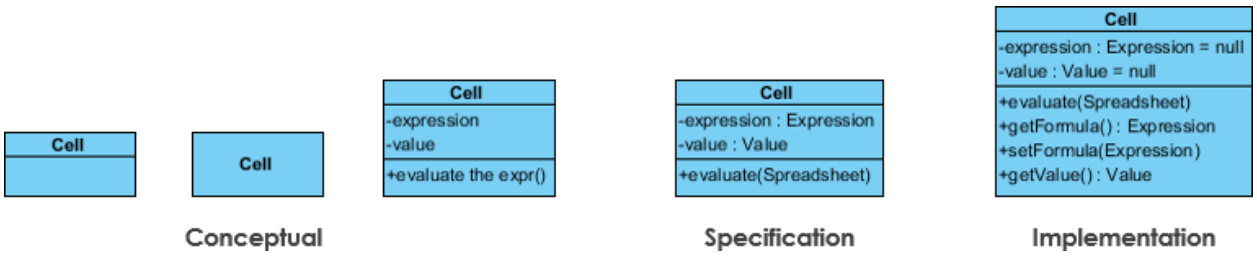


Figure 7 Different perspectives of UML class diagrams (visual-paradigm.com, DOA 01.01.19)

[12]The name of the class appears in the first, the attributes in the second and the methods in the third partition. The return type of method parameters is shown after the colon following the parameter name. Access modifiers are implied by + public, - private as well as # protected. Each parameter in a method signature may be marked as in, out or inout which specifies if the given parameter is called by value or by reference.

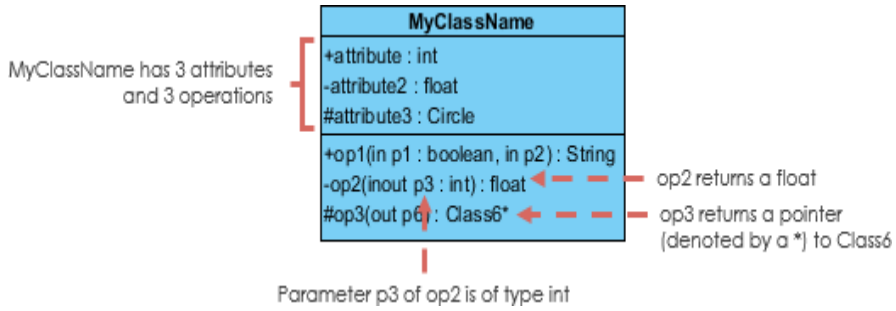


Figure 9 UML class diagram relationships and cardinalities (visual-paradigm.com, DOA 01.01.19)

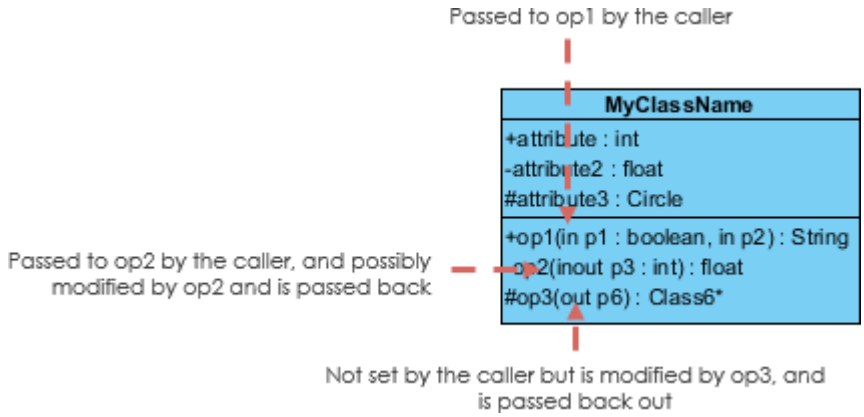


Figure 10 UML class diagram parameter directionality (visual-paradigm.com, DOA 01.01.19)

[12] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram 04.03.2019

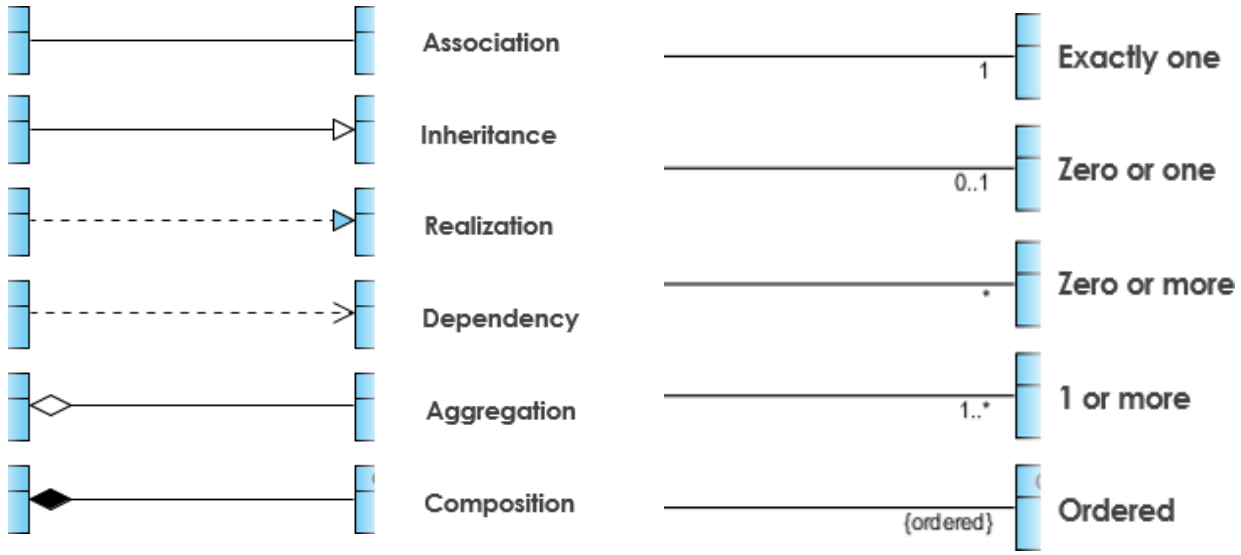


Figure 11 UML class diagram relationships and cardinalities (visual-paradigm.com, DOA 01.01.19)

Association

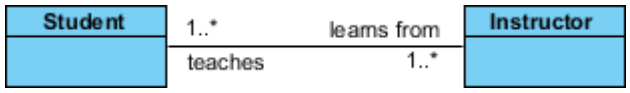


Figure 12 UML class diagram association (visual-paradigm.com, DOA 01.01.19)

A broad term that encompasses a logical connection or relationship between two given classes. They often come with a cardinality and a verbose description on either side.

Inheritance (Generalization / Specialization)

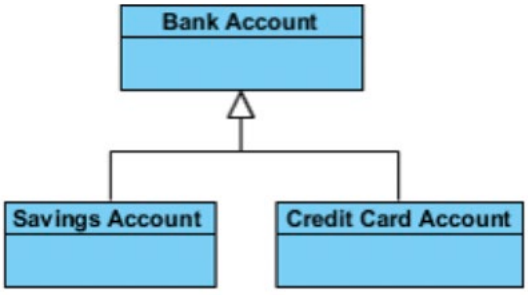


Figure 13 UML class diagram inheritance (visual-paradigm.com, DOA 01.01.19)

Refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To show inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

Aggregation

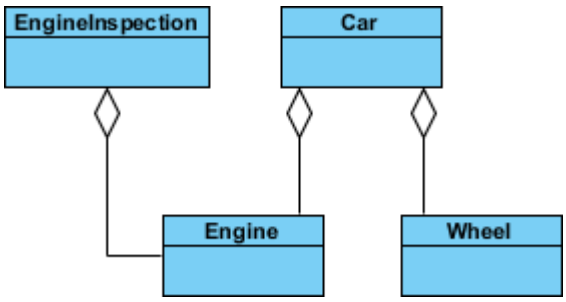


Figure 14 UML class diagram aggregation (visual-paradigm.com, DOA 01.01.19)

It is important to note that the aggregation link doesn't state that Class A owns Class B nor that there's a parent-child relationship, which would imply that if a parent was deleted all its children would be removed as well. Aggregation link is usually used to stress the point that the instance of class A is not the exclusive container of Class B.

Composition

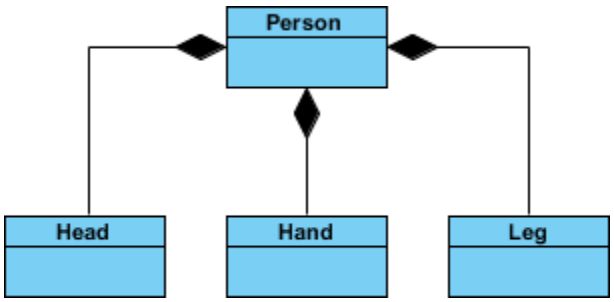


Figure 15 UML class diagram composition (visual-paradigm.com, DOA 01.01.19)

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also removed as a result.

Association vs. Aggregation vs. Composition

Aggregation and composition represent subsets of an association. Both aggregation and composition visualize a relation between class A and class B, but there is a subtle difference:

Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

3.6 UML Use Case Diagram and Specifications

“Use cases contain the rules that specify how and when the Critical Business Rules within the Entities are invoked. Use cases control the dance of the Entities.
Notice also that the use case does not describe the user interface other than to informally specify the data coming in from that interface, and the data going back out through that interface. From the use case, it is impossible to tell whether the application is delivered on the web, or on a thick client, or on a console, or is a pure service.” (Robert C. Martin)

[13]In 1992 Ivar Jacobson published one of his works called Object-Oriented Software Engineering – A Use Case Driven Approach (In fact, his employees did most of the work and he took credit for it). Anyways, the upcoming of a jacobson use case centric approach was essentially responsible for an invasion of consultants entering the market. The specifications became subsequently more elaborate resulting in primary actors, secondary actors, post conditions and so forth.

Modeling and specifying use cases has been adopted as an approach in OOA/(D) to identify, clarify and organize system requirements. A use case consists of a set of possible sequences of interactions between defined systems and actors representing a certain role. Each use case needs to be explicitly described in a textual specification. Use cases are text documents and not diagrams. Use case modeling is primarily an act of writing text, not drawing diagrams.

In Craig Larman’s belief, use cases can be broken down in three different levels of formality. In his book “UML and Patterns” he decomposes and categorizes use cases as such:

Much has been written about use cases, and though worthwhile, creative people often obscure a simple idea with layers of sophistication or over-complication. It is usually possible to spot a novice use-case modeler by an over-concern with secondary issues such as use case diagrams, use case relationships, use case packages, and so forth, rather than a focus on the hard work of simply writing the text stories.

Brief

Informally, use cases are text stories of some actor using a system to meet goals. Here is an example brief format use case:

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Use cases often need to be more detailed or structured than this example, but the essence is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, cases of use.

[13] https://en.wikipedia.org/wiki/Ivar_Jacobson 04.03.2019

Casual

A use case is a collection of related success and failure scenarios that describe an actor using a system to support a goal. For example, here is a casual format use case with alternate scenarios to handle returns:

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item …

Alternate Scenario: If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash. If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

Fully dressed

Specification for a standard use case

Use Case:	A unique title. In most cases a noun plus a verb is suitable.
Use Case ID:	A unique identification / ID.
Description:	A short description of the main functionality.
Precondition:	Specification of all conditions that have to be met before the use case can successfully start.
Actors (Primary):	A list of actors that actively interact with a use case.
Actors (Secondary):	A list of actors that transfer information to a use case or receive information from a use case but don't directly interact with it.
Procedure: A list of all individual steps mentioned in the use case. A step always starts with an actor and ends with the actual functionality, e.g. 1. <Actor> <Functionality> 2. … Possible errors that may occur during the procedures are to be neglected. It is only a visualization of the happy path.	
Postcondition:	Specification of all changes/states that occur after the execution of the use case. E.g. 1) The order is now labelled as confirmed.
Alternative flows: A list of all alternative flows.	

Specification for a use case with a FOR loop

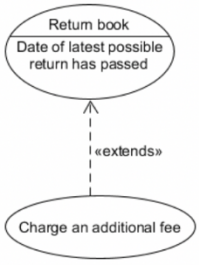
Use Case:	Find product.
Use Case ID:	3
Description:	System finds products based on the clients search criteria.
Precondition:	None
Actors (Primary):	Client
Actors (Secondary):	None
Procedure: 1. Use case starts as the client selects "search product" 2. The system asks the client for his search criteria 3. The client selects specific search criteria 4. The system searches products that meet the clients search criteria 5. FOR each product found 5.1 The system shows the summary of products 5.2 The system shows the price of each product	
Postcondition:	None
Alternative flows: None	

Specification for a use case with an WHILE loop and an alternative flow

Use Case:	Create account.
Use Case ID:	5
Description:	The system creates a new account.
Precondition:	None
Actors (Primary):	Client
Actors (Secondary):	None
Procedure: 1. The use case starts as the client selects “create new account” 2. WHILE the user input is invalid 2.1 The system asks the client to reconsider his input 2.2 The system checks the input of client 3. The system creates new account	
Postcondition:	Specification of all changes/states that occur after the execution of the use case. E.g. 1) The order is now labelled as confirmed.
Alternative flows: invalidEmailAddress invalidPassword	

Alternative flow:	Create account: invalidEmailAddress
ID:	5.1
Description:	The system informs the client about his invalid email input.
Precondition:	The client has given an invalid email input.
Actors (Primary):	Client
Actors (Secondary):	None
Alternative flow: 1. The alternative flow starts after step 2.2 in the main procedure 2. The system informs the client about the fact that his email input is invalid	
Postcondition:	None

Specification of two use cases in an extend relationship



Use Case:	Return book.
Use Case ID:	12
Description:	The librarian returns a borrowed book.
Precondition:	The librarian is logged in.
Actor (Primary):	Librarian
Actor (Secondary):	None
Procedure: 1. The librarian provides the id of the client. 2. The system shows the clients settings and a list of all borrowed books. 3. The librarian finds the returned book in the list of borrowed books. extension point: Date of latest possible return has passed. 4. The librarian returns the book. 5. …	
Postcondition:	The book is labeled as returned.
Alternative flow: None	

Extension Use Case:	Charge an additional fee.
ID:	13
Description:	The librarian charges an additional fee for the book to be returned late.
Precondition:	The date of latest possible return has passed.
Actor (Primary):	Librarian
Actor (Secondary):	None
Procedure: 1. The librarian charges an additional fee.	
Postcondition:	1. The additional fee is entered in the system. 2. The system has printed out a report.

Specification of two use cases in an include relationship

Use Case:	Change settings of employee.
Use Case ID:	7
Description:	The employer changes settings of employee.
Precondition:	The employer is logged in.
Actor (Primary):	Employer
Actor (Secondary):	None
Procedure: 1. Include (Find settings of employee) 2. The system shows settings of employee. 3. The employer changes the settings of the employee. 4. …	
Postcondition:	The settings of the employee changed.
Alternativer flows: None	

Use Case:	Find settings of employee.
Use Case ID:	9
Description:	The employer finds the settings of the employee.
Precondition:	The employer is logged in.
Actor (Primary):	Employer
Actor (Secondary):	None
Procedure: 1. The employer provides the id of the employee. 2. The system finds the settings of the employee.	
Postcondition:	The system has found the settings of the employee.
Alternative flow: None	

The following use case diagram visualizes the check-in process at the airport. Individual elements and their descriptions are outlined below:

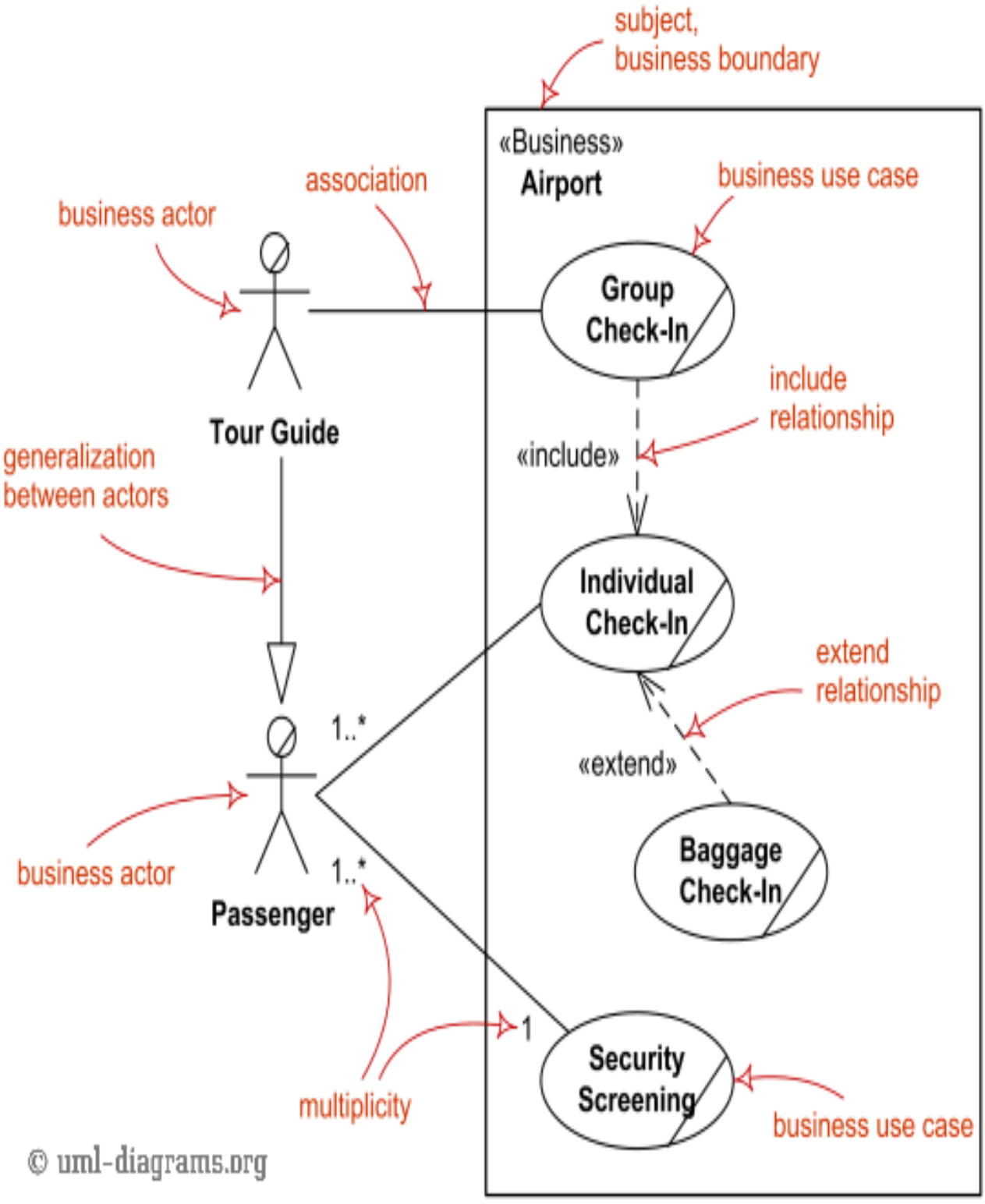
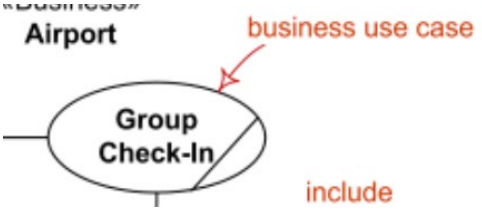
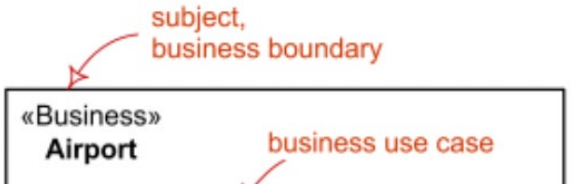


Figure 16 Use case diagram (uml-diagrams.org, DOA 04.03.2019)

[14]An actor models a type of role played by an entity that interacts with the system. Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., "role") of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances.

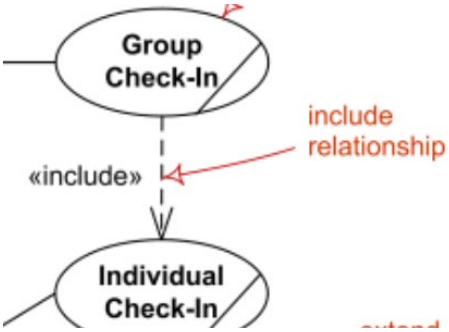
A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

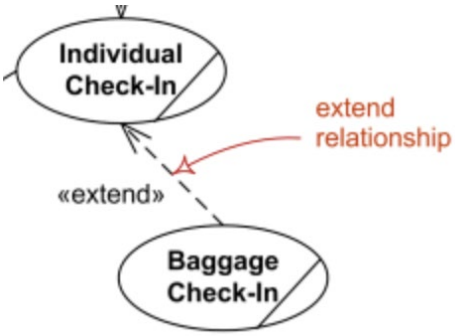




Defines the boundaries of a specific system or subject.

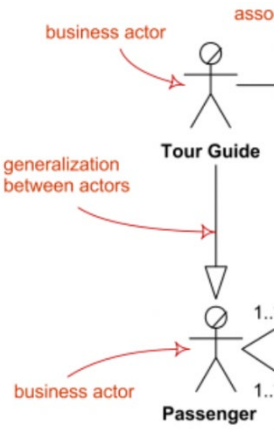
The relation include visualizes that a use case A imports the behaviour of a use case B. The imported use case will therefore always be executed.



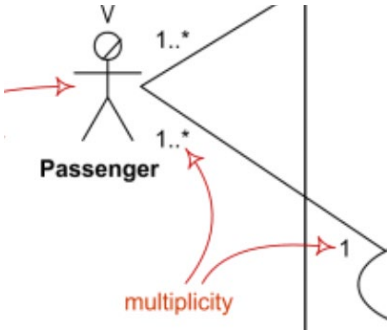


The relation extend visualizes that the behaviour of a use case A could get extended by a use case B but does not necessarily have to. A use case can hold multiple extension points.

[14] https://en.wikipedia.org/wiki/Actor_%28UML%29 04.03.2019



A generalization implies that all use cases concerning the abstract actor tour guide also relate to the subclass passenger. Individual check-in as well as security screening can only be conducted by the actor passenger.



Allowable number of instances of described item. Sets a none-negative lower and (possibly infinite) upper bound.

3.7 Domain Model and Use Cases Combined

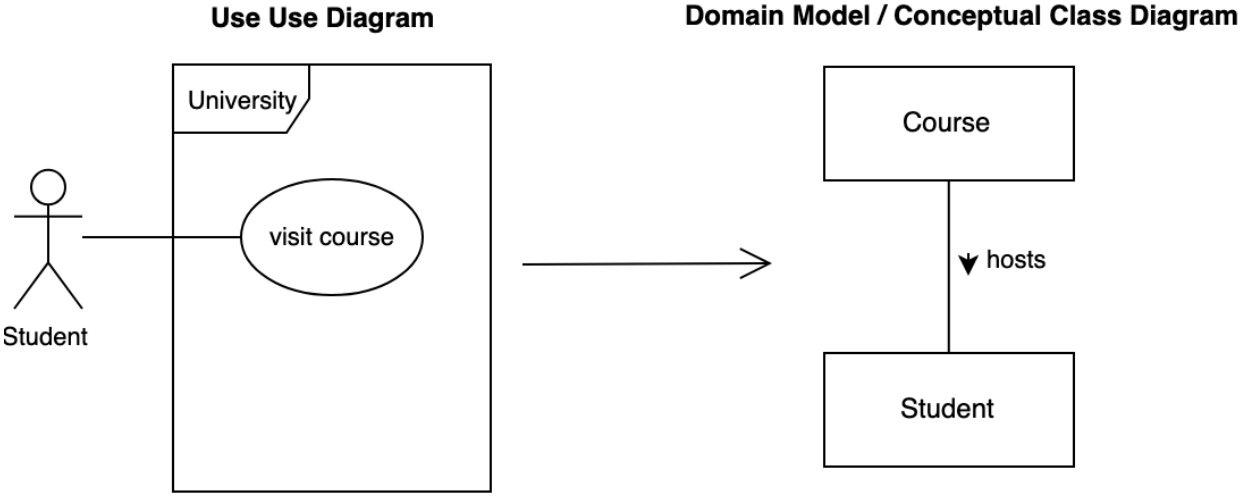


Figure 17 UML use case into domain model

3.8 Entity Relationship Diagram (ERD) vs. Relational Model (RM)

It is of great importance to understand the characteristics of an Entity Relationship Diagram (ERD) and its key differences to a Relational Model (RM). Even though both approaches contain entities and define relationships, they not only fundamentally differ in their purpose but are also targeted towards different audiences.

Figure 18 Dr. Peter Pin-Shan Chen (https://www.csc.lsu.edu/~chen/, DOA 01.01.19)

Sometimes you have to be conscious of what happened in the past in order to understand the present. Doctor Peter Pin-Shan Chen, a Taiwanese American computer scientist, popularized the entity relationship modelling approach in the year of 1976. Chen's initial paper is commonly cited as the definitive reference for entity relationship modelling though the concept of object relationship had been invented and developed a year earlier by Schmid and Swenson. Not only is his achievement and work considered a great cornerstone in software engineering but also serves as a foundation in object oriented analysis and design methodologies. It also greatly influenced the upcoming of the semantic web as well as the UML modelling language.

Entity relationship modelling is considered conceptual. It is advised to create a conceptual model at the very beginning of a project to ensure that all participants have an identical understanding of all coherences in a specific field of work. Conceptual modelling is an approach to better describe and model real world entities and define relationships between them. This helps solving the gap between the understanding of a certain problem domain and its interpretation. Not only that but it also makes the process of resolving countless ambiguities on both the requirements and the design intent easier. It incorporates representations of both behaviour and data the same time.

An entity relationship diagram is a graphical representation of an entity relationship modelling approach and is mostly used as a conceptual foundation for a relational database design. It is highly abstract and independent of all physical considerations (e.g. DBMS, variable data types). Individual elements are listed below.

Figure 19 ERD entity

Entity: Real-world object with an independent existence and distinguishable from other objects.

Attribute: An entity is represented by a set of attributes, which function as descriptive properties. An underlined attribute implies a unique identifier, or formerly known, a Primary Key (PK). The illustrated example on your right defines its combined identifier as such: {Surname, Forename}.

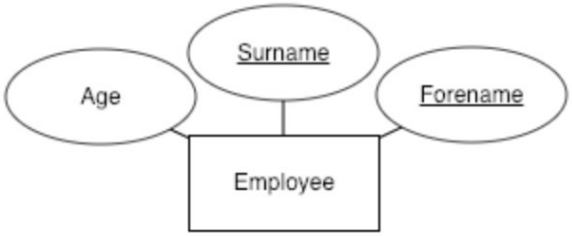


Figure 20 ERD attributes

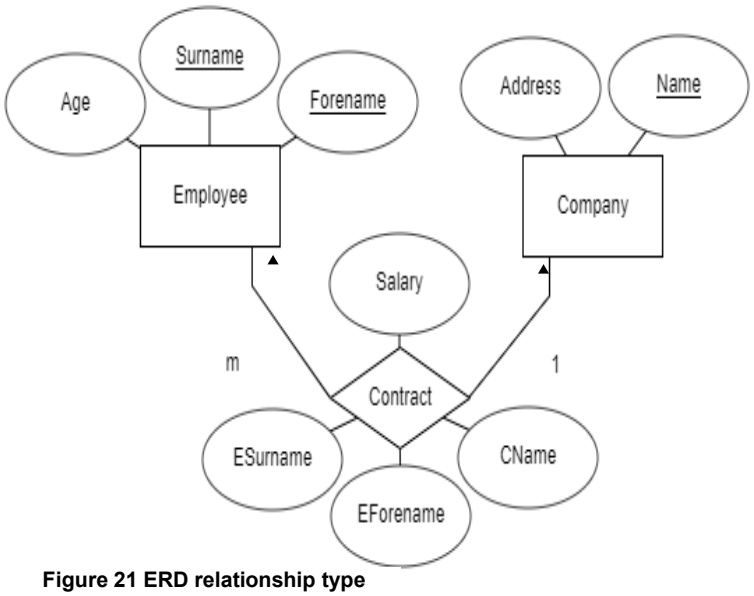


Figure 21 ERD relationship type

Relationship: An association among two or more entities. The relationship type “Contract” inherits the primary key attributes of the given entities “Employee” and “Company”.
If desired, additional attributes can be added. Unless relationship types are referenced, they do not define primary keys but keys. In a relational model the key of “Contract” is to be defined as {ESurname, EForename}. The entities “Employee” and “Company” are independent. However, the relationship type “Contract” is dependent as he references the entities “Employee” as well as “Company”.

Although most properties of entities and relationships can be expressed using the basic modeling constructs, some of them are costly and difficult to express. Let us look into some more advanced entity relationship diagram elements.

In some cases, an entity has numerous subgroupings of its entities that are meaningful and need to be represented explicitly based on their significance. This is modelled and visualized by an ISA dependence. The stated attributes of the given superclass are inherited to its subclasses. As seen below, the primary key of “Business Associate” is explicitly enforced on the subclasses “Client” as well as “Supplier”. This ensures that both the “Client” as well as the “Supplier” theoretically represent a “Business Associate”. If desired, the subclasses can be extended by additional attributes.
An ISA relation is both used as specialization as well as generalization. “Client” and “Supplier” are generalized to “Business Associate” as either of the entities defines an attribute “Address”. “Client” and “Supplier” are specialized as either of the entities defines individual attributes “Ordered” as well as “Delivered”.

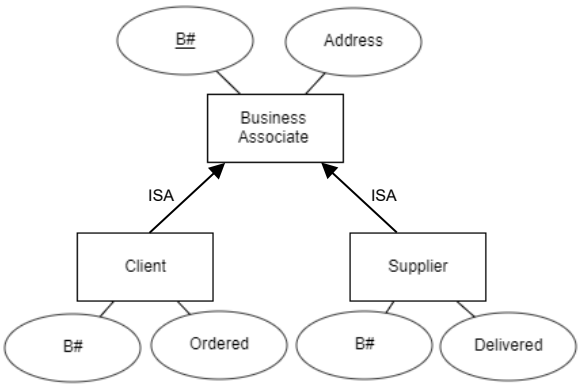


Figure 23 ERD ISA-dependent entity type

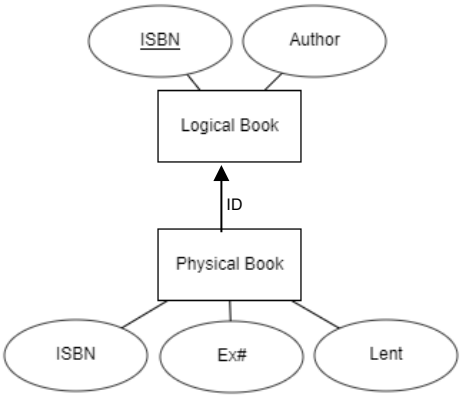
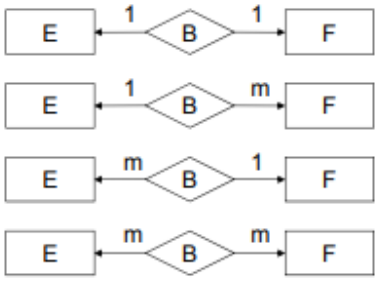


Figure 22 ERD ID-dependent entity type

Last but not least, an ERD also defines ID-dependent entity types.
The entity “Physical Book” is ID-dependent of its parent “Logical book”. To be capable of differentiating between individual examples of “Physical Book”, an additional attribute “Ex” has to be added. {ISBN, Ex} now functions as a combined key.

The following table visualizes the definition of the key in non-referenced relationship types:



Inherits keys of E and F

Inherits keys of F

Inherits keys of E

Inherits keys of E combined with keys of F

Figure 24 Keys of non-referenced ERD relationship types

To further summarize the key points of the knowledge you have acquired in this chapter so far, let us have a glance at the following points:

The entity relationship modelling is a conceptual design approach that follows requirements analysis and yields a high-level description of data. Constructs are expressive, close to the way people think about applications. An entity relationship diagram functions as a graphical representation of the above. Its elements are entities, relationships and attributes. For rather difficult abstractions, ID and ISA dependent entities can be included.

In the above examples, the Chen notation was applied. Nevertheless, you should also be familiar with other ERD notations such as Crow’s Foot or UML.

Now that we are conscious of how an entity relationship diagram is visualised, let us go further ahead and outline its key differences to a logical and physical relational model.

An entity Book in its 3 different stages of design:

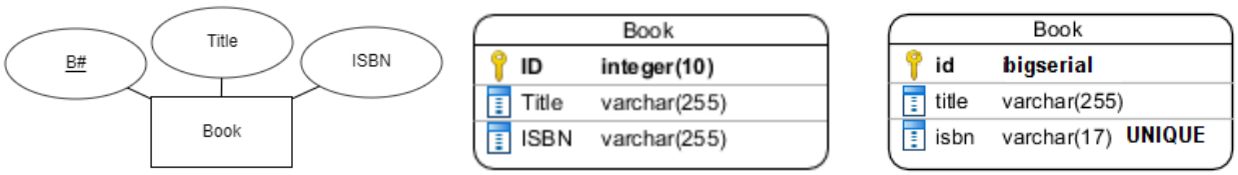


Figure 25 ERD vs logical RM vs physical PostgreSQL RM

A logical relational model is created at the requirements gathering and functions as a top level design, communication and specification instrument. It describes the data as detailed as possible, without concern about the actual physical implementation. Primary and foreign keys are explicitly defined, and normalization occurs.

The physical relational model however, is drawn when you translate top level design approaches into actual tables for a further implementation in some sort of a data layer. It shows all table structures, including column names, column data types and column constraints.

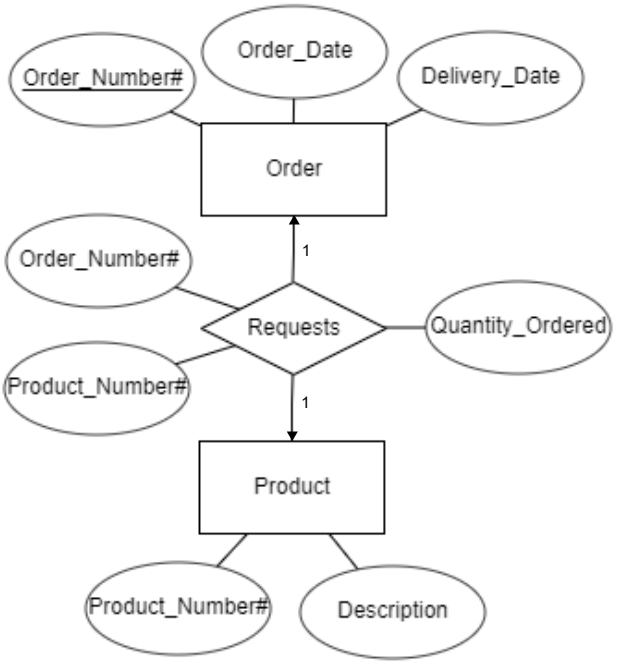


Figure 26 Conceptual ERD

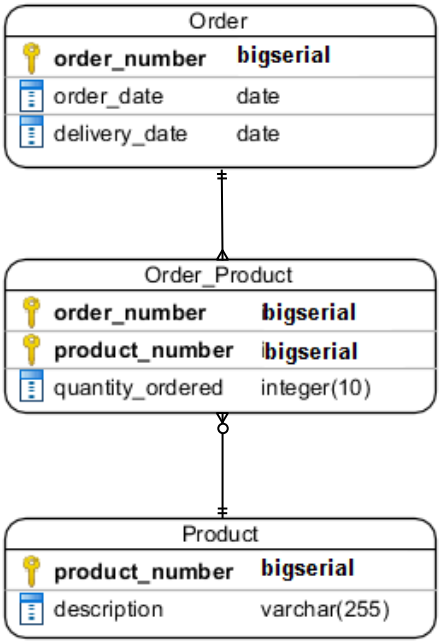


Figure 27 Physical PostgreSQL RM

4 Architecture and Design Patterns

“is not the language that makes programs appear simple. It is the programmer that make the language appear simple!” (Robert C. Martin Clean Architecture & Clean Code)

4.1 Chapter Objectives

The objectives for this chapter are as follows:

- You elaborated on MVC as well as MVP and know its strengths and weaknesses
- You are familiar with client-server and peer-to-peer architectures
- You read upon multi-tier and multi-layered web applications

4.2 Architecture vs. Design

Before we can dive into the section of tooling and developing code, we have to understand the fundamental differences between software architecture and software design since these terms are closely linked and therefore often cause confusion.

Suppose government wants to build a new city. First, they will create a high-level abstract diagram explaining where the housing area and the factory will be, where parks should be located, where to locate grid systems, hospitals, schools, markets and so on. This is Architecture. Now after that they will create diagrams explaining how houses are built, how much maximum area a single house can have. How to build houses side by side. How much wider the school front would be. How many beds and rooms the hospital would have. This is Design.

An architecture pattern is a solution to a reoccurring problem. They have an extensive impact on the code base, most often impacting the whole application either horizontally or vertically. Design patterns differ from architectural patterns in their scope. They are more localized and have less impact on the code base as they only impact a specific section.

Possible architecture patterns are: MVC, MVP, MVVM, client-server, peer-to-peer

Possible design patterns are: Singleton, factory pattern, observer pattern

4.3 MVC

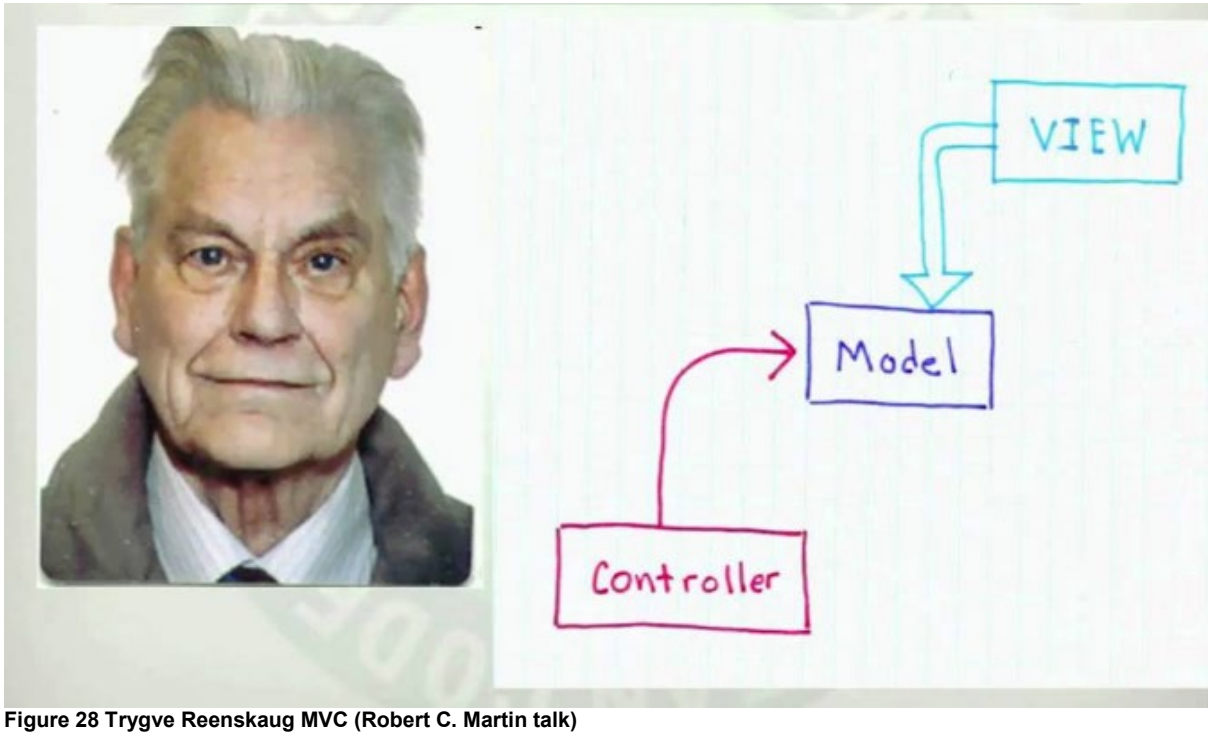


Figure 28 Trygve Reenskaug MVC (Robert C. Martin talk)

[15]Trygve Reenskaug introduced MVC in 19070s. The architecture pattern model view controller has subsequently evolved resulting in other variants such as MVP, MVVM or other that adapted MVC to different contexts.

The MVC pattern shows low coupling, enables simultaneous development and makes modification rather simple. However, code navigability can be hard. Interfaces promote boilerplate. Rather pronounced learning curve to execute MVC in an adequate manner.

Model
The application's dynamic store of data, independent of UI or other view related concerns.

View
Representation of data. Gets informed by the model through some kind of observer like pattern if changed occurred.

Controller
Accepts and validates user inputs and performs interactions with the data model.

[15] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller 05.03.2019

4.4 MVP

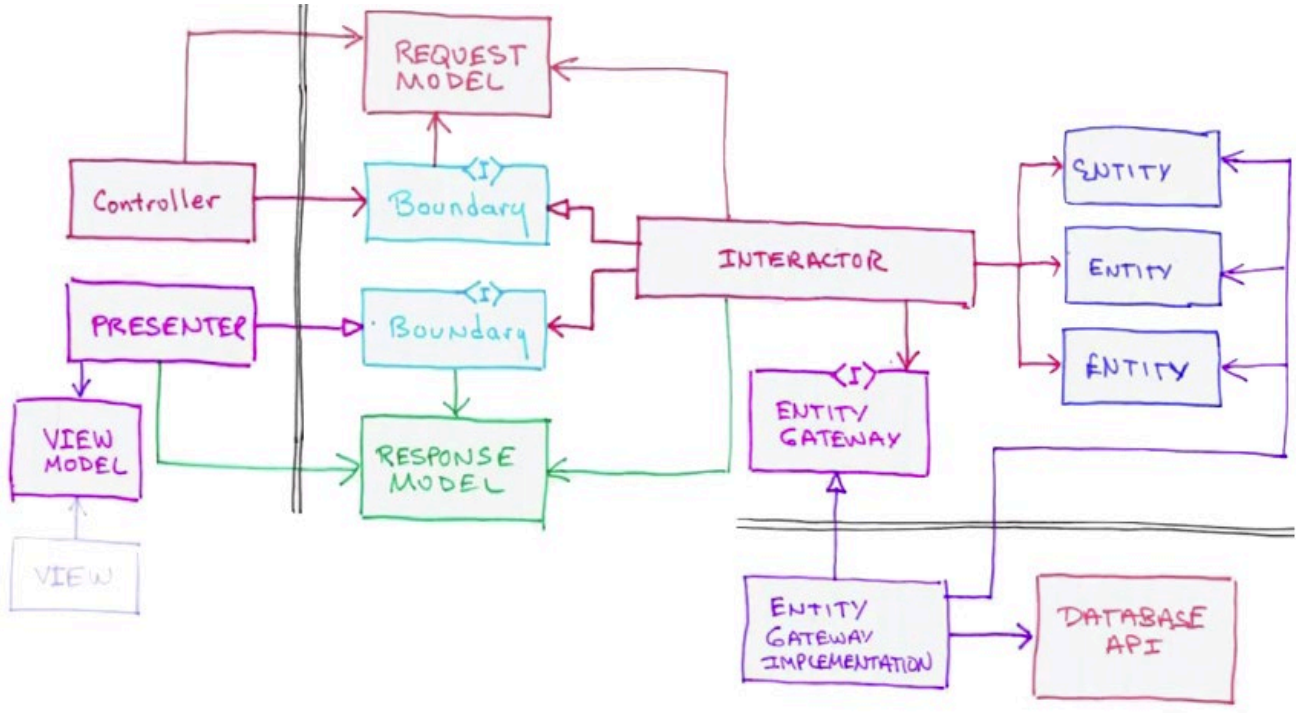


Figure 29 MVP (Robert C. Martin talk)

[16]Model view presenter is a derivation of the previously mentioned model view controller architectural pattern. It is mostly used for building interfaces and was first published and elaborated upon at Taligent, a joint venture of Apple, IBM and HP.

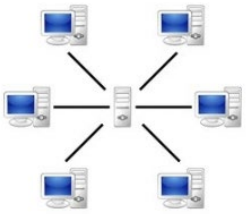
Model
The model defines the data to be displayed or otherwise acted upon in the user interface.

View
Passive interface that displays data and forwards user events to the presenter.

Presenter
Interacts with the model and view. Retrieves data from the model and formats it to be eventually displayed in the view. Furthermore, it is responsible for user events forwarded by the view.

[16] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter 05.03.2019

4.5 Client-Server and Peer to Peer

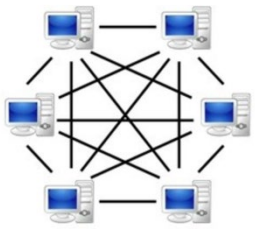


Server-based

Client/Server

[17]In a client/server environment, there are a number of client computers that the users can utilize. These computers make requests to the server. The server merely processes the requests of the clients. Some examples of typical client request to a sever would be: loading files from the server, printing files over a network, and saving files to a network share.

- The advantages of a client/server environment include stronger and less intrusive security, central file storage, and centralized management and speed. With a client/server setup, users can usually specify the file or object level security and a password that is needed to access the server. Having all the files located on one computer makes performing backups and file administration much easier than if they were spread across a number of computers. The increase in speed is garnered from the fact that a server does nothing but process client requests and as such is optimized to perform that particular task.
- The disadvantages of the client/server model are mostly cost related. It is initially expensive to purchase a server capable of supporting a large number of users. Network operating systems also tend to be more expensive than non-network optimized operating systems and servers require at least one administrator.



P2P-network

Peer-to-Peer

In a peer-to-peer networking environment, each computer can act as both a server and a client. Therefore, each computer can process the requests of another computer or send requests to another computer.

- Peer-to-peer gains its largest advantage from cost, as it does not require a centralized server, a network operating system, or an administrator. Peer-to-peer networks are much easier to set-up than a client/server network and they do not rely upon a server for operation.
- There are several disadvantages for peer-to-peer networks. They are slower than client/server networks and they are difficult to manage since each user becomes a network administrator. Peer-to-peer networks are not capable of supporting as many users as a client/server model nor do they have a central location for file storage. As a result, file management and backups can be difficult. Most peer-to-peer networks only allow specification of a separate password for each device, rather than defining access by account and assigning the account a single password. Furthermore, peer-to-peer networks typically only allow security on a specific drive or directory. Peer-to-peer networks are also incapable of supporting as many connections as a client/server configuration allows.

Figure 30 Client-Server / P2P (cssouth.com, DOA 01.01.19)

[17] https://pub.tik.ee.ethz.ch/students/2002-2003-Wi/SA-2003-16.pdf 07.03.2019

4.6 General Responsibility Assignment Software Patterns - GRASP

- Low Coupling
Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Low coupling is an evaluative pattern that dictates how to assign responsibilities to support lower dependency between the classes, change in one class having lower impact on other classes and ultimately higher reuse potential.

- High Cohesion
High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, reuse, maintain and change.

- Information Expert
Information expert is a principle used to determine where to delegate responsibilities such as methods and attributes. Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. This will lead to placing the responsibility on the class with the most information required to fulfill it.

- Indirection
The indirection pattern supports low coupling and reuse potential between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view control pattern. This ensures that coupling between them remains low.

- Controller
The controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event. A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case. For instance, for the use cases Create User and Delete User, one can have a single class called UserController, instead of two separate use case controllers. The controller is defined as the first object beyond the UI layer that receives
and coordinates a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself.

- **Polymorphism**

According to the polymorphism principle, responsibility for defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.

- **Creator**

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.
In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:
Instances of B contain or compositely aggregate instances of A
Instances of B record instances of A
Instances of B closely use instances of A
Instances of B have the initializing information for instances of A and pass it on creation.

- **Pure Fabrication**

A pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived. E.g. Service / Data Access Layers.

- **Protected Variations**

The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

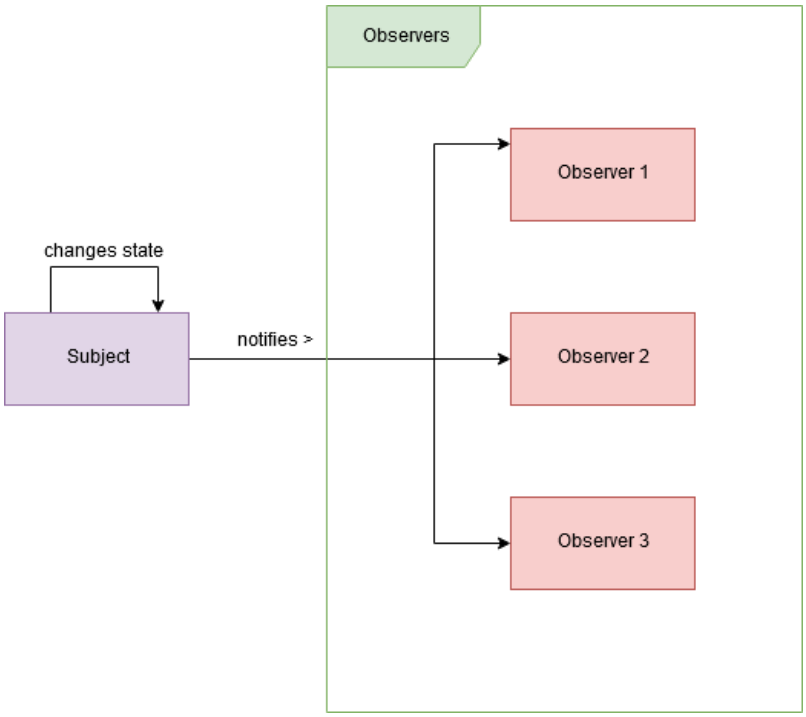
4.7 Gang of Four Patterns – Go4

Singleton

The singleton pattern restricts the numbers of instances of an object to a single object, hence the name. The singleton pattern has a private constructor which is only called once, in the getInstance() method of its class. From then on said method only returns the created singleton object. With this pattern, we can prevent the creation of multiple instances of a single object, despite only needing a single instance. This makes our program faster and lighter. An example of a class which implements the singleton pattern:

```
public class SingleObject {
    private static SingleObject instance;

    private SingleObject() {}

    public static SingleObject getInstance() {
        if (instance == null) {
            instance = new SingleObject();
        }

        return instance;
    }

}
```

Observer - Observable



The observer pattern consists of two components, the observer and the observable. This pattern defines a one-to-many dependency between objects so that when one object (subject) changes state, all of its dependents (observers) are notified and updated automatically. Java 9 has replaced the observer class with listeners, but the pattern is still in use.

4.8 Three-Tier Web Applications

Presentation Tier
The presentation layer is the frontend and consists of the user interface. Most often the UI is a graphical one and accessible through a web browser or web-based application.
It is built on web technologies such as HTML 5, JS, CSS or through other popular web development frameworks. The presentation layer communicates with the application layer through API calls.

Application Tier
The application layer provides the business logic which drives an application's core functionalities and capabilities.

Data Tier
The data layer comprises of the data storage. Examples of such systems are MySQL, PostgreSQL.

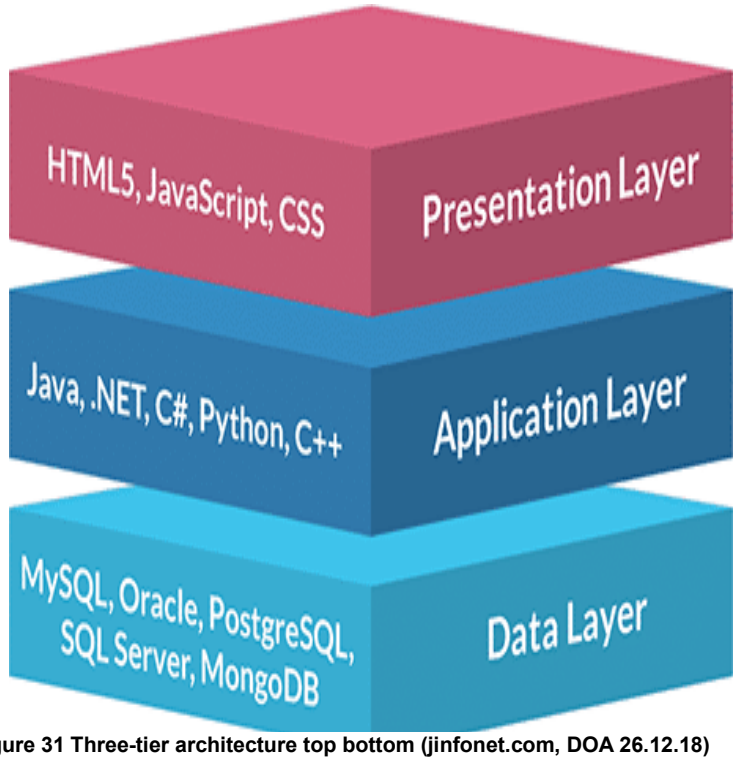


Figure 31 Three-tier architecture top bottom (jinfonet.com, DOA 26.12.18)

With the usage of three layers, each part can be developed by different teams using different languages and frameworks. In fact, each layer can easily be changed or relocated without affecting others. This enables companies to continually evolve software as new needs and opportunities arise. A typical structure of a three-layer architecture would have the presentation logic deployed to an end-user gadget such as a desktop, tablet or mobile phone. The underlying application layer is mainly hosted on a tier with a single or with multiple application servers, however, can also be hosted in the cloud or on a workstation depending on the scale and complexity. Last but not least, the data layer usually consists of one or more relational databases, big data sources or other types of database systems hosted either on premises on in the cloud.

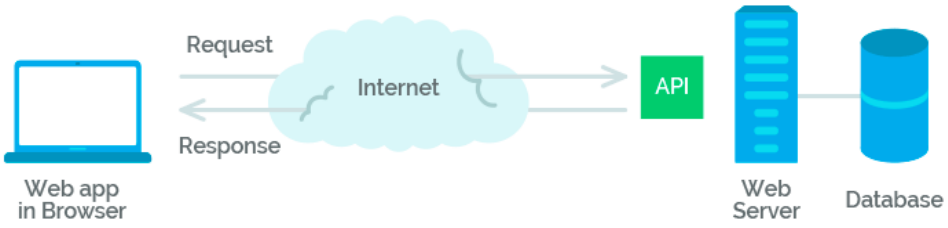


Figure 32 Client-server (mlsdev.com, DOA 31.12.18)

4.8.1 Application Tier Architecture – Three Layered Application

Now that we know the fundamental principles of each tier, let us have a closer look at how the logic within the application tier is split up. It should be able to take care of the following concerns:

1. It needs to process and handle the clients input and return an adequate response.
2. It needs to keep track of the transactions.
3. It needs an exception handling strategy that informs the client about occurring errors in a reasonable fashion.
4. It needs to take care of both authentication as well as authorization.
5. It needs to implement the core business logic of the application.
6. It needs to be able to access the data tier or other external resources.

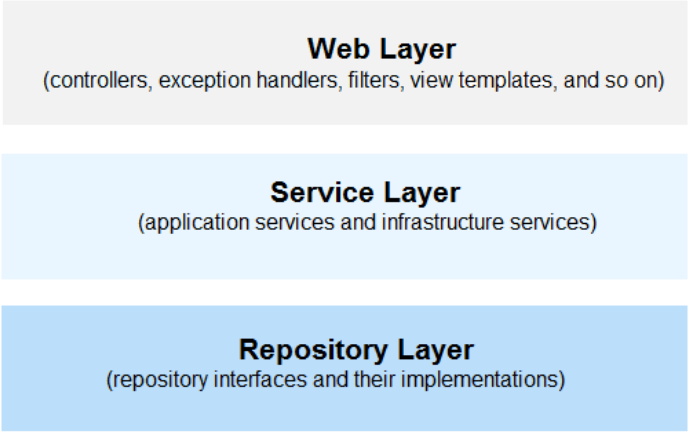


Figure 33 Web, service and repository layer (petrikainulainen.net, DOA 26.12.18)

We can fulfil those concerns by splitting up the logic in 3 independent layers:

- The web layer functions as the top layer of a web application. It is responsible for processing and handling the clients input and returning correct responses. The web layer also needs some sort of an exception handling strategy to be able to deal with errors thrown by other layers. This layer also functions as an entry point and therefore needs to take care of authentication and authorization and act as a first line of defence.
- The service layer is located below the web layer and acts as an instance in between. It provides a public service API to the web layer. Those services collect corresponding data from the data layer and add business logic on top if needed.
- The repository layer is the lowest layer of a web application. It holds the data access layer which communicates with the used data storage.

Now that we have all three layers defined, let us look at the interface of each layer. That's where we run into terms like data transfer object (DTO) and data access object (DAO).

A data transfer object (DTO) is an object that carries data between processes. The motivation for its use is that communication between processes is usually done resorting to remote interfaces (e.g. web services), where each call is an expensive operation. Because the majority of the cost of each call is related to the round-trip time between the client and the server, one way of reducing the number of calls is to use an object (the DTO) that aggregates the data that would have been transferred by the several calls, but that is served by one call only.

The difference between data transfer objects and business objects or data access objects is that a DTO does not have any behaviour except for storage, retrieval, serialization and deserialization of its own data. In other words, DTOs are simple objects that should not contain any business logic but may contain serialization and deserialization mechanisms for transferring data over the wire.

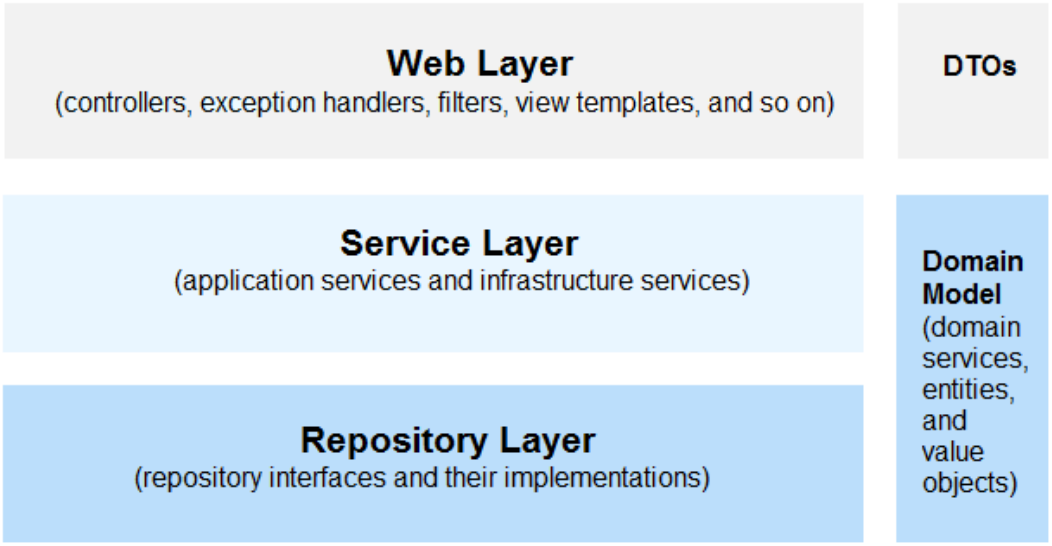


Figure 34 Data transfer object and data access object (petrikainulainen.net, DOA 26.12.18)

4.8.2 Application Tier Architecture – Onion Architecture

Another way to structure the application tier is the onion architecture first formalized in Jeffrey Palermo's blog (refer to this page for a more detailed explanation). In terms of application components, it is similar to the three tier application architecture but it aims to restructure the general flow of dependencies. In traditional three-layered applications all dependencies essentially point towards the lowest layer, usually the data layer. This means that the whole application is strongly coupled to the underlying infrastructure, which, especially for long lived applications, makes it very hard to deal with scaling and other non-functional requirements. With the onion architecture dependencies are only allowed to point towards the centre. The core being the domain model and the correlating business logic, which should be completely agnostic to the application's infrastructure. This enables to take the "clean" application core (onion) out of its soil and plant it somewhere else (employ a different infrastructure), relatively hassle-free.

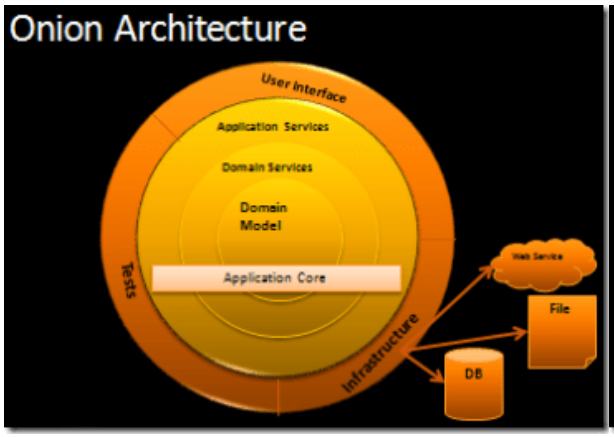


Figure 35 onion arch. (jeffreypalermo.com, DOA 20.04.19)

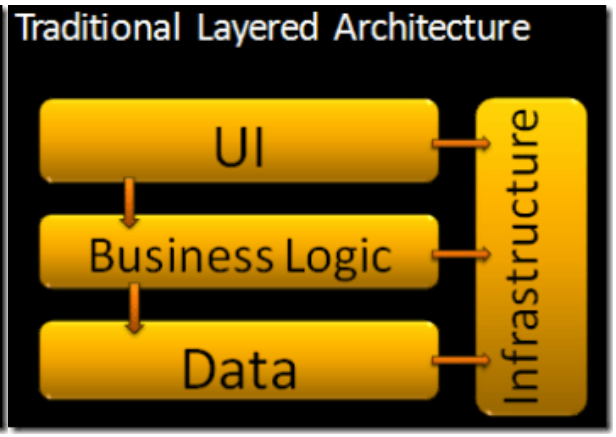


Figure 36 traditional (jeffreypalermo.com, DOA 20.04.19)

4.9 Single Page Application (SPA) vs. Multi Page Application (MPA)

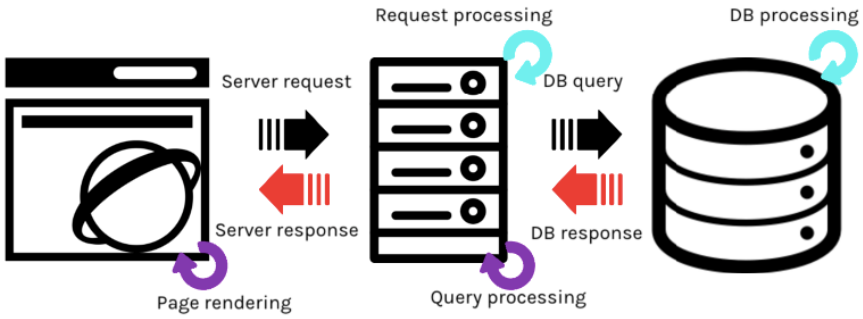


Figure 37 MPA (codingninjas.co, DOA 2.1.18)

MPA's are more of a "traditional" way of delivering a dynamic user experience. Every change to the page that has been initiated by some action sends a request back to the server to require the rendering of a new page in order to update the appearance and content from the user's perspective.

Single page applications on the other hand, avoid reloading the entire page, but instead rewrite a specific part of the current page using AJAX requests.

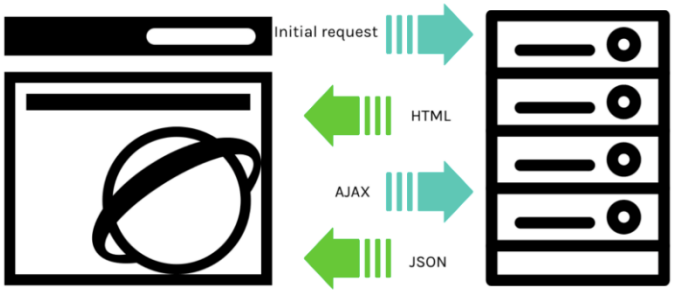


Figure 38 SPA (codingninjas.co, DOA 2.1.18)

4.10 Representational State Transfer (REST)

Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style provide interoperability between computer systems on the Internet. RESTful web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

In a RESTful web service, requests made to a resource's URI will elicit a response with a payload formatted in HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When HTTP is used, as is most common, the operations available are GET, POST, PUT, DELETE, PATCH and other predefined CRUD HTTP methods.

Six guiding constraints define a RESTful system. These constraints restrict the ways that the server can process and respond to client requests so that, by operating within these constraints, the system gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability. If a system violates any of the required constraints, it cannot be considered RESTful.

The formal REST constraints are as follows:

1. Client–server – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

2. Stateless – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

3. Cacheable – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

4. Uniform interface – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behaviour of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

5. Layered system – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behaviour such that each component cannot “see” beyond the immediate layer with which they are interacting.

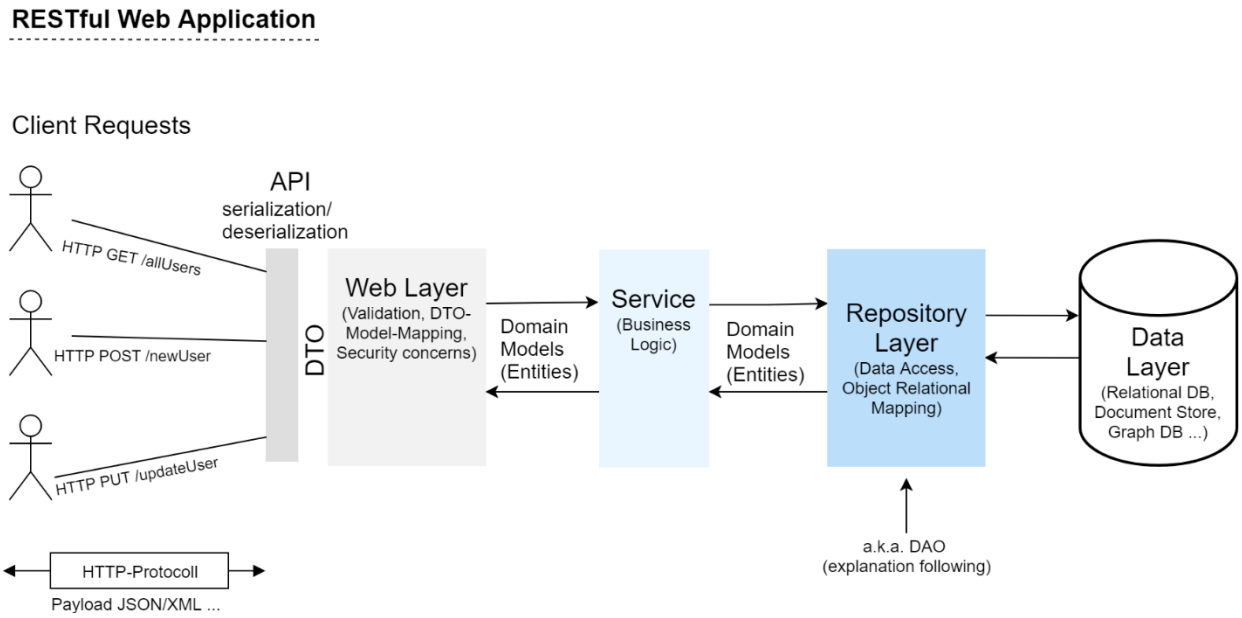


Figure 39 RESTful API

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

GET:
The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

HEAD:
The HEAD method asks for a response identical to that of a GET request, but without the response body.

POST:
The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

PUT:
The PUT method replaces all current representations of the target resource with the request payload.

DELETE:
The DELETE method deletes the specified resource.

CONNECT:
The CONNECT method establishes a tunnel to the server identified by the target resource.

OPTIONS:
The OPTIONS method is used to describe the communication options for the target resource.

TRACE:
The TRACE method performs a message loop-back test along the path to the target resource.

PATCH:
The PATCH method is used to apply partial modifications to a resource.

4.10.1 REST Resource Naming TODO attr

In REST, primary data representation is called **Resource**. Having a strong and consistent REST resource naming strategy is a good decision for the long term.

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.
>
> – Roy Fielding's dissertation

A **resource can be a singleton or a collection**. For example, `customers` is a collection resource and `customer` is a singleton resource (in a banking domain). We can identify the `customers` collection resource using the URI `/customers`. We can identify a single `customer` resource using the URI `/customers/{customerId}`.

A **resource may contain sub-collection resources** also. For example, sub-collection resource `accounts` of a particular `customer` can be identified using the URN `/customers/{customerId}/accounts` (in a banking domain). Similarly, a singleton resource `account` inside the sub-collection resource `accounts` can be identified as follows: `/customers/{customerId}/accounts/{accountId}`.

Note: The two guidelines presented above are not strict requirements but rather best-practices. Under certain circumstances

REST APIs use Uniform Resource Identifiers (URIs) to address resources. REST API designers should create URIs that convey a REST API's resource model to its potential client developers. When resources are named well, an API is intuitive and easy to use. If done poorly, that same API can feel difficult to use and understand.

CRUD operations are usually mapped onto resources as follows:

Operation	Mapping	Description
Create	`POST /collection`	Adds the element in the request body to the collection => a new element is stored in the DB
Retrieve	`GET /collection/{id}`	Reads the element with the specified id
Update	`PUT /collection/{id}`	Updates the element in the collection with the specified id to match the element supplied in the request body
Delete	`DELETE /collection/{id}`	Removes the element with the specified id from the collection

5 Development Methodologies

5.1 Chapter Objectives

The objectives for this chapter are as follows:

- You are familiar with the iterative and incremental approach
- You understand the key principles and manifesto of agile software development
- You know how scrum is applied in a real-life scenario
- You know which elements a story map backlog is built upon

5.2 Iterative and Incremental

One of the first decisions to be made is the choice of an adequate development methodology.

An iterative approach makes progress through successive refinement. Software is deployed and distributed even though it might be partially incomplete. Those parts then get iteratively refined until the product reaches a satisfactory state. With each iteration the quality of the product improves through the addition of greater detail.

Figure 40 Iterative approach (adambikrn.com, DOA 01.01.19)

An incremental approach is one in which software is built and delivered in individual pieces. Each increment represents a complete subset of functionality and is fully tested. It is a common expectation that the work of such an increment will not to revisited or altered with again.

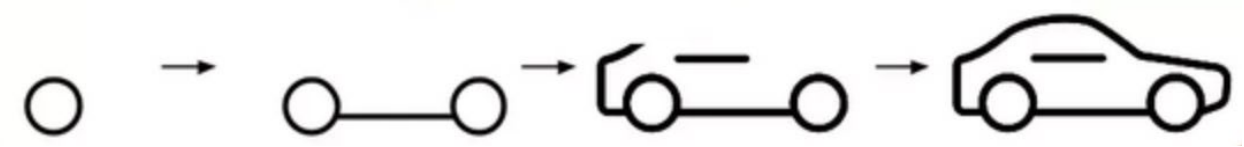


Figure 41 Incremental approach (adambikrn.com, DOA 01.01.19)

5.3 Agile Software Development and Scrum

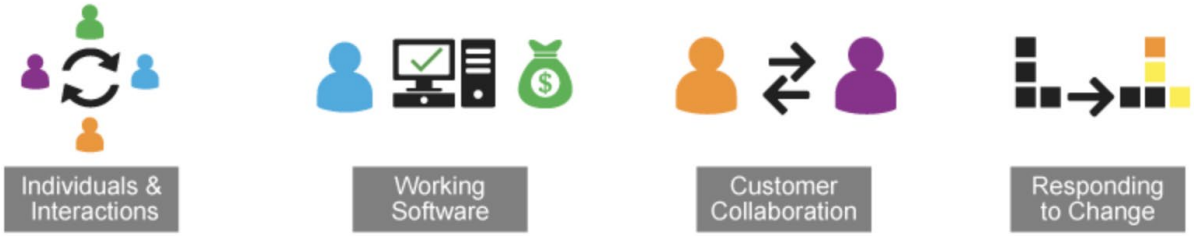


Figure 42 Agile software development principles (visual-paradigm.com, DOA 01.01.19)

Agile software development emphasizes the ability to create and respond to alternations in order to succeed in an uncertain and often unforeseen environment. The methodology focuses on rapid and frequent successive solutions that can be evaluated and used for further steps. An agile strategy generally promotes a disciplined project management process that encourages frequent adaptation and a business approach that aligns development with customer needs and company goals.

The agile manifesto values are captured in twelve principles:

- **Customer satisfaction** through early and continuous software delivery. Customers are happier when they receive working software at regular intervals, rather than waiting extended periods of time between releases.
- **Accommodate changing requirements** throughout the development process. The ability to avoid delays when a requirement or feature request changes.
- **Frequent delivery of working software**. Scrum accommodates this principle since the team operates in software sprints or iterations that ensure regular delivery of working software.
- **Collaboration between the business stakeholders and developers** throughout the project. Better decisions are made when the business and technical team are aligned.
- **Support, trust, and motivate the people involved.** Motivated teams are more likely to deliver their best work than unhappy teams.
- **Enable face-to-face interactions**. Communication is more successful when development teams are co-located.
- **Working software is the primary measure of progress**. Delivering functional software to the customer is the ultimate factor that measures progress.
- **Agile processes to support a consistent development pace**. Teams establish a repeatable and maintainable speed at which they can deliver working software, and they repeat it with each release.
- **Attention to technical detail and design enhances agility**. The right skills and good design ensures the team can maintain the pace, constantly improve the product, and sustain change.
- **Simplicity**. Develop just enough to get the job done for right now.
- **Self-organizing teams encourage great architectures, requirements, and designs**. Skilled and motivated team members who have decision-making power, take ownership, communicate regularly with other team members, and share ideas that deliver quality products.
- **Regular reflections on how to become more effective**. Self-improvement, process improvement, advancing skills, and techniques help team members work more efficiently.

Scrum is an agile way to manage a project, usually software development. It is often perceived as a methodology but in reality, rather functions as a framework. Scrum follows both the incremental as well as iterative principles if conducted successfully.

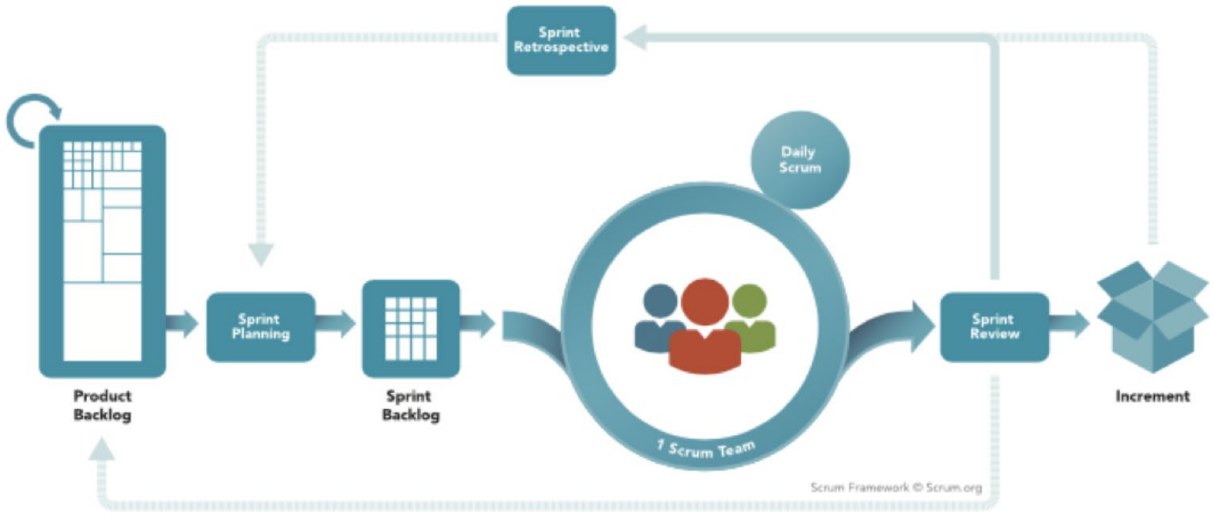


Figure 43 Scrum (scrum.org, DOA 01.01.19)

- The **product backlog** functions as a collection of issues that are to be built into the product being created. Creating a backlog often starts with various discovery sessions and researches to develop a fundamental understanding of all given problem domains. Those meetings usually include parties such as project owner, project manager, developers including scrum master and other involving groups.

- The elaborated features will now be packaged and delivered through a series of timeboxed iterations called sprints. The priorization and allocation of features is usually done by the product owner in a **sprint meeting**. Sprints have a fixed duration of 1-4 weeks. Each delivering a subset of the actual product backlog. To provide additional features, sprints are dynamically planned and conducted. Previous feedback is being incorporated. An individual sprint is both iterative and incremental. He provides improvements to work completed in past sprints and adds new features to the product.

- **A daily scrum** takes place every 24 hours. All team members report to the scrum master. What has been done the previous day? What will assumingly be done today? Any issues or challenges that might occur?

- **The sprint review** meeting is held at the end of each sprint. It is usually kept very informal and typically does not take longer than 1 hour / week of sprint. The delivered subset is assessed against the sprint goals determined earlier.

- Next to the sprint meeting, a **retrospective meeting** is held. The sprint is reflected upon. What has gone wrong? Which topics could be improved in the next sprint?

A scrum team usually consists of the following roles:

Scrum Development Team

Development teams are cross-functional. Meaning they hold all of the skills necessary to create a product increment. Scrum recognizes no titles for individual members. Nor does it support sub-teams within the team. Individual development team members may have specialized skills and areas of focus, but accountability belongs to the development team as a whole.

Product Owner

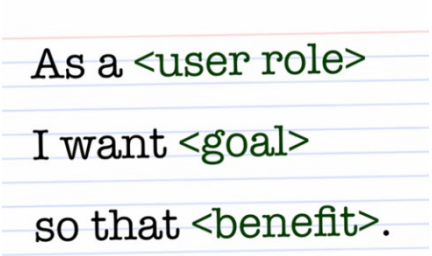
Clearly identifies and describes the product backlog items in order to build a shared understanding of all given problem domains. He is responsible for the priorization and allocation of the given features. He also determines if an increment was satisfactorily delivered after a sprint.

Scrum Master

The scrum master is a member of a software development team. He ensures that Scrum is applied acted upon in the right manner. He shields the team from external interference and distractions. He does per se not have any management authority over the team however.

5.4 User Stories

The initial step of a project is not actual programming, but rather a thorough analysis of the client's requirements. A well-founded analysis and design of an application is a crucial step towards an implementation that satisfies both the developers as well as the clients.



A user story quickly captures the "who", "what" and "why" of a specific product requirement.

Figure 44 User story (scrumwithstyle.com, DOA 01.01.19)

Requirements always change as teams and customers learn more about the system as the project progresses. It's not exactly realistic to expect project teams to work off a static requirements list and then deliver functional software months later. User stories reduce the time spent on writing exhaustive documentation by emphasizing customer-centric conversations. The simple and consistent format saves time when capturing and prioritizing requirements while remaining versatile enough to be used on large and small features alike.

A good user story should be - **INVEST**:

- **Independent**: Should be self-contained in a way that allows to be released without depending on one another.
- **Negotiable**: Only capture the essence of user's need, leaving room for conversation. User story should not be written like contract.
- **Valuable**: Delivers value to end user.
- **Estimable**: User stories have to able to be estimated so it can be properly prioritized and fit into sprints.
- **Small**: A user story is a small chunk of work that allows it to be completed in about 3 to 4 days.
- **Testable**: A user story has to be confirmed via pre-written acceptance criteria.

5.5 Flat Product Backlog and User Story Map

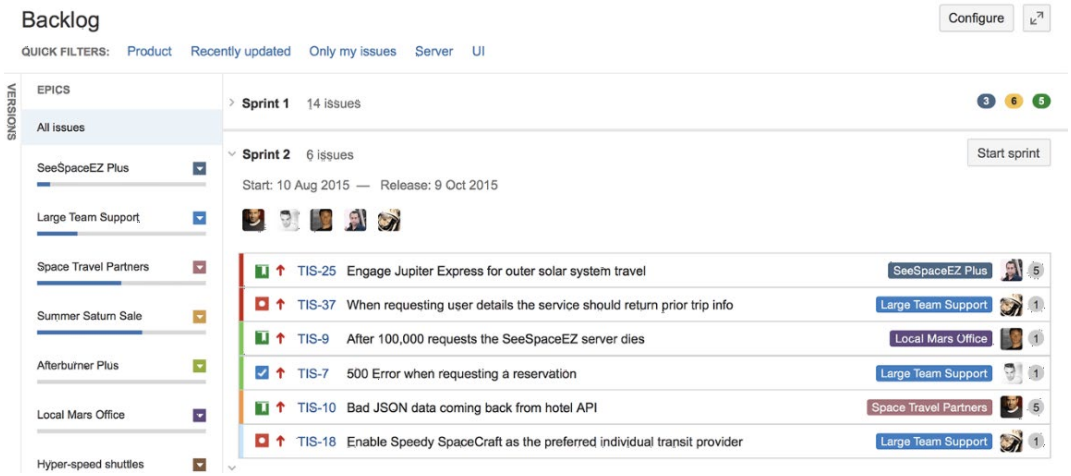


Figure 45 Flat product backlog (blog.easyagile.com, DOA 01.01.19)

A flat product backlog is one of the most common practices in agile software development. We have all been exposed and contributed to them. It is rather difficult to prioritize individual tasks at a sprint meeting and determine if the relevant user stories have been identified. It provides no big picture and it is merely impossible to discover the backbone of your product.

"We spend lots of time working with our customers. We work hard to understand their goals, their users, and the major parts of the system we could build. Then we finally get down to the details—the pieces of functionality we'd like to build. In my head I see a tree where the trunk is built from the goals or desired benefits that drive the system; big branches are users; the small branches and twigs are the capabilities they need; then finally the leaves are the user stories small enough to place into development iterations.
After all that work, after establishing all that shared understanding, I feel like we pull all the leaves off the tree and load them into a leaf bag—then cut down the tree.
That's what a flat backlog is to me. A bag of context-free mulch" - **Jeff Patton**

Jeff Patton wasn't quite a big fan of flat product backlogs, as you might have seen above.
In his mind, story maps were a more suitable approach. Not only do they facilitate the process of shaping epics and creating user journeys but also help discovering requirements, which might not have been apparent at first sight. The client walks through the problem by telling a story of activities and tasks that are performed. Initially, his perception of things might not be too apparent and linear. The journey becomes a lot clearer as the analysis advances.

Figure 46 Story map initial phase (agilevelocity.com, DOA 01.01.19)

Once we have the majority of activities and tasks involved, it is time to order. Let us Identify groupings and bundle them into epics. Any activities that are strongly related? Any tasks that might be redundant?

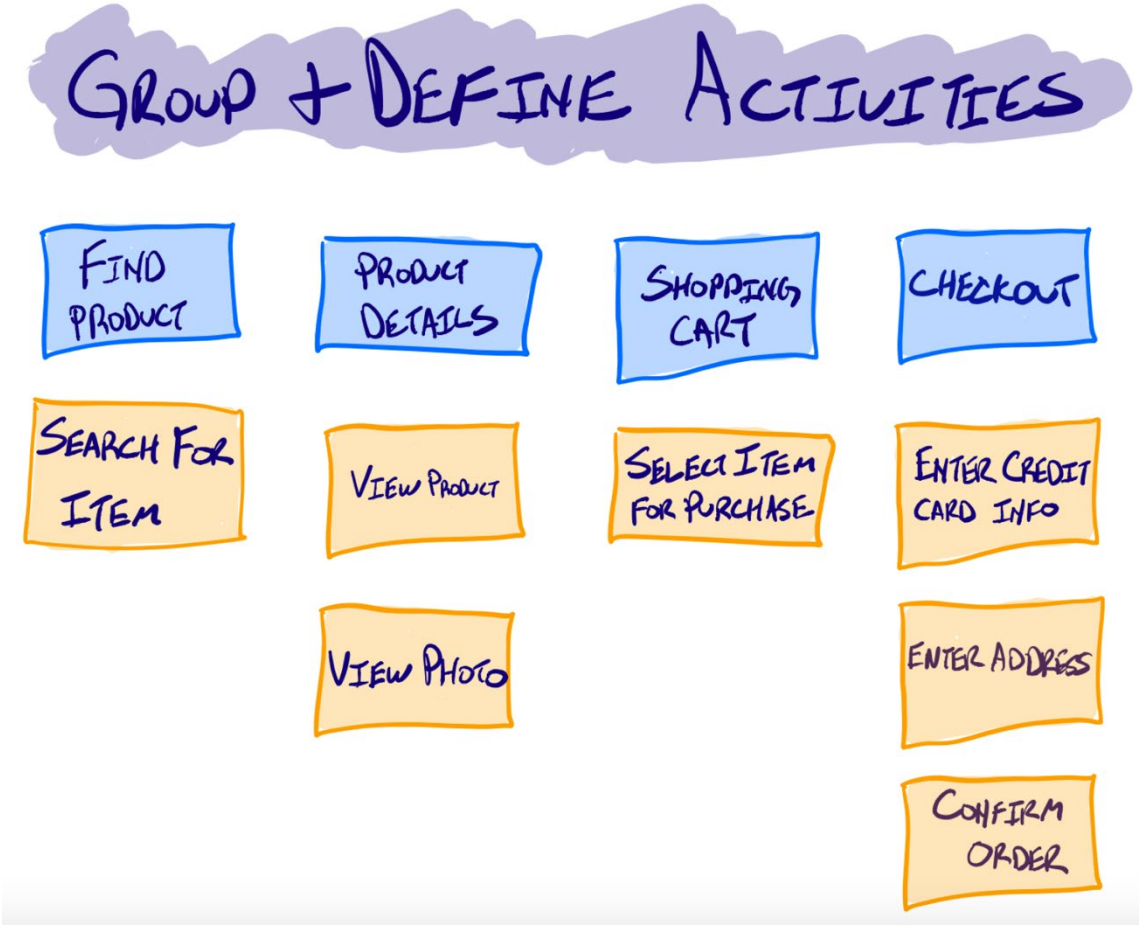


Figure 47 Story map grouping (agilevelocity.com, DOA 01.01.19)

Last but not least we want to ensure we have no major gaps or missing activities within our map. Have someone else walk through the scenario form a different point of view. Only then we can review and prioritize individual tasks. Within columns, items that are least important are being moved down. Others are moved up. Iterations or Sprints can be set by drawing a line between activities.

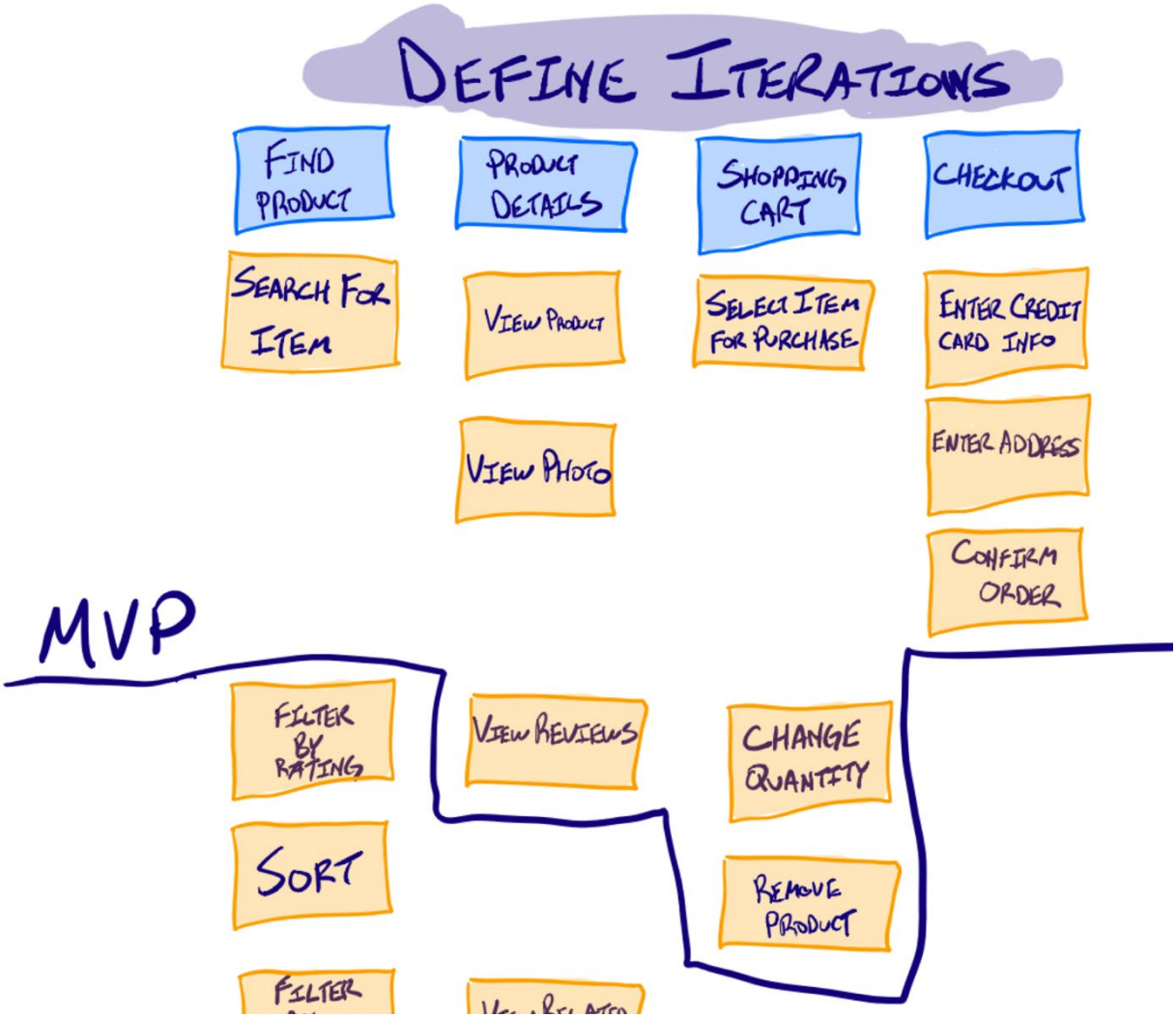


Figure 48 Story map (agilevelocity.com, DAO 01.01.19)

6 Tooling – Version Control, Git and GitHub

6.1 Chapter Objectives

You are familiar with …

- The benefits of using a version control system
- The differences between a centralized and a distributed version control system
- The basic principles of Git and its distinctions to other existing version control systems
- The individual file status lifecycles
- The different Git buckets
- The basic local workflows of git (init, status, add, commit, log, diff etc.)
- Creating, switching and merging branches
- Collaborating with co-workers using remote repositories (push, pull, fetch, clone, fork)
- Working with remote repositories in Git

6.2 Purpose of Version Control

Figure 49 "Final".doc (PHDCOMICS 1531, DOA 25.12.18)

Edit multiple files
What has changed?
How can I assess possible differences between files?

Trying to access an older version of a file
Restore an existing backup?

Working with multiple colleagues
Whose change is it?
How can I correctly integrate different changes?
Do I have the latest version?
Can I interchange files with a team consistently?
What about an agile development process?

Working on different versions
How can I support older versions while working on a newer version?
Can I maintain software on different systems?
Is it possible to support customer specific extensions?

6.3 Centralized vs Distributed Version Control Systems

A centralized version control system (CVCS) works on a client-server model. There is a single master copy of the code base. Pieces that are being worked on are typically locked or formerly “checked out” so that only one individual developer is allowed to work on that specific part of the code at one time. Once the work is checked back in, the lock is released and the fragment is available for others to check out again. As soon as each member of the team has finished working on their different pieces and it is time to make a new release, modules are gathered and combined together and then rolled out in a new version. Subversion or SVN is a good example of a CVCS.

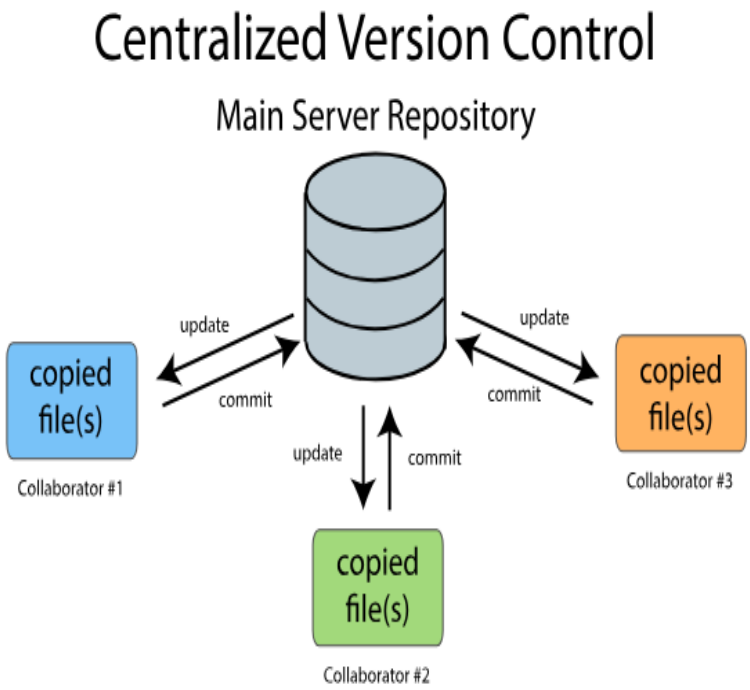
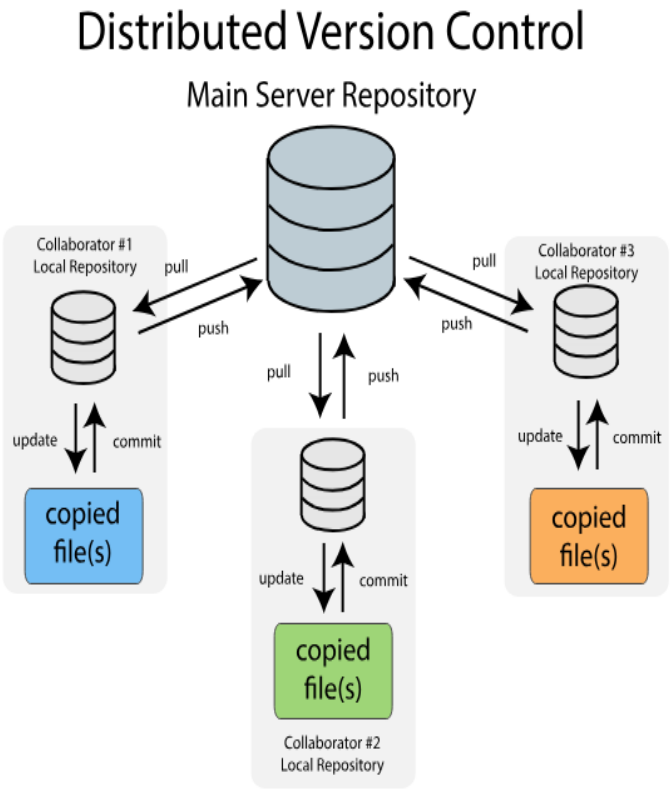


Figure 50 Centralized version control (code.snipcademy.com, DAO 25.12.18)



A distributed version control (DVCS) works on a peer-to-peer model. The code base is distributed amongst the team members’ computers. In fact, the complete history of data is (i.e. can be) mirrored on each system. Technically there is still a master copy of the code base, but it is kept on a client machine rather than on a server. Locking individual parts of files is not possible. Thus, developers make changes in their local copy and then, once they are ready to integrate their changes into the master copy, issue a pull request to the owner to merge their changes. Ultimately, the emphasis switches from versions to changes. A new version is simply a combination of a number of different sets of changes. Git is a good example of a DVCS.

Figure 51 Distributed Version Control (code.snipcademy.com, DAO 25.12.18)

6.4 Git

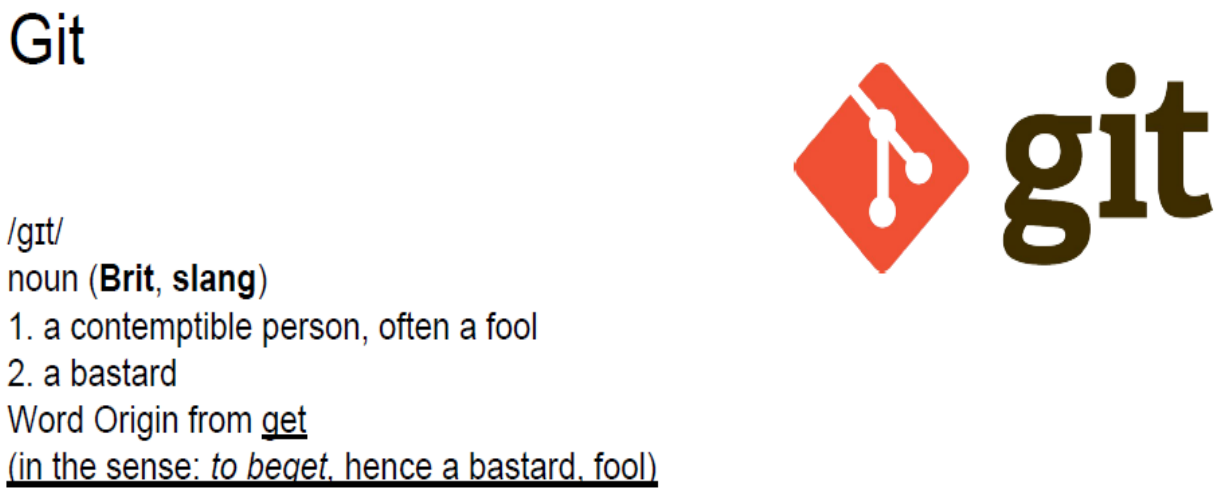


Figure 52 Tool git (Urban dictionary, DAO 25.12.18)

6.4.1 Concepts and basics

So, what is Git in a nutshell? This is a crucial section to absorb, because if you grasp the fundamentals of Git, using it effectively will probably be much easier for you. Before we continue, attempt to free your mind of the things you may or may not know about other VCS. Git stores and deals with information in a very distinctive way. Understanding these differences will eventually help you to avoid subtle confusion when using the tool.

Commits
In Git, commits are the basic building blocks of a repository [the thing that contains the source code and its history]. Commits represent a set of changes, namely insertions and deletions, made to the source code at a specific point in time. Optionally each commit has a single focus on a change / feature implemented. In Git, commits are identified by the SHA-1 hash of its content and metadata.

Snapshots, not Differences
One of the most fundamental differences between Git and any other VCS is the way Git keeps track of different versions. Most conventional systems store data as a collection of file-based changes. They think of information they store as a set of files and track the changes made over time. When a file is looked at, its contents are reconstructed by going through all changes made to it and applying them to the “origin-version” of the file. This is commonly known as delta-based version control.

Figure 53 Storing data as changes to base version of each file (git.scm.com, DAO 25.12.18)

Git however, stores its data more like a continuous series of file snapshots [copy of a file at a certain point in time] and stores them in its own miniature file system. With each commit, a new snapshot for each changed file is stored. If a file has remained unchanged, a link to a previously taken snapshot of it is stored, since it is still identical to the current version of the file.

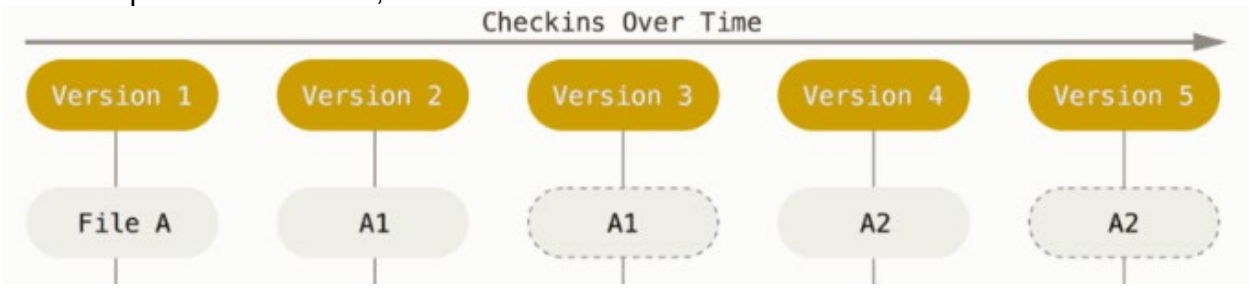


Figure 54 Storing data as snapshots of the project over time (git.scm.com, DAO 25.12.18)

Operations are mostly conducted locally
Most commands do not rely on remote files or resources to operate. If you are used to a CVCS where most operations come with a network latency overhead, this certain aspect of Git will make you think that the almighty gods of speed have descended from heaven and blessed Git with unworldly powers. This is because the entire history of the project is located on your local disk, which makes most operations seem almost instantaneous.
For instance, if you desire to browse the history of your project, Git does not need to go out to the server in order to get the latest history, but rather reads it directly from your local image. This comes in handy when you do not have access to an active network connection at a given time.

Git has Integrity
Git stores its data in a content-addressable file system. Great. What does that mean? It basically means that the core of Git is a simple key-value data store. Meaning you can insert any given content into a Git repository, for which Git will then hand you back a unique identifier, which can later be used to retrieve the saved data.
The mechanism that Git uses for check summing is called a SHA-1 hash (check appendix for further information).

Git generally only adds data
Nearly all commands add rather than remove data from the database [the key-value store]. It is complicated to get the system to do anything that is not reversible. You can lose or mess up changes you have not committed yet, but after you committed a snapshot into Git, it is merely impossible to lose, especially if you regularly push your local image to a remote repository (which is strongly recommended)

Branches

When creating commits, Git stores a commit object that contains a pointer to the snapshot of its content, some other information and a pointer to the commit that was created immediately before it, also called its parent commit. The pointer to a commit's parent is also hashed to create a commit's identifying hash [as described in 'commits'], meaning that multiple commits (in the same commit history) essentially form an immutable chain of commit's, where changing the parent of any commit in the chain would mess up all hashes in that chain.

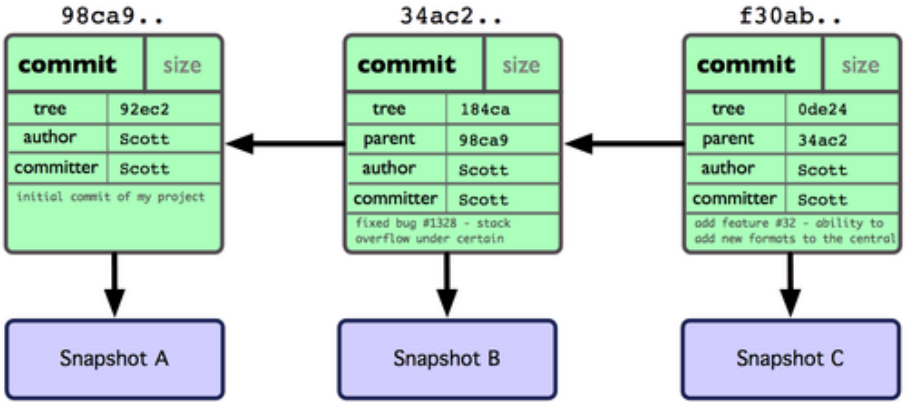


Figure 55 Chain of commits (git-scm.com, DOA 08.03.2019)

With this system in place, it is also possible for a commit to have two or more children in which case, a chain of commits would diverge into two or more separate histories / chains (/branches but not really). Developers could now work separately on these two distinct histories without interfering with each other. They could later merge both histories together by creating a merge commit which has the two latest commits of the divergent histories as its parents (yes they can also have two parents…).

Now that we've been talking about histories and chains, you might be asking yourself: "What the heck are branches then?" Well, branches are nothing more than pointers used to identify different histories more easily. They point to some commit in the history but they don't really 'contain' commits (which is a common misconception). The `HEAD` is a special 'branch', which always points to the currently checked out branch or commit (in which case it's 'detached').

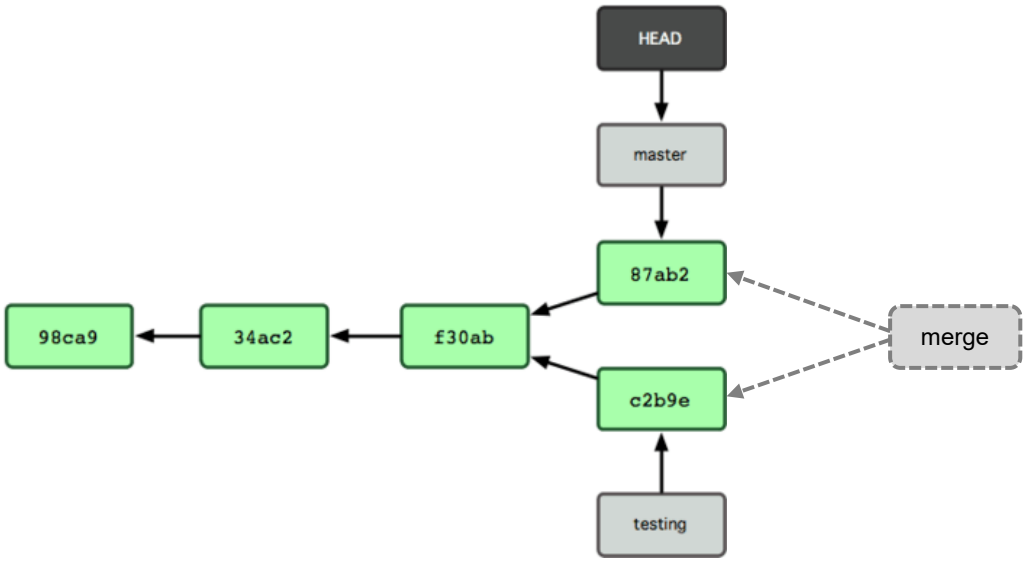


Figure 56 Divergent commit histories (git-scm.com, DOA 08.03.2019)

6.4.2 File status lifecycle

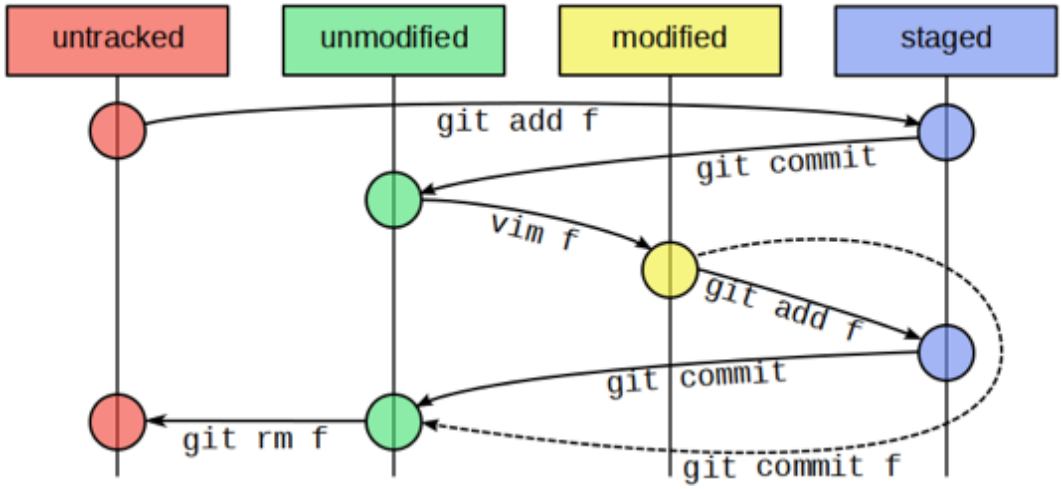


Figure 57 Git file status lifecycles (git.scm.com, DAO 25.12.18)

Each file in your working directory can be in one of two states: tracked or untracked. Tracked files are the files that were part of the most recent snapshot. These files can be unmodified, modified or staged. In short, tracked files are files Git knows / cares about.
The remaining files are labelled as untracked, which means that Git doesn't check them for changes and doesn't put then into commits. All the files that are not in your latest commit nor in the staging area are essentially untracked. Once you clone a repository [copying it from a remote computer to yours], all your files will be labelled tracked and unmodified.
As you edit files, Git marks them as modified, because they have gone through a change since the last commit. As your work evolves, you selectively stage these modified files and then commit those staged changes.

Untracked files: Files that are not under version control by git

- New files
- Files that are currently marked to be ignored by git (.gitignore)
- "Files that isn't exists"

Tracked files: Files that are under version control by git

- **Unmodified:**
  Content or metadata (e.g access rights) of the file are identical to the current version in the local repository i.e. the latest commit
- **Modified:**
  The file content or metadata has been altered since the last commit
- **Staged:**
  The modified file has been added to the staging area to be included in the next commit

6.4.3 Buckets

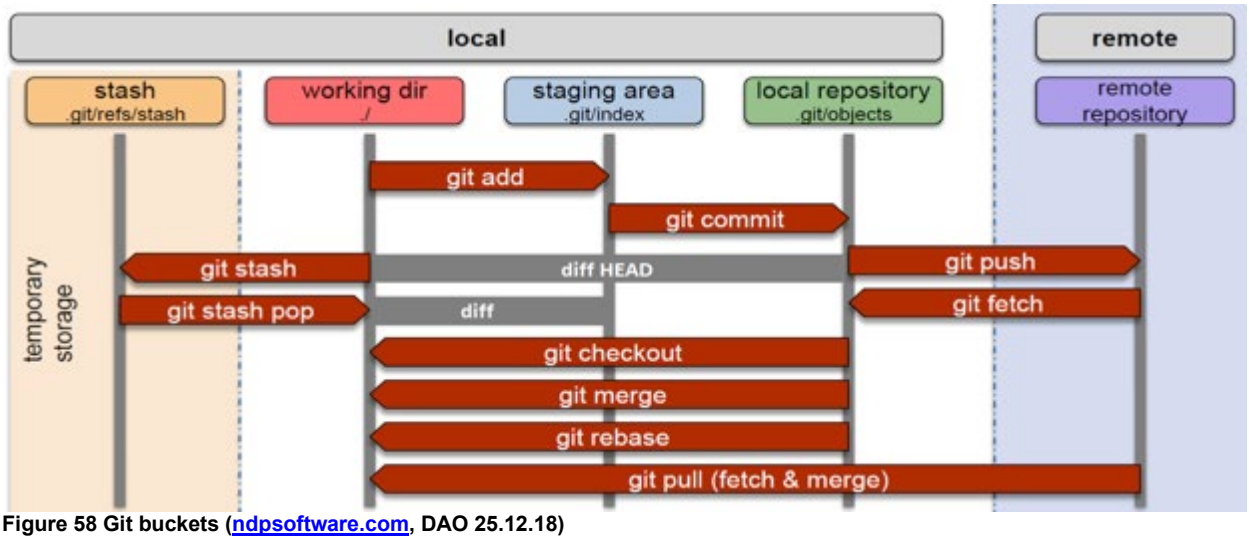


Figure 58 Git buckets (ndpsoftware.com, DAO 25.12.18)

To get a better understanding of how to work with Git on a daily basis, it is wise to familiarize oneself with its different layers. They are as follows:

Stash
Storage to temporary keep modified tracked files of a certain branch while working on a different branch. Imagine you have been working on a part of your project, things are in a messy state and you would like to switch to another branch to work on another topic for a bit. Problem being is that you do not want to commit unfinalized work just so you can return to this point at a later stage. The solution to this issue is git stash. Stashing is nothing else than taking a snapshot of your current working directory and storing it for later use. Git saved it on a stack of unfinished changed that you can reapply at any given time.

Working directory
Holds the checked out version of files you are currently working on (the stuff that is on your actual file system). Think of the working directory as a sandbox. An environment where you can try out changes before actually putting them into your staging area (index). The index as well as the local repository store their content in a rather inconvenient manner inside the .git folder. The working directory unpacks them in actual files, which makes it significantly easier for you to edit them.

Staging area (Index)
Location to accumulate changes for the next proposed commit. Git populates a list of all file contents that were last checked out into your working directory and what they looked like when they were originally checked out. During Development you replace some of those files with new versions.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0       README
100644 8f94139338f9404f26296befa88755fc2598c289 0       Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0       lib/simplegit.rb
```

Figure 59 Possible content of a staging area (git.scm.com, DAO 25.12.18)

Local repository / HEAD

The local repository holds a compressed image of all the files and their commit history. Full diffs, cherry picks and offline commits are only a few possible commands to be used.
Head is a pointer to the currently checked out branch, which in turn is a pointer to a commit. It can also point directly to a commit, in which case it is in the 'detached' state. Generally said, the HEAD embodies a snapshot of the files that are currently checked out. The following example illustrates a possible directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

Figure 60 Possible content of a local repository / HEAD (git.scm.com, DAO 25.12.18)

6.4.5 Commands

Git offers approximately 150 commands, of which only a subset is used on a daily basis. It also offers a very extensive help system:

- Overview of main commands: git help
- List of all commands: git help -a
- Details of a specific command: git help <command>
- Git a list of basic guides: git help -g

The most frequently used commands are as follows:

Local commands	Branch / merge	Remote commands
git init		git branch	git remote
git add		git checkout	git clone
git commit	git merge	git fork
git status	git rebase	git fetch
git log		git stash	git pull
git diff		
git push		
git rm		
git config		
git blame		

PRO TIPP: If you aren't familiar with the Git commands listed below you can use https://learngitbranching.js.org/ to level up your Git knowledge (If unfamiliar → better go through all chapters)

For this course, you should know all of the following commands and should be able to apply them in real-life scenarios (consider using the 'Pro tip' resource if you are insecure).

git init – Create a repository

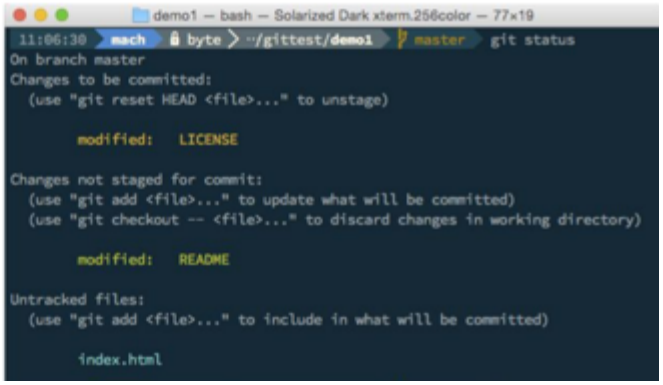
- `git init` creates a new repository for an existing directory
  - Creates a .git directory in the root of the project
  - This directory contains ALL information for the Repository (no pollution by scattered subdirectories like with CVS or SVN)
  - Git assumes that the whole subtree belongs to the project
- `git init [path/to/project/root]`
  - default for project root path is the current directory (`./`)

git add and commit – Create a commit

- Creating a commit consists of two steps
  1. Add changes (files, „hunks") or untracked files to staging area
     `git add LICENSE` adds all changes in the LICENSE file
     `git add html/` adds all new/changed files in subdirectory html
     directories are not tracked → no empty directories
     (use hidden empty dummy file e.g. `.gitkeep`)
     `git add -i` interactive add (choose files, stage „hunks",...)
  2. Commit staged changes into repository
     `git commit -m "Fixed bug ILM-1325: Wrong license"`
     The commit message is mandatory. If omitted the configured editor is opened.
- Both steps can be combined to commit ALL changes
  - `git commit -am "Fixed bug ILM-1325: Wrong license"`
  - Note: -a does not add untracked files

git status – Show status

- Shows the status of the working directory relative to the local repository
  - Current branch
  - Staged files
  - Unstaged changes
  - Untracked files
- Gives hints to change the state of files



git log – Show history

- Shows the commit history of the current branch
  - Entire history up to the root (paged)
    ```
    git log
    ```
  - The last 10 commits
    ```
    git log -10
    ```
  - Filter by date
    ```
    git log --since=2.weeks
    git log --since="2 years 1 day 3 minutes ago"
    ```
  - Show only commits of a specific path
    ```
    git log -- html/
    ```
  - Show only commits which changed specific content
    ```
    git log -Sfunction_name
    ```

git diff – Show changes

- Show content differences
  - unstaged changes
    ```
    git diff
    ```
  - staged changes
    ```
    git diff --cached
    ```
  - relativ to specific version
    ```
    git diff 176bc4    (first few chars of the id are sufficient)
    git diff HEAD^     (last commit)
    ```

git rm – Remove files

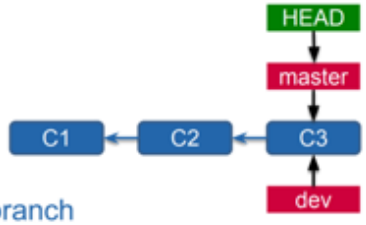
- Removing Files
  - untrack and delete file (→ delete)
    `git rm README`
  - untrack file but keep it in working directory (→ untrack)
    `git rm --cached README`
  - from staging area (→ unstage)
    `git reset HEAD -- README.md`
- Moving/Renaming Files
  - Content is not changed, only reference is moved in the tree
    `git mv README README.md`
  - Is technically the same as
    ```
    mv README README.md
    git rm README
    git add README.md
    ```

Config .gitignore – Ignore files

- Some files should not be tracked at all
  - e.g. config info, generated/compiled files, temporary files, caches, ...
  - add text file containing a list of files to ignore
    - `~/.gitignore` global user specific settings
    - `project-root/.git/info/exclude` project wide, not distributed to remotes
    - `project-root/.gitignore` project wide, shared with remote repos
    - `project-subdir/.gitignore` overrides settings in project-root
  - File list can use wildcards:
    - `/.project` ignores the .project file in project root
    - `*.html` ignores all files with extension *.html
    - `!index.html` do NOT ignore index.html

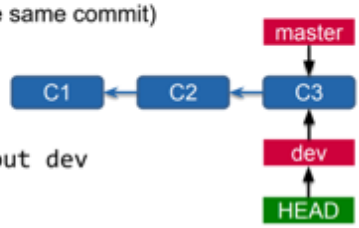
git branch – Create a branch

- In git a branch is a pointer to a commit
  → `master`
- Create a new branch:
  `git branch dev`
  (points to the current commit)
- HEAD always points to the current branch
  (checked out branch)



git checkout – Switch branch

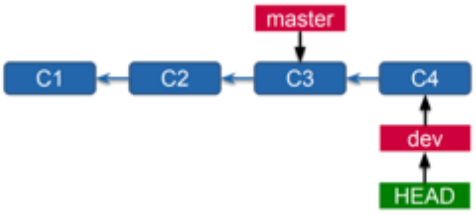
- Git checkout moves the HEAD pointer to the specified branch
  `git checkout dev`
- Replaces files in working directory with files of dev
  (not required here because they point to the same commit)




```
git checkout -b dev
→ git branch dev && git checkout dev
```

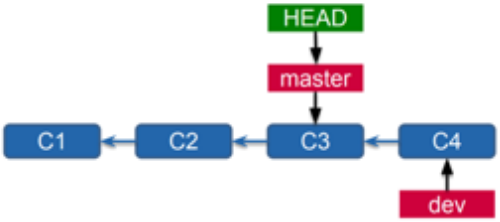
git checkout – Modify branch

- Committing changes on the new branch
  moves dev branch pointer ahead of `master`
  `git commit -am "C4 on dev branch"`



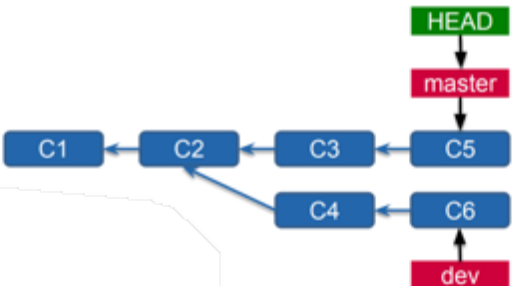
git checkout – Switch branch back

- Switch current branch back to master
  `git checkout master`
- Changes of C4 disappear
  in working directory



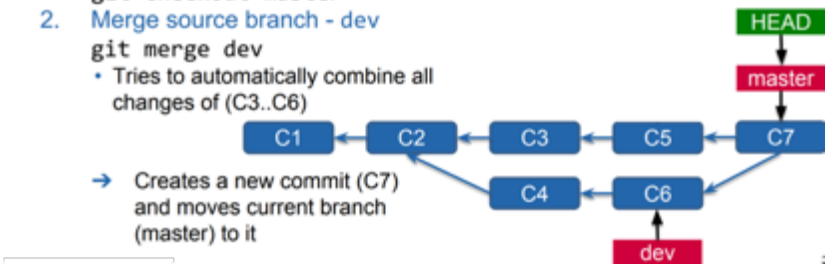
git merge – Merging two branches (1)

- After some time the two branches diverge
- Time to merge:
  1. Switch to target branch - `master`
     `git checkout master`



git merge – Merging two branches (2)

- After some time the two branches diverge.
- Time to merge:
  1. Switch to target branch - `master`
     `git checkout master`
  2. Merge source branch - dev
     `git merge dev`
     - Tries to automatically combine all changes of (C3..C6)



→ Creates a new commit (C7) and moves current branch (master) to it

Merge conflicts

- Sometimes the auto-merge/rebase algorithms fail
- Then conflicts have to be fixed by hand

```
$ git merge dev
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

- The merge commit is NOT created
- `git status` shows the conflicting files
- The conflicting files contain markups showing the differences

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">please contact us at support@github.com</div>
>>>>>>> dev:index.html
```

- Fix all the sections either by taking either one or the other version, or replace it with a combined solution:

```
<div id="footer">please contact us at email.support@github.com</div>
```

- Remove the separators lines `<<<<<<<<<`, `=========` and `>>>>>>>>>`
- Alternatively you can use an interactive merge tool:
  `git mergetool`
  - The preferred merge tool can be set in the configuration: `merge.tool`
  - If none is set, the default built-in tool is used.
- If all conflicts are fixed, mark them as resolved and continue the commit:
  `git add index.html`
  `git commit`

git remote – Remote Operations

- Until now we have been working only on the local repository
- One of Git's strength is to **sync local repositories with remote repos**
- Show all configured remotes

```
$ git remote -v
origin    https://github.engineering.zhaw.ch/mach/PROG2.git (fetch)
origin    https://github.engineering.zhaw.ch/mach/PROG2.git (push)
```

- Adding a new remote
  - Syntax: `git remote add <nickname> <url>`
    Example: `git remote add test-area git@github.com:jdoe/test.git`
- Removing a remote
  - Syntax: `git remote remove <nickname>`
    Example: `git remote remove test-area`

git clone – Create a copy of a repository

- To create a copy of the repository it can be cloned
  - Form: `git clone <URL> [<localdir>]`
    Example: `git clone https://github.com/jdoe/test.git jdoe`
- If you clone a repository the following happens

– A local directory with an empty git repo is created	`git init <localdir>`
– Add remote with name origin (default)	`git remote add origin <url>`
– Fetch active branch from origin	`git fetch origin`
– Add tracking branch	`git branch master origin/HEAD`
– Checkout active branch	`git checkout master`

- After the clone
  - A plain `git fetch` fetches all the remote (tracked) branches
  - A plain `git pull` will in addition merge the remote active branch into the local active branch

git fetch and pull – Update from a remote repository

- **Fetch**
  Retrieve info from a remote that is not in the local repository
  - Syntax: `git fetch [<remote>] [<branch>...]`
    Example: `git fetch origin`
    - Gets all references of remote branches
    - Tracked branches are updated automatically
    - Nothing is merged
    - If "remote reference" is missing the configured upstream "remote" is used, otherwise "default reference (origin)"
- **Pull** (= Fetch + Merge)
  Fetch info from remote repository and merge active branch
  - Syntax: `git pull [<remote>] [<branch>]`
    Example: `git pull origin master`

```
→ git fetch origin master && git merge
```

git push – Publish to a remote repository

- First check that your local branch is ahead of the remote branch
  - Verify that no changes on remote branch are not yet merged into local branch
  - Do a `git pull` first and fix conflicts
- **Push** local <branch> to <remote>/<remote-branch>
  Adds the branch on remote if it does not exist
  - Syntax: `git push <remote> <remote-branch>`
    Example: `git push origin master` (push to master branch on remote origin)
- Set default upstream for active branch (add as a tracking branch)
  - Syntax: `git push -u <remote> <remote-branch>`
    Example: `git push -u testarea master` (push and set tracking branch)
- Push (current or all) tracked remote branches
  - Syntax: `git push [--all]`
    Example: `git push` (push current branch to matching remote branch)
    `git push --all` (push all local branches to tracked remote branches)

6.5 Gitflow Workflow

Gitflow is one amongst many workflows which can be applied to a project. It was first published and made popular by Vincent Driessen at nvie. It defines a strict branching model and is designed around the project releases. It provides a robust framework and is meant to be for larger projects with a scheduled release cycle, enabling many developers to simultaneously work on the same project in an efficient manner.

Develop and Master branch

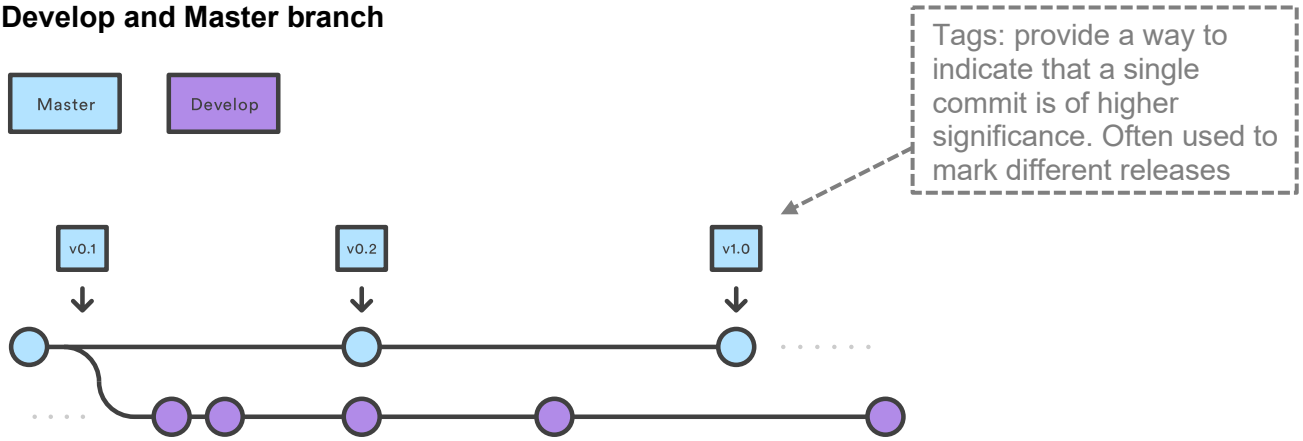


Figure 61 Gitflow develop and master branch (atlassian.com, DAO 25.12.18)

As seen above, this workflow uses two branches to record the history of a project, rather than one single master branch. The master branch's duty is to store and maintain the official release history. The develop branch on the other hand, serves as an integration branch for features. It is the branch the developers are actively working on.

Feature branch

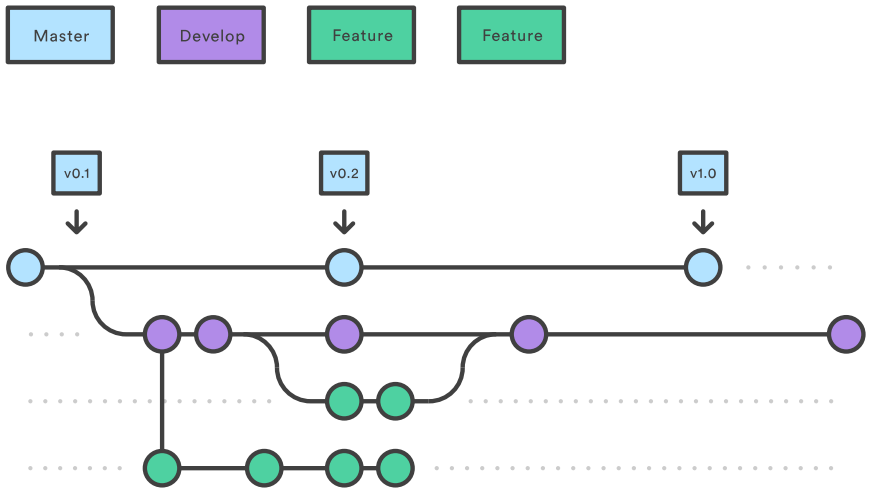


Figure 62 Gitflow feature branches (atlassian.com, DAO 25.12.18)

Alterations (formerly called features) are changes that reside in their own individual branch. However, instead of branching off of the master branch, feature branches use develop as their parent branch (because all development activity is happening there). For bigger features there one may also create sub-feature branches. As soon as a feature is finalized, it gets merged back into develop (or its parent branch). Features should never directly interact with the master.

Release branches

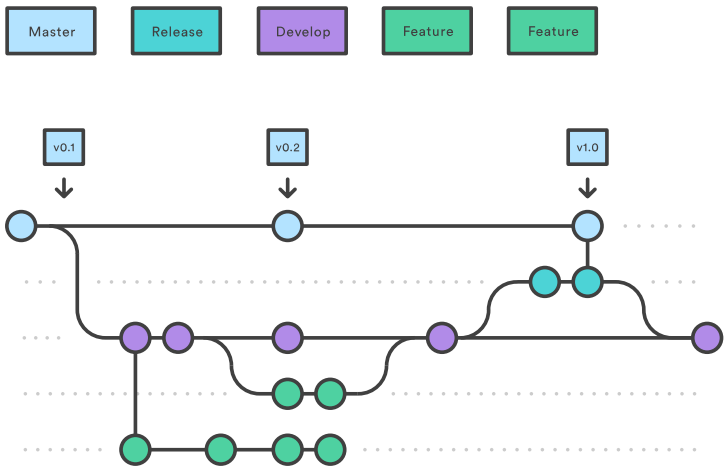


Figure 63 Gitflow release branches (atlassian.com, DAO 25.12.18)

At one stage the develop branch has accumulated enough features for a new release. In this case, you fork a release branch off of the develop branch. Creating such a branch creates the next release cycle. Apart from bug fixes (hot fixes), documentation generation and other release oriented tasks, there are no features to be added after this point. Once everything is ready for a rollout, the release branch gets merged into the master and the resulting merge commit is tagged with a version number. You should also make sure to merge it back into the develop, in order to also integrate any hotfixes etc. into it.
This dedicated approach enables teams to both polish the current release and continue working on features for the next release.

Hotfix branches

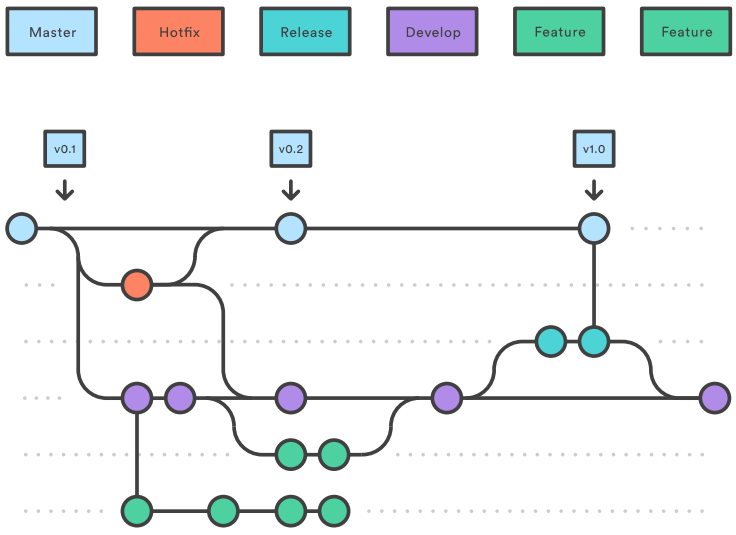


Figure 64 Gitflow Hotfix branches (atlassian.com, DAO 25.12.18)

Maintenance or so called hotfix branches are used to quickly patch production releases. This is the only branch that should directly be forked off the master. Once the fix is finalized and complete, it should then be merged back into both master and develop or the current release branch. The merge commit on the master should be tagged with an updated version number.

6.5.1 Collaboration model

There are multiple collaborative models to use with Git. We look at the branch and pull approach.

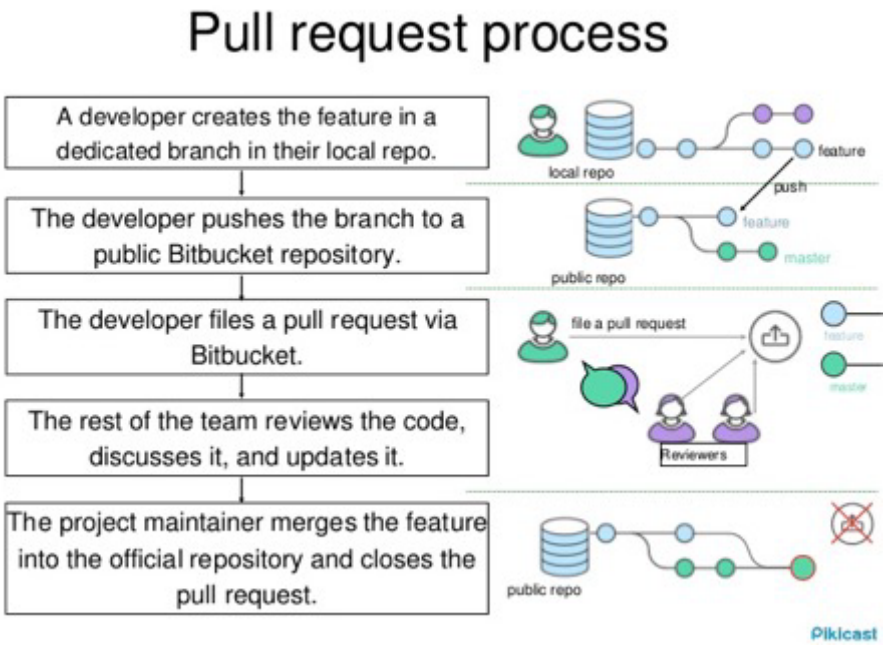


Figure 65 Branch and pull collaboration model (slideshare.net Pham Quy, DAO 25.12.2018)

6.6 Appendix

6.6.1 A short history of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.
The Linux kernel is an open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development.

6.6.2 Re-Hashed: SHA-1 and SHA-2

What is a Hash?
A hashing algorithm is a mathematical function that condenses data to a fixed size.
As an example, if we took the sentence: "The Quick Brown Fox Jumps Over The Lazy Dog" and ran it through a specific hashing algorithm known as CRC32, the result would be: "07606bb6".
This result is known as a hash or a hash value and sometimes hashing is referred to as a one-way encryption.

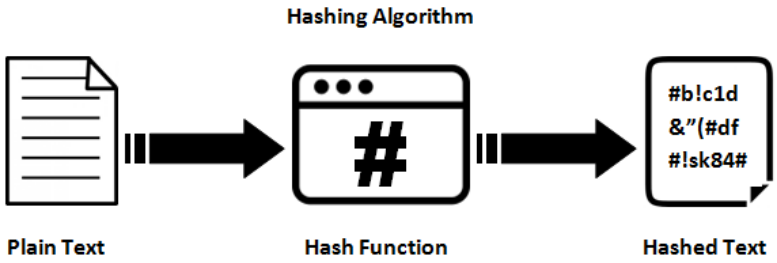


Figure 66 Example Hashing Algorithm (www.thesslstore.com, DOA 16.01.2019)

It is easier for the computer to compute a hash and then compare the values than it would be to compare the original files.

If you only take away one thing from this section, it should be: cryptographic hash algorithms produce irreversible and unique hashes.

Secure Hashing Algorithm
SHA-1 and SHA-2 are two different versions of that algorithm. They differ in both construction (how the resulting hash is created from the original data) and in the bit-length of the signature. You should think of SHA-2 as the successor to SHA-1, as it is an overall improvement.

Primarily, people focus on the bit-length as the important distinction. SHA-1 is a 160-bit hash. SHA-2 is actually a "family" of hashes and comes in a variety of lengths, the most popular being 256-bit.

The SHA-2 "family":
- SHA-2
- SHA-256 or SHA-256 bit

alternate bit length:
- SHA-224
- SHA-384
- SHA-512

Example, how a may website looks with SSL Certification:

www.thesslstore.com.cer	(application/x-x509-ca-cert) - 2042 bytes
SHA-1	2377b8642eac9f622b826f7a83500704f3cc3488
SHA-256	0feb43ff09dbea9487421fd3930527828703995e9f1d95d01656e3474cff2493

Figure 67 SSL Certification (www.thesslstore.com, DOA 16.01.2019

7 Tooling – Build Automation

7.1 Chapter objectives

You are familiar with …

- the concepts of software automation
- the basic principles of build automation
- the build automation tool Gradle

7.2 Benefits of using software automation

What does the term software automation actually imply?
“Act of scripting or automating a wide variety of tasks that software developer and operator do in their day-to-day activities”

Figure 68 Build automation (cyberciti.biz, DOA 25.12.2018)

If developers would build their large scaled applications locally, they would soon run into issues like:

- Lack of versioning
- No consistent execution of unit tests
- Unknown state of build
- Undefined dependencies between components
- Non-transparent deployment
- Prone to errors
- Repeating tasks
- Lack of documentation

Therefore, main goals of software automation could be defined as such:

Improve Product Quality

- Automated testing
- Automated code auditing
- Documented history of builds to verify issues

Faster time to market

- Accelerate the process from building to deployment
- Immediate feedback and shorter innovation cycle

Minimize risks

- Proof that software is building
- Find broken build efficiently
- Awareness of current software status
- Eliminate dependencies on key personnel

7.3 Software automation pipeline

Automation can be implemented in every stage of the software cycle (dev, integration, QA, operation). Individual modules can only continue if the previous steps have passed successfully beforehand. In case of failure, the responsible people are notified about existing errors and the process is halted until fixed.

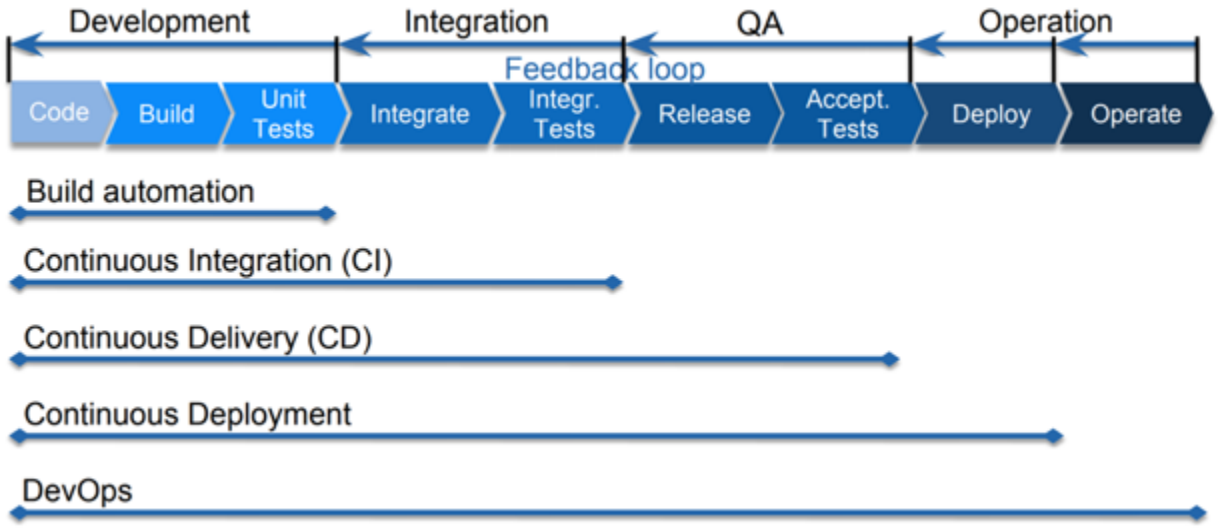


Figure 69 DevOps approach (ZHAW Prof. Dr. M. Bohnert, DOA 25.12.18)

Build automation

- Building individual components, resolving dependencies, run unit tests, package software, create documentation, clean-up. Tasks are typically run by the developer on his local machine.

Continuous Integration

- Automatically build, test and integrate components and run Integration Tests (Code auditing, Security tests, Database tests, UI tests, …). Tasks are typically run on CI servers.

Continuous Delivery

- Also create releases, deploy to staging environment and run automatic acceptance tests (Stress test, Load Tests, Compliance tests, …). Ready for production, but deployment still required a manual step.

Continuous Deployment

- Automatically deploy to production after successful passing acceptance tests.

DevOps

- Automatically run the operation of the production system (configuration management, infrastructure provisioning, backup, monitoring, health management, scaling, …)

7.4 Build automation tools

Depending on your Software environment different tools are common. Possible examples below:

– C, C++:	make, gmake, SCons, distcc,…
– Python:	Waf, Snakemake, …
– Ruby:	Rake,…
– JavaScript/Node.js:	Grunt, Gulp, Bower,…
– Microsoft Environment:	MS Build, Visual Build, Psake,…
– Java, JVM languages:	Apache Ant, Apache Maven, Gradle

A majority of the tools work for multiple environments however. As you will work with a java stack during this module, Ant, Maven as well as Gradle should concern you the most.

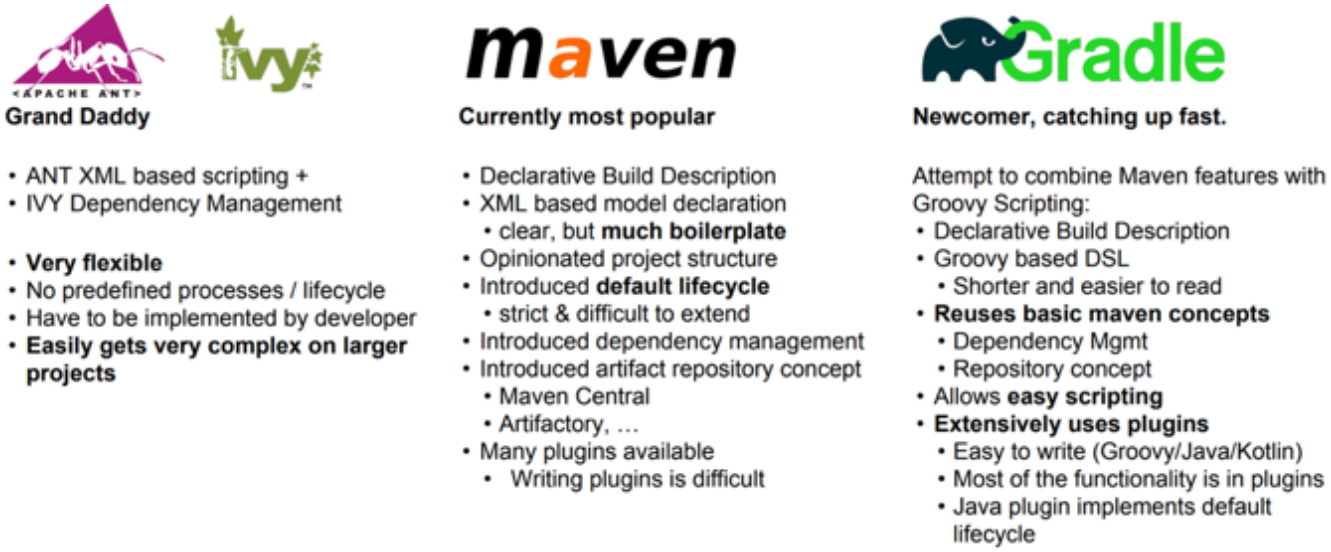


Figure 70 Java, JVM languages build tools (ZHAW Prof. Dr. M. Bohnert, DOA 25.12.18)

7.5 Gradle

7.5.1 Overview

In the middle ground between configuration and convention, Gradle a DSL groovy-based build automation tool. Gradle is a powerful tool and we can simply not cover all the logic in the course. But just like Git, we will acquire the knowledge needed to use the environment in everyday tasks.

7.5.2 Building blocks – Projects and tasks

In Gradle, builds consist of one or more projects and each project holds one or more tasks.
A task can be described as a single piece of work. Compiling classes, executing tests or creating and publishing web archives are only a few possibilities among many others.
The following snippet illustrates a possible task:

```
task hello {
    doLast {
        println 'Module223'
    }
}
```

The above task can be executed using the gradle –q hello command.

You can even go further and define tasks a little more complex. Remember Gradle's scripts are nothing but Groovy:

```
task toLower {
    doLast {
        String someString = 'Welcome to module 223'
        println "Original: "+ someString
        println "Lower case: " + someString.toLowerCase()
    }
}
```

What about defining a task that depends on another one? DependsOn: taskName gives you just that:

```
task helloGradle {
    doLast {
        println 'Hello Gradle!'
    }
}

task fromBaeldung(dependsOn: helloGradle) {
    doLast {
        println "I am currently learning how to use you"
    }
}
```

Let us define a task and enhance it with some additional behaviour:

```
task helloGradle {
    doLast {
        println 'I will be executed second'
    }
}

helloGradle.doFirst {
    println 'I will be executed first'
}

helloGradle.doLast {
    println 'I will be executed third'
}

helloGradle {
    doLast {
        println 'I will be executed fourth'
    }
}
```

Last but not least, properties:

```
task ourTask {
    ext.theProperty = "aValue"
}
```

7.5.3 Managing plugins

Gradle supports multiple types of plugins. We will only look at the script however.
To benefit from such an additional functionality, every plugin goes through two steps: resolving and applying.

Resolving takes care of finding the correct version of the plugin jar and of adding that to the corresponding classpath of the project.
Applying is then executing Plugin.apply(T) on the project.

Let us create a possiblePlugin.gradle and define a task within:

```
task fromPlugin {
    doLast {
        println I get executed from a plugin
    }
}
```

We could now go and apply this plugin to our current project build.gradle file. All we have to is writing the following line of code:

```
apply from: 'possiblePlugin.gradle'
```

Check all your declared tasks by using the command gradle tasks.

7.5.4 Dependency management

Now that we acquired a broad understanding of the fundamentals of Gradle projects and tasks, we can go further ahead and look at managing dependencies. It is done in two steps:

- Specifying the repositories to search for the dependencies
- Specifying the dependencies of the project

The following example gives you an idea of what a build.gradle file with defined dependencies could look like in reality:

```
apply plugin: 'java'

repositories {
  mavenCentral()
}

dependencies {
  compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
  testCompile group: 'junit', name: 'junit', version: '4.0'
}
```

Dependencies can hereby be declared in various formats:

- compile: Dependencies required to compile the production source of the project.
- runtime: Dependencies required by production classes at runtime. Also includes the compile time dependencies.
- testCompile: Dependencies required to compile the test source of the project. Also includes the compiled production classes and the compile time dependencies.
- testRuntime: Dependencies required to run the tests. Also includes the compile,runtime and test compile dependencies.

7.5.5 Gradle Wrapper

To be able to build a project without an actual installation of Gradle, we can get gradlew,sh file for linux or the gradlew.bat for windows. If we execute gradlew build in windows or ./gradlew build in linux, a Gradle distribution specified in gradlew file will be downloaded automatically.

A Wrapper can be added as such:

```
gradle wrapper --gradle-version 4.2.2
```

By executing this command in the root of the project, we will get all necessary files and folders to tie Graddle wrapper to the project.

The same can be achieved with another approach:

```
task wrapper(type: Wrapper) {
    gradleVersion = '4.2.2'
}
```

Now we added the wrapper task to the actual build script. Besides the gradlew files, a wrapper folder is generated inside the gradle folder containing a jar and a properties file.

If we want to switch to a new version of Gradle, we only need to change an entry in gradle-wrapper.properties.

7.6 Appendix

7.6.1 Evolution of java build tools

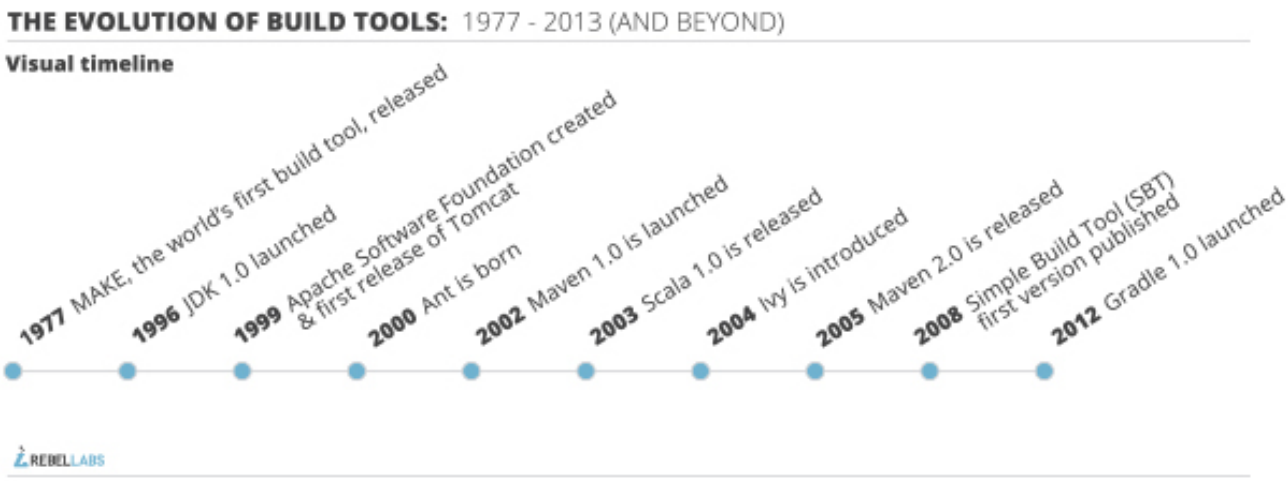


Figure 71 The evolution of build tools (zeroturnaround.com, DOA 25.12.2018)

7.6.2 Groovy

Apache Groovy is a Java-syntax-compatible object-oriented programming language for the Java platform. It is both a static and dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as both a programming language and a scripting language for the Java Platform, is compiled to Java virtual machine (JVM) bytecode, and interoperates seamlessly with other Java code and libraries. Groovy uses a curly-bracket syntax similar to Java's. Groovy supports closures, multiline strings, and expressions embedded in strings.

7.6.3 Domain specific language (DSL) vs general purpose language (GPL)

We want to keep the input minimal here. A domain specific language (DSL) is essentially a specialized computer language designed solely for a specific task. General purpose language like C, Python or Haskell are designed to let you code any sort of program with any sort of logic you need. Generally said, DSLs can be less powerful than GPLs but are however tailored and meant to be for a specific domain. Here are a few DSLs to be mentioned:

- CSS is a domain specific language designed to style HTML on websites. All it does is specifying rules for how HTML tags are to be displayed.
- SQL is a DSL as well. It was created for the purpose of querying databases.
- Same goes for Cron. It gives you the option to schedule tasks and instruct them on when and how often they should run.

8 Application Security

8.1 Authentication, Authorization

8.2 JSON Web Tokens - JWT

8.3 Security Leaks

8.3.1 XSS

8.3.2 CSRF

8.3.3 SQL Injection

9 Spring Basics

9.1 IoC and Dependency Injection - Theory

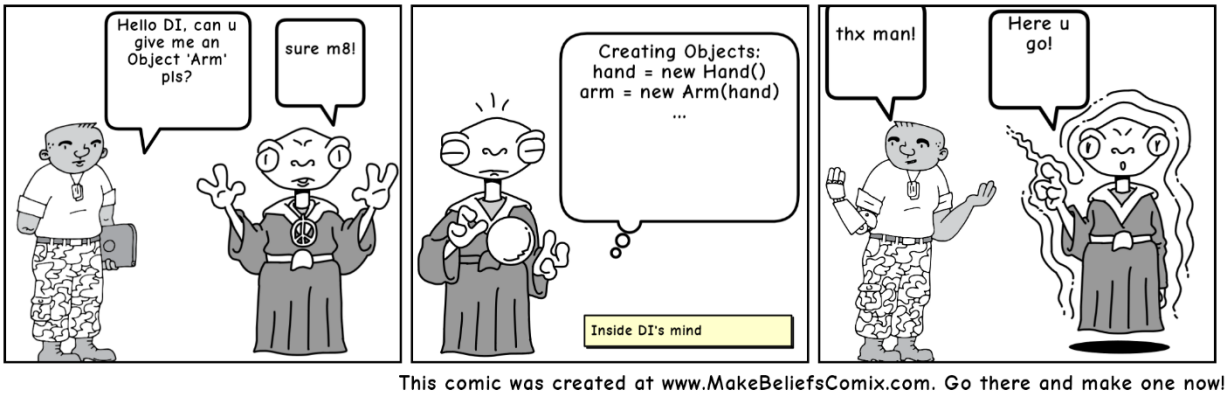


Figure 72 Dependency injection comic

Inversion of control (IoC) is a programming principle where an application delegates the control over its flow to some kind of generic framework or other third party. Compared to traditional programming where custom code, expressing an application's purpose, calls into reusable libraries to take care of generic tasks, with inversion of control, a generic framework or other third-party calls into the custom i.e. task-specific code. This essentially moves software from a purely imperative style, where the programmer expresses a clear sequence of steps that should be performed, to more of a declarative style, where the programmer expresses what he wants, and the exact sequence of steps gets 'figured out' for him. Take a web server for example: Without IoC, the programmer would write the code for opening a socket, listening for HTTP requests, parsing incoming HTTP requests and calling the correct handler methods himself. He might utilize some libraries along the way to make things easier but the control of the program's flow [set up socket > wait for request > parse request > handle] remains completely with him. With IoC however, the programmer would only define the general 'structure' of his web service (e.g. for this path call this method, use this method to configure the server, use this method to handle certain exceptions, et cetera). The control over the application's flow remains with the framework at all times.

As an introduction to dependency injection, consider following ELI5's (the first one even made it into Mark Seeman's book: Depedency Injection in .Net):

> When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.
>
> What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.
>
> – John Munsch on stack overflow

> Basically, What you're doing with dependency injection is passing a service as a parameter. Where without it you would create an instance of a service inside a method. Instead of doing this, you instantiate the service outside, and pass it into the client. That way the client does not have to be aware of the service, it only needs to be aware of the interface of the service, or how to use it.
>
> So why? Well you might have the need to get information within a client. Let's say you have multiple paths to get data using a service. The client doesn't care where this information comes from, only that it can use the service. So you choose to abstract the construction of the service outside the client, where you have some logic to determine what type of service you need at that particular moment, and just pass the service into the client where it can be used without worry that it's getting information from the best source.
>
> When we say injection, we're really just talking about passing parameters. We're implying that we are actively trying to isolate parts of the system from certain tasks which can be better handled elsewhere.
>
> – u/CoopNine on reddit

Dependency injection is a form of IoC in which the creation of objects or more specifically the flow needed to perform object creation is outsourced to a dependency injection framework. This includes the creation of an object's dependencies. Dependencies are objects that can be used (services) / are used by another object. In other words, an object might (most time 'does') not function properly without all of its dependencies. Consider the two versions of the 'Stuff' class below:

```
class Stuff {
    Dependency dep;

    public Stuff(Dependency dep) {
        this.dep = dep;
    }
}
```

Class conforming with DI principles

```
class Stuff {
    Dependency dep;

    public Stuff() {
        this.dep = new Dependency();
    }
}
```

Class violating DI principles

Why use dependency injection?

- Makes it easy to follow the single responsibility and open-closed principles
- Helps to avoid strong coupling between components, improving the flexibility of an application
- Automatically makes all classes testable (in unit tests through mocking)
- Makes code cleaner and more readable (transparent dependencies)

9.2 Dependency Injection in Spring

To demonstrate how DI can be used in Spring we will create a small exemplar application. The application will be able to create car objects that are composed of various car parts and then print them to the console. Our cars will be composed according to the class diagram on the right of this text.

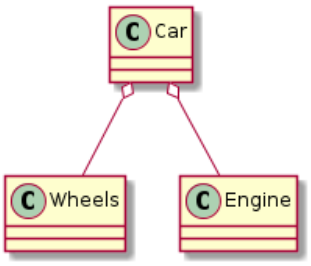


Figure 73 DI project class diagram

The Application Context

The `ApplicationContext` is the central interface within a Spring application for providing configuration information to it. A number of classes implement the `ApplicationContext` interface, allowing for a variety of configuration options and types of applications. The `ApplicationContext` provides factory methods for accessing application components/beans and manages the lifecycle for managed beans (explanation ahead!). In earlier versions of Spring, all beans had to be defined in an XML document and a `ClassPathXmlApplicationContext` would be used to read the bean definitions from the file and make the specified beans accessible through the `ApplicationContext` interface. In later spring versions however, this approach was deprecated in favour of defining beans with Java annotations. To get started with this approach we have to create an `AnnotationConfigApplicationContext` object and pass one or more configuration classes to it. Spring would then take a look at the annotations of these classes and configure the application context according to them. In the following example we create an `AnnotationConfigApplicationContext` and pass our Main class to it. This class is annotated with two annotations `@Configuration` and `@ComponentScan`. The latter one tells Spring that it should scan the given packages [in our case: `demo`] for classes which are defined as components/beans. We will investigate the meaning of `@Configuration` later in this course. Note that we could also tell spring that it should perform a package scan by simply passing the packages to scan (as `String`) to the constructor of the `AnnotationConfigApplicationContext` class.

```
@Configuration
@ComponentScan("demo")
public class Main {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(Main.class);

        // close the context
        context.close();
    }

}
```

Defining Components

In Spring, components can be defined by annotating classes with `@Component`. When performing a component-scan, Spring would register classes annotated with `@Component` and register them in the IoC container [application context]. In our example application we would want our Car, Wheels and Engine to be registered as components so that we could then access them through our application context. To do this we can write the classes as follows:

```
@Component
public class Engine {

    public void doEngineStuff() {
        System.out.println("Doing totally normal engine stuff");
    }

}
```

```
@Component
public class Wheels {

    public void roll() {
        System.out.println("ROLL! ROLL! ROLL! ROLL!");
    }

}
```

```
@Component
public class Car {

    private Wheels wheels;

    private Engine engine;

    public Car(Wheels wheels, Engine engine) {
        this.wheels = wheels;
        this.engine = engine;
    }

    public void drive() {
        System.out.print("Now driving:\n\t");
        engine.doEngineStuff();
        System.out.print('\t');
        wheels.roll();
    }

}
```

This would register our three classes as beans / components in our application context (because of the component-scan). In our main method, we could now actually obtain instances of our `Engine` and `Wheels` classes by calling the `getBean(Class)` method on our context. Consider following example and its output:

```
// inside our main method (see example above)
Engine engine = context.getBean(Engine.class);
engine.doEngineStuff();
```

```
output:

Doing totally normal engine stuff
```

Wiring components up

Now that we've declared our classes to be components, we need to tell Spring how to wire them up i.e. how to do the actual dependency injection. In our case Spring would need to inject an `Engine` and a `Wheels` instance into a new `Car` instance. To do this we can use the `@Autowired` annotation. There are three different ways that this annotation can be utilized:

```
@Autowired
private Wheels wheels;

@Autowired
private Engine engine;
```

```
@Autowired
public void setWheels(Wheels wheels) {this.wheels = wheels;}

@Autowired
public void setEngine(Engine engine) {this.engine = engine;}
```

```
@Autowired
public Car(Wheels wheels, Engine engine) {
    this.wheels = wheels;
    this.engine = engine;
}
```

There is no consensus in the developer community about which of these three ways of performing DI is best, but field injection is generally frowned upon. At first glance it might seem like field injection is the 'nicest' way to do it since there is no additional boilerplate code needed. However, using field injection comes with some problems: Classes become strongly coupled to the DI framework since fields are usually made private and thus making it impossible to create functional instances of a class without the DI or some kind of reflection (good luck creating instances with mocked dependencies in unit tests…); It cannot be used in conjunction with final fields (only constructor injection can); It hides a class' dependencies from 'the public' and overall performance might suffer when using it.
Now, as for the question of whether to use constructor or setter injection, following Guidelines were expressed in the Spring docs:

- For mandatory dependencies or when aiming for immutability, use constructor injection. Generally, a class should be fully functional after calling its constructor.
- For optional or changeable dependencies, use setter injection. A class' functionality shouldn't depend on dependencies injected through setters.

In our example we're going to use constructor injection because a car isn't fully functional without its engine and its wheels. To do this we simply add `@Autowired` to the constructor of our `Car` class.

Loose coupling with the help of interfaces

A common design pattern when utilizing dependency injection is to remove strong coupling between components by using interfaces. This way, components don't need to specify the actual implementations of their dependencies but can only refer to the interface that any dependencies would need to implement. This effectively delegates the question of which implementations are used to the DI framework. This makes it easy to switch between different implementations; Overall increasing the flexibility of our application and making it easier to adapt software to new requirements.

In our application we could utilize this principle enable for a looser coupling between our car and our engine, because we might want to change it sometime in the future. For this we'll need to rename our plain `Engine` class to something more specific (so that we can name our interface 'Engine'). Let's just call it `CombustionEngine`. When we've done that, we can define our interface (assuming the classes are in the same package the 'Car' class can be left as-is):

```
public interface Engine {
    void doEngineStuff();
}
@Component
public class CombustionEngine implements Engine {

    @Override
    public void doEngineStuff() {
        System.out.println("Doing totally normal engine stuff");
    }

}
```

Now when requesting an instance of type 'Engine', Spring would look for known implementations of the 'Engine' interface and return us a 'CombustionEngine' instance (since it's the only implementation). But what if we want to add other possible engines to our application? Let's try adding an 'ElectricMotor':

```
@Component
public class ElectricMotor implements Engine {

    @Override
    public void doEngineStuff() {
        System.out.println("Electricity stuff going on");
    }

}
```

But if we try to run our application now, we get the following:

```
Unsatisfied dependency expressed through constructor parameter 1; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'demo.Engine' available:
expected single matching bean but found 2: combustionEngine,electricMotor
        at org.springframework.beans.factory.support.ConstructorResolver.createArgumentArray(ConstructorResolver.java:769)
        at org.springframework.beans.factory.support.ConstructorResolver.autowireConstructor(ConstructorResolver.java:218)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireConstructor(AbstractAutowireCapableBeanFactory.java:1325)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBeanInstance(AbstractAutowireCapableBeanFactory.java:1171)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:555)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:515)
        at org.springframework.beans.factory.support.AbstractBeanFactory.lambda$doGetBean$0(AbstractBeanFactory.java:320)
        at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:222)
        at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:318)
        at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:199)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons(DefaultListableBeanFactory.java:849)
        at org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplicationContext.java:877)
        at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:549)
        at org.springframework.context.annotation.AnnotationConfigApplicationContext.<init>(AnnotationConfigApplicationContext.java:88)
        at demo.Main.main(Main.java:14)
Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'demo.Engine' available: expected single matching bean but found 2: combustionEngine,electricMotor
        at org.springframework.beans.factory.config.DependencyDescriptor.resolveNotUnique(DependencyDescriptor.java:221)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:1225)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java:1167)
        at org.springframework.beans.factory.support.ConstructorResolver.resolveAutowiredArgument(ConstructorResolver.java:857)
        at org.springframework.beans.factory.support.ConstructorResolver.createArgumentArray(ConstructorResolver.java:760)
        ... 14 more
```

```
expected single matching bean but found 2: combustionEngine,electricMotor
```

Here Spring doesn't know which of the two implementations to pick, so we need to manually specify the implementation to be used. In order to resolve such ambiguities, Spring names all of its beans, so that they are not only identified by their types but also by their names. Actually we've already seen the names of our two implementations in the error message shown before: `combustionEngine` and `electricMotor`. By default, a bean is named after the implementation's class name in Dromedary Case (a.k.a. lower camel case) but they can also be given a custom name using the value attribute of the `@Component` annotation. To retrieve beans by their name using the application context the method `getBean(String name, Class requiredType)` can be used. Retrieving beans by their name in conjunction with dependency injection can be done with the `@Qualifier` annotation:

```
@Autowired
public Car(Wheels wheels, @Qualifier("electricMotor") Engine engine) {
    this.wheels = wheels;
    this.engine = engine;
}
```

Alternatively, the `@Primary` annotation can be placed on a bean class / custom bean definition to mark it as the preferred bean of its type.

Defining custom beans

When using the annotations covered above, Spring will create our beans generically with its own code using the information we provided through the annotations. However, we can also supply Spring with a custom method for creating our beans which it will then use instead. This might be useful in cases where beans need some additional configuration or are impossible / unnecessarily difficult to assemble using the default Spring logic.
Supplying custom code for bean assembly can be done with the `@Configuration` annotation. It allows do define custom beans by annotating bean creation methods with `@Bean` (the bean will be named after the method name):

```
@Configuration
public class BeanConfig {

    @Bean
    public Car hoverCar(@Qualifier("electricEngine") Engine engine) {
        var wheels = new Wheels() {
            @Override public void roll() {
                System.out.println("hover!");
            }
        };

        return new Car(engine, wheels);
    }

}
```

Bean scopes

Each Spring bean is bound to a so-called bean scope. It defines the visibility and some lifecycle aspects of a bean.
Out of the box, the latest version of the Spring Framework supports the following six bean scopes:

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container. (i.e. only one instance is ever created per application instance)
prototype	Scopes a single bean definition to any number of object instances. (i.e. each time a bean is requested a new instance gets created)
request*	Scopes a single bean definition to the lifecycle of a single HTTP request. That is one and only one new instance is created per request (~singleton)
session*	Scopes a single bean definition to the lifecycle of an HTTP `Session`. (~singleton)
application*	Scopes a single bean definition to the lifecycle of a `ServletContext`. (~singleton)
websocket*	Scopes a single bean definition to the lifecycle of a `WebSocket`. (~singleton)

* Only valid in the context of a web-aware Spring ApplicationContext.

Generally, the `singleton` and `prototype` scopes are used most of the time. Following graphic explains their difference further (singleton top; prototype bottom). It is also to be noted that in contrast to the prototype scope, the singleton scope's lifecycle is fully managed. This means that for prototype beans only the lifecycle callbacks (hooks) in the creation phase but not in the destruction phase get called (while for singleton beans all lifecycle callbacks get called).

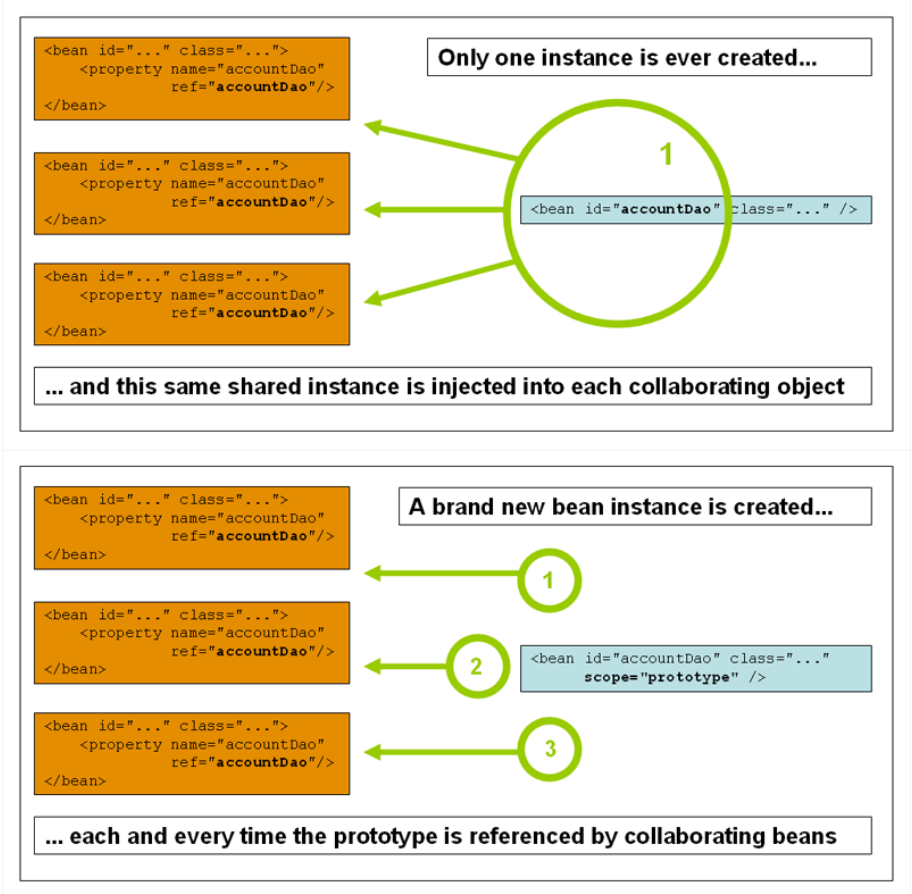


Figure 74 Singleton vs prototype scope (docs.spring.io, DOA 12.03.19)

10 Spring Boot

10.1 The Spring initializr

Spring provides a tool for easily creating new spring boot applications; The spring Initializr. To use it go to https://start.spring.io/, select the appropriate parameters and generate a template of your first real application. For our purposes be sure to include the web, jpa and test dependencies.

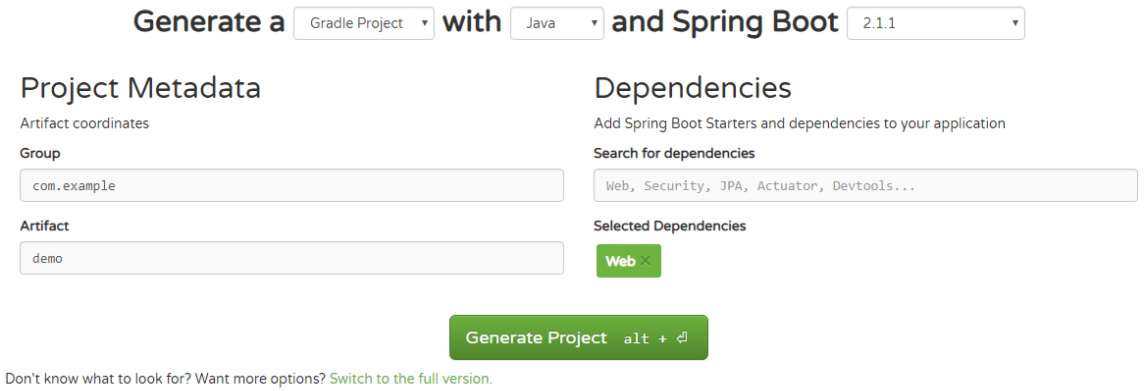


Figure 75 Setup spring initializr (start.spring.io, DOA 07.01.18)

Your main class should look something like this:

```
@SpringBootApplication
@ComponentScan
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

10.2 Architecture overview

Spring boot applications (generally) have the structure of a three-layered web application (as discussed in the architecture chapter). Spring boot applications have three main components: Controllers which define the interface to the outside world, Services which contain an application's business logic and Repositories which control the access to persistent data stores [databases]. Each layer can only communicate with its directly adjacent layers. For inter-layer communication so-called entities or domain models are sent around. To save bandwidth and conceal confidential fields, domain transfer objects (DTOs) are (generally) used to encapsulate data between the controller layer and the users of a system. They are nothing more than slight variations of the domain models, presenting them in a more convenient way for outsiders. Following graphic depicts an overview of a Spring Boot RESTful application:

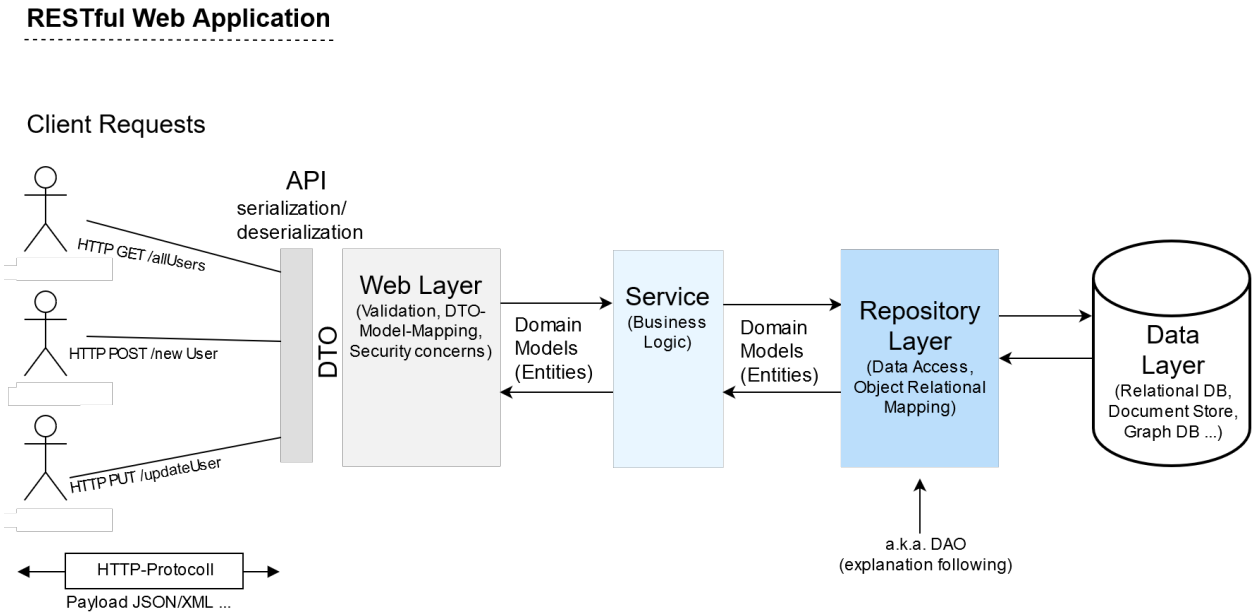


Figure 76 Spring Boot RESTful application overview

10.3 Controllers

By defining REST endpoints, we essentially grant access to our web application. Endpoints can be thought of as being the “methods” of an API which are accessible from the outside (i.e. accessible over HTTP). Controllers are objects which define the aforementioned endpoints and (hopefully) group them together in a meaningful way. To create a controller in spring, a class must be annotated with `@Controller` or one of its ‘sub-annotations’ such as `@RestController`, which also automatically applies the `@Component` annotation to it. Controllers can also have a so-called parent mapping, which is a path that is prepended to all endpoints defined in a controller. It can be supplied using the `@RequestMapping` annotation.

To define the actual endpoints, methods of a controller class can be annotated with `@RequestMapping` or a more concrete annotation like `@GetMapping`, `@PostMapping` or `@DeleteMapping`. These annotations all take the desired request mapping or path through their `value` fields. Using them on methods marks these as handlers for their specified paths, meaning that whenever an HTTP-Request to an endpoint’s path (parent mapping + endpoint mapping) is made, the appropriate handler method is called.

Take the following class as example; It defines two endpoints that can be accessed on the paths `/courses` and `/courses/{id}` where `{id}` can be any number which will be copied into the parameter id when called. Note that the request mapping defined for the controller [`/courses`] is prepended to the request mapping of every endpoint defined in that controller.

```
@RestController
@RequestMapping("/courses")
public class CourseController {

    /**
     * This method returns the requested course
     *
     * @param  id
     * @return    ResponseEntity with the course that was requested
     */
    @GetMapping("/{id}")
    public @ResponseBody ResponseEntity<Course> getById(@PathVariable long id) {
        Course course = new Course(id,"module223");
        return new ResponseEntity<>(course, HttpStatus.OK);
    }

    /**
     * This method returns all courses
     *
     * @return ResponseEntity with all existing courses
     */
    @GetMapping({ "", "/" })
    public @ResponseBody ResponseEntity<Iterable<Course>> getAll() {
        Course course1 = new Course(1L,"module223");
        Course course2 = new Course(2L,"module223");
        return new ResponseEntity<>(Arrays.asList(course1,course2), HttpStatus.OK);
    }
}
```

10.3.1 Validation

In most applications, one would want some form of validation for incoming data. Spring boot provides us with two different ways to do it:

Using the 'Validator' interface

This interface is the backbone of all validation operations in spring. It defines two methods, `supports` and `validate`: (you're supposed to read the Javadoc now …)

```
public interface Validator {

    /**
     * Can this {@link Validator} {@link #validate(Object, Errors) validate}
     * instances of the supplied {@code clazz}?
     * <p>This method is <i>typically</i> implemented like so:
     * <pre class="code">return Foo.class.isAssignableFrom(clazz);</pre>
     * (Where {@code Foo} is the class (or superclass) of the actual
     * object instance that is to be {@link #validate(Object, Errors) validated}.)
     * @param clazz the {@link Class} that this {@link Validator} is
     * being asked if it can {@link #validate(Object, Errors) validate}
     * @return {@code true} if this {@link Validator} can indeed
     * {@link #validate(Object, Errors) validate} instances of the
     * supplied {@code clazz}
     */
    boolean supports(Class<?> clazz);

    /**
     * Validate the supplied {@code target} object, which must be
     * of a {@link Class} for which the {@link #supports(Class)} method
     * typically has (or would) return {@code true}.
     * <p>The supplied {@link Errors errors} instance can be used to report
     * any resulting validation errors.
     * @param target the object that is to be validated
     * @param errors contextual state about the validation process
     * @see ValidationUtils
     */
    void validate(Object target, Errors errors);

}
```

In order for validators to be used, they have to be registered with a controller's `WebDataBinder`. This can be done with an init binder, a method executed before calls to specific endpoints, to setup the binding of incoming data. An init binder can be created by annotating a method with `@InitBinder` and then specifying for which so-called 'commands' it should be executed. Commands are either endpoint method names or endpoint parameter names. If no command is specified, the init binder will apply to all endpoints of a controller. Note that each endpoint method or parameter can only have one init binder attached to them.
Following code snippet registers a `CourseValidator`, implementing the `Validator` interface as validator for all endpoints:

```
@InitBinder
public void init(WebDataBinder binder) {
    binder.addValidators(new CourseValidator());
}
```

Note: The example code validates the DI principle for simplicity reasons.

Using bean validation

As an alternative to writing the validation logic manually, one can also use validation annotations, a.k.a. constraints, and have some framework write the validation logic for them (this is usually referred to as bean validation). Per JSR 380 a reference for such annotations was created. A popular reference implementation (and also the one used by the spring docs in its examples) is the Hibernate Validator (which has nothing to do with the persistence stuff from Hibernate). Take following class as an example of how to use bean validation:

```
public class Course {

    @Min(1337)
    private Long id;

    @Size(min = 3, max = 21)
    @Pattern(regexp = "[\\w\\s]*")
    private String subject;

    @Email
    private String professorEmail;
```

When using this approach, it is not necessary to setup anything validation-related in the init binder. Custom constraints can also be created by creating a new annotation, annotating it with `@Constraint` and then tying it to a validation class implementing the `ConstraintValidator` interface. In the background this approach actually utilizes the infrastructure of the first approach.

Marking data for validation

In order for incoming data, in the form of endpoint parameters, to be validated the `@Valid` annotation must be used to mark parameters for validation. Consider following example:

```
@PostMapping("")
public void createCourse(@RequestBody @Valid Course course) {
```

Formatters and Editors

Besides registering Validators, the init binder also allows for the registration of so-called formatters and editors. Formatters provide the ability to specify custom 'JSON string' <-> 'java type' conversions (Converters can be used for arbitrary 'java type' <-> 'java type' conversions) while editors allow for incoming data to be reformatted i.e. 'edited'

10.3.2 Exception Handling

Spring boot provides a mechanism for handling uncaught exceptions that might occur on the controller level. The goal is that, instead of simply returning an undescriptive `500 Internal Server Error`, specific exceptions can be handled differently and a more useful error message may be returned. To define exception handlers in Spring, exception handling methods (inside a controller) can be annotated with `@ExceptionHandler` where the `value` element can be used to specify which kind of exceptions it should handle. If a controller defines multiple exception handlers the most specific one of them will get called in case of an exception (same behaviour as a try-catch block). If a handler method is executed it's return value is what gets sent back to a client. Consider following example:

```
@ExceptionHandler(RuntimeException.class)
public String handler(RuntimeException ex) {
    return "Something done fucked up! " + ex.getMessage();
}
```

10.3.3 ControllerAdvice

When developing bigger applications, it is useful to be able to share certain controller configurations across multiple controllers. To do this the `@ControllerAdvice` annotation can be used. Classes annotated with it can define init binders and exception handler (and model attributes which are not relevant in this course) and have them applied to targeted controllers. By default, `@ControllerAdvice` configurations automatically apply to all controllers of an application but the set of targeted controllers can be narrowed using the `annotations()`, `basePackageClasses()`, and `basePackages()` (or its alias `value()`) selectors.
Consider following example which adds an init binder and an exception handler to all `@RestController`'s in the base package `stuffs`:

```
@ControllerAdvice(annotations = { RestController.class }, basePackages = "stuffs")
public class Advice {

    @ExceptionHandler(RuntimeException.class)
    public String handler(RuntimeException ex) {
        return "Something done fucked up! " + ex.getMessage();
    }

    @InitBinder
    public void init(WebDataBinder binder) {
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(false));
    }

}
```

10.4 Services

> When a significant process or transformation in the domain is not a natural responsibility of an entity or value object, add an operation to the model as standalone interface declared as a **service**. Define the interface in terms of the language of the model and make sure the operation name is part of the ubiquitous language. Make the **service** stateless.
>
> – Eric Evans in *Domain-Driven Design*

In Spring boot, services (are supposed to) contain an application's business logic. As described in Domain-Driven Design, Services are usually written as interfaces with independent, stateless implementations. For this Spring provides the `@Service` annotation. Interfaces can be annotated with it and Spring will look for the implementations using class path scanning and automatically register found implementations as components. For simpler applications, implementations may be annotated directly with `@Service`, in order to avoid creating an interface for it.

10.5 Communicating with a database

Most applications in the backend will want to read and write to a database in order to be useful. To do this, today's applications often use what is called an object-relational mapping tool (ORM tool). Such tools map OO-models onto a database without the programmer needing to write any database-specific code (i.e. no more SQL). This essentially means that the programmer can interact with data on a database (rows) as if it were composed of normal objects.

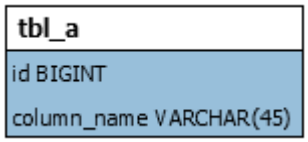
Figure 77 Meme

To do the aforementioned, we will use the Java Persistence API (JPA). It specifies a common interface for using ORM but doesn't actually implement any logic to perform ORM operations, so it is rather an SPI than an actual API. There exist several implementations for the JPA, also called JPA providers. Examples are EclipseLink (reference implementation), OpenJPA or Hibernate just to name a few. In this course we will be using Hibernate.

10.5.1 Configuring the Hibernate model mapping

Telling Hibernate how to map our entities can be done by annotating our model classes with the annotations exemplified in the following paragraphs:

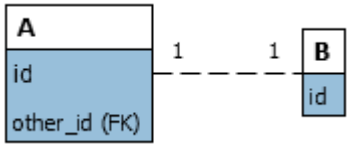
Simple table mapping:




```java
@Entity
@Table(name = "tbl_a")
public class A {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "column_name")
    private String someColumn;
```

`@Entity`: This class is an entity
`@Table`: The table where entities of this class are to be stored in
`@Id`: This field is the primary key of this entity
`@GeneratedValue`: The value for this field is generated by the DB
`@Column`: A simple column mapping. If the Java- and DB- field names are different, the column name of DB must be specified

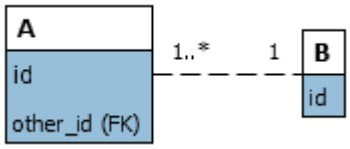
1 - 1 Relationship




```java
// inside class A
@OneToOne
@JoinColumn(name = "other_id")
private B another;
```

`@JoinColumn`: The foreign key column(s) referencing the other table
`@OneToOne`: This relationship has the cardinality [1-1]/[c-c] etc.

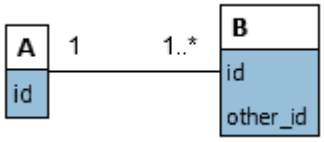
n - 1 Relationship




```java
// inside class A
@ManyToOne
@JoinColumn(name = "other_id")
private B another;
```

`@ManyToOne`: This relationship has the cardinality [n-1]/[nc-c] etc.

1 – n Relationship



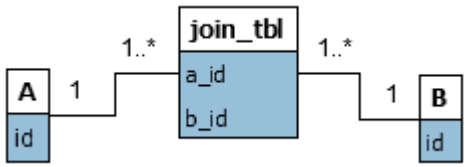

```java
// inside class A
// (B references A in field 'example')
@OneToMany(mappedBy = "example")
private Set<B> others;
```

`@OneToMany`: This relationship has the cardinality [1-n]/[c-nc]. If the relationship is bi-directional, the referencing field of the other class can be specified with `mappedBy`

```java
// inside class A
// (B doesn't reference A)
@OneToMany
@JoinColumn(name = "other_id")
private Set<B> others;
```

`@JoinColumn`: Must be specified if the relationship is unidirectional (the FK-column of the referencing (other) table is specified)

n - n Relationship




```
// inside class A
@ManyToMany
@JoinTable(
    name = "join_tbl",
    joinColumns = @JoinColumn(name = "a_id"),
    inverseJoinColumns = @JoinColumn(name = "b_id")
)
private Set<B> others;
```

`@ManyToMany`: this relationship has the cardinality [n-n]/[nc-nc] etc.

`@JoinTable`: Specifies the join table for the two entities with `name` being the name of that table, `joinColumns` being the columns referencing the entity wherein this annotation is specified and `inverseJoinColumns` being the columns referencing the other entity

10.5.2 The Spring Data JPA

To understand what the Spring Data JPA (SDJ) is all about, we will first take a look at how Hibernate is structured. Following is a high-level overview of Hibernate's architecture.

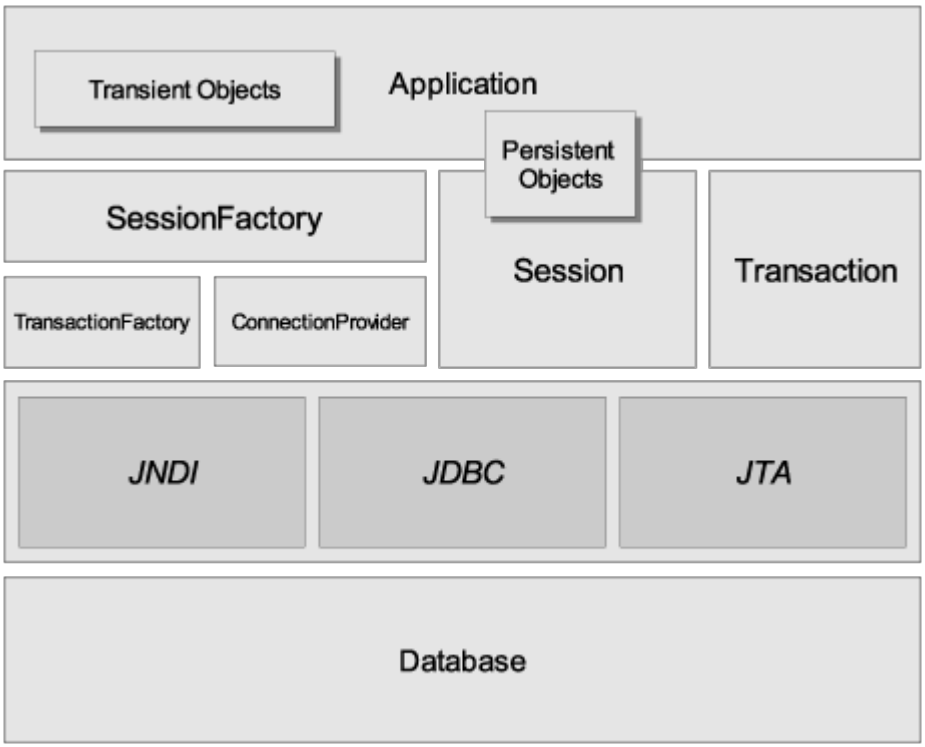


Figure 78 Hibernate Architecture (docs.jboss.org, DOA 05.02.19)

SessionFactory: A thread safe, heavyweight object used throughout the application to obtain `Session` objects for a particular database. It is usually created at application start-up and kept for later use. SessionFactories are created by a `Configuration` object.
Session: A lightweight object that wraps a JDBC connection and acts as a factory for Transactions. It offers create/retrieve functionalities and is supposed to be short-lived and is not thread safe. It can also execute `Query` objects.
Transaction: A lightweight object that represents a unit of work. All communication with the database (through a `Session`) is required to happen within the boundaries of a `Transaction` (it is possible to execute read operations without a transaction, though not recommended). If no `Transaction` is explicitly created, `autocommit` mode is assumed.

With these building blocks, a programmer can create interactions with the database. However, doing so requires an extensive amount of boilerplate code because the programmer needs to handle the creation of sessions and transactions manually and can't solely focus on implementing the required logic.

For this reason, the Spring Data JPA was created. It abstracts all the "low-level" Hibernate code away from the programmer so that he only has to implement the required logic. It even abstracts the usage of the methods provided by the `Session` class away.

Figure 79 SDJ Magic

In fact, the SDJ makes it possible to remove all DAO implementations entirely, making the interface of the DAO the only artefact that needs to be defined explicitly. In order to leverage this functionality with a JPA model, a DAO interface extending the `JpaRepository` marker interface must be created and annotated with `@Repository`, which will make it accessible as a bean/component. Note that Spring Data calls DAOs, Repositories, as originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behaviour which emulates a collection of objects".

By extending the `JpaRepository` interface we already get a wide range of CRUD methods out of the box. To add custom data access functionalities to our Repositories, we can simply add new method signatures to the Repository-interface and the SDJ will then generate the hibernate logic based on the methods' names, arguments and return types. For more complex queries it is also possible to specify custom HQL or JPQL queries (Hibernate Query Language; Java Persistence Query Language).

Following example demonstrates this idea. At runtime, Spring will automatically generate fully functional implementations for the methods defined in the interface and be able to provide the implementation through DI.

```
@Repository
interface UserRepository extends JpaRepository<User, Long> {

    // finds the User with the given name
    User findByUsername(String name);

    // checks if a user with the given ID exists
    boolean existsById(Long id);

    // finds the user with the given email and the given
    // streetname for his address
    @Query("from User u "
            + "join u.address as a "
            + "with u.email = ?1 and a.street = ?2")
    User findByComplexQuery(String email, String street);

    // deletes the User with the given name
    void deleteByUsername(String name);

}
```

Figure 80 SDJ Repository example

Derived query method structure

The names of SDJ derived query methods (methods where the query gets created from the name) are all structured by this basic pattern. ({…} = mandatory, […] = optional)

- `{operation}[additional keywords][By{criteria}][OrderBy{order}]`

Where the `operation` specifies what kind of operation needs to be performed, `additional keywords` specify additional things like `DISTINCT` or `TOP`, `By{criteria}` defines a `WHERE` clause with the given criteria and `OrderBy{order}` specifies the sort to be used. `All` can also be added as additional keyword (mostly) for aesthetical purposes.

Supported operation keywords

Following operation keywords can be used to create derived queries:

- `find, read, query, get, (stream)`
  a. Query the database for entities
  b. Additionally:
     i. `count` → get count of resulting entities
     ii. `exists` → check if there are results for a query
- `delete, remove`
  a. Delete matching entities

Specifying criteria

Criteria are specified after the `By` keyword. They consist of property names (of an entity) in conjunction with the following keywords. Multiple criteria can be combined with the keywords `And` and `Or`.

Keyword	Sample

`Is,Equals`	`findByFirstname (implicit 'Is'),`
	`findByFirstnameIs, findByFirstnameEquals`
	`… where x.firstname = ?1`

`Or`	`findByLastnameOrFirstname` `… where x.lastname = ?1 or x.firstname = ?2`
`And`	`findByLastnameAndFirstname` `… where x.lastname = ?1 and x.firstname = ?2`
`Between`	`findByStartDateBetween` `… where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan` `… where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual` `… where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan` `… where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual` `… where x.age >= ?1`
`After`	`findByStartDateAfter` `… where x.startDate > ?1`
`Before`	`findByStartDateBefore` `… where x.startDate < ?1`
`IsNull`	`findByAgeIsNull` `… where x.age is null`
`IsNotNull,NotNull`	`findByAge(Is)NotNull` `… where x.age not null`
`Like`	`findByFirstnameLike` `… where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike` `… where x.firstname not like ?1`

`StartingWith`	`findByFirstnameStartingWith` `… where x.firstname like ?1 (parameter bound with appended %)`
`EndingWith`	`findByFirstnameEndingWith` `… where x.firstname like ?1 (parameter bound with prepended %)`
`Containing`	`findByFirstnameContaining` `… where x.firstname like ?1 (parameter bound wrapped in %)`
`Not`	`findByLastnameNot` `… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)` `… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)` `… where x.age not in ?1`
`True`	`findByActiveTrue()` `… where x.active = true`
`False`	`findByActiveFalse()` `… where x.active = false`

`IgnoreCase`	`findByFirstnameIgnoreCase`
	`… where UPPER(x.firstame) = UPPER(?1)`
	
	Note: to apply `IgnoreCase` to all criteria, `AllIgnoreCase` can be added to the end of them. Example:
	`findByFirstnameAndLastameAllIgnoreCase`

Traversing nested properties

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Assume a `Person` has an `Address` with a `ZipCode`. In that case, the method creates the property traversal `x.address.zipCode`. Here, the resolution algorithm would actually start by checking if `AddressZipCode` is a known property. It would then split the name up to check if there is a property `AddressZip` with a nested property of `Code`. This goes on until a valid property has been found or the name can't be split anymore (in our case it would find `Address->ZipCode`). If there are any ambiguities, the '_' character can be used to manually define traversal points (see example).

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

Sorting results

To sort results `OrderBy{prop}(Asc)/(Desc)` can be appended to the end of a query method's name. (Note: the first `By` is required because of ugliness and doesn't have any meaning…)

```
// returns all users ordered by username
List<User> findAllByOrderByUsernameAsc();
```

Alternatively, a method can also define a special parameter of type Sort to take in a customized sort criterion.

```
// returns all users ordered by username
List<User> findAll(Sort sort);
```

Limiting query results

The results of query methods can be limited by using the `first` or `top` keywords, which can be used interchangeably. An optional numeric value can be appended to `top` or `first` to specify the maximum result size to be returned. If the number is left out, a result size of `1` is assumed. The following examples show how to limit the query size: (Note the ugliness `By` in the first two…)

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

List<User> findFirst42ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

Pagination

Pagination is the task of dividing query results into pages and retrieving the required pages, one by one on demand. With the SDJ this can be done by adding an additional parameter of type `org.springframework.data.domain.Pageable` to a query method. Consider following examples:

```
Page<User> findAll(Pageable pageable);

Slice<User> findAll(Pageable pageable);
```

A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive (depending on the store used), a `Slice` can be returned instead. A `Slice` only knows about whether a next `Slice` is available or not, which might be sufficient when walking through a larger result set.

Streaming and async

Results from query methods can also be streamed by using a java 8 `Stream` as result type. This may have data store- and/or JPA vendor-specific advantages over returning the usual `List`. An example of such advantages would be using Streams in conjunction with Hibernates `ScrollableResult`, which is way more efficient for processing big result sets.
Repository queries can also be run asynchronously by using java's `Future`/`CompletableFuture` or Springs `ListenableFuture` as return type and by annotating the query method with `@Async`. This means that the query method returns immediately upon invocation while the actual query execution occurs in a task, submitted to a Spring `TaskExecutor`.

Named Queries - Custom query definitions

Although getting a query derived from the method name is quite convenient, one might face the situation in which either the method name parser does not support the keyword one wants to use or the method name would get unnecessarily ugly (formulation taken from the official documentation). Due to this reason, queries can also be manually declared using the `@Query` annotation with a query method. To specify the query either JPQL (Java Persistence Query Language) or some JPA vendor specific language can be used (HQL for Hibernate). Following example uses HQL to declare a custom query:

```
// finds the user with the given email and the given
// streetname for his address
@Query("from User u "
        + "join u.address as a "
        + "with u.email = ?1 and a.street = ?2")
User findByComplexQuery(String email, String street);
```

Modifying queries

The SDJ can also be used to create `UPDATE` and `DELETE` queries (`INSERT` queries are already provided by `save` and `saveAll`). However, only `DELETE` queries can be derived. UPDATE queries must be declared manually using the `@Query` annotation. Also, if modifying queries are declared on a query method with `@Query`, that method must also be annotated with `@Modifying`. Following code exemplifies this:

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);

void deleteByRoleId(long roleId);

@Modifying
@Query("delete from User u where user.role.id = ?1")
void deleteInBulkByRoleId(long roleId);
```

Although the `deleteByRoleId(…)` method looks like it basically produces the same result as the `deleteInBulkByRoleId(…)`, there is an important difference between the two method declarations in terms of the way they get executed. As the name suggests, the latter method issues a single JPQL query (the one defined in the annotation) against the database. This means that no JPA lifecycle stuff gets invoked for the entities to be deleted (e.g. `@PreDestroy`). `deleteByRoleId(…)` on the other hand, first executes a query and then deletes the returned instances one by one, so that the persistence provider can actually invoke it's lifecycle stuff.

In fact, a derived delete query is a shortcut for executing the query and then calling `CrudRepository.delete(Iterable<User> users)` on the result and keeping behaviour in sync with the implementations of other `delete(…)` methods in `CrudRepository`.

Appendix - Additional information

The SDJ actually provides a lot more features than what already has been presented here. Unfortunately listing them all would go beyond the scope of this project. If you want to learn more about the SDJ right here (reference doc) is a good place to start. Some topics worthy of reading about are custom implementations with repository fragments, null handling, JPA named queries and stored procedures with `@Procedure`.

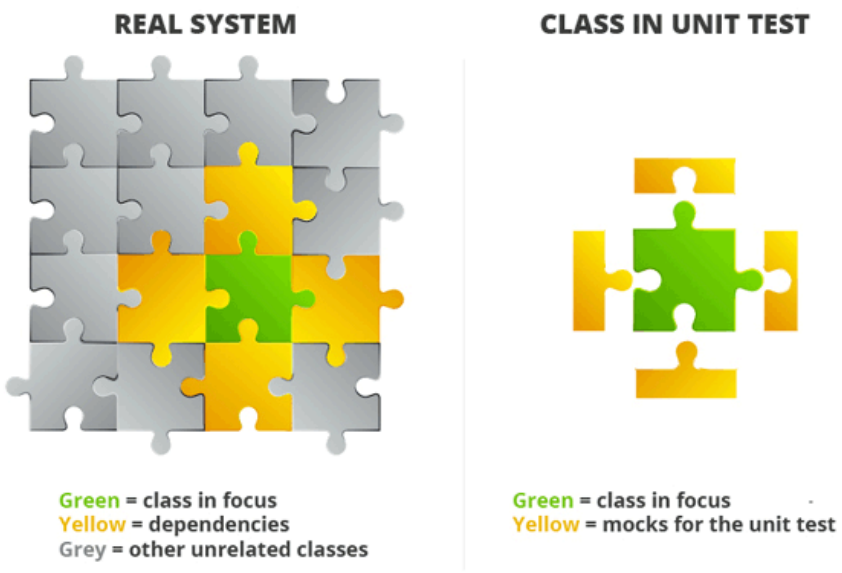
10.6 Security

10.7 Testing

Difference between unit tests and integration tests

10.7.1 Mocking with Mockito

Mocking is used to effectively run unit tests. To isolate the behaviour of an object (which is required for unit testing), all of its dependencies are replaced by mocks that simulate the behaviour of the real ones. Mock objects return dummy data corresponding to some dummy input passed to them.



In unit testing we want to test methods of one class in isolation. But classes are not isolated. They are using services and methods from other classes. So in that situation, we mock the services and methods from other classes and simulate the real behavior of them using some mocking frameworks and use that mocked methods and services to do unit testing in isolation. This is where Mocking frameworks come into play. [18]

Mockito is such a framework. It facilitates creating mock objects seamlessly and supports many features like runtime mock creation, return value- and exception support, method call monitoring and the creation of mocks using annotations.

Lucy Shea Danielson
Albert Hoffman
Dimitri Dimitrovic

[18] https://medium.com/@piraveenaparalogarajah/what-is-mocking-in-testing-d4b0f2dbe20a

10.7.2 Unit tests in spring boot

Controller Layer

Service Layer

Repository Layer

10.7.3 Intergration tests in spring boot

10.8 Running the Project

Now that we've got everything set up, we can actually run our project and test it. To do so we will use the gradle task called `bootRun` which will build and run our Spring Boot application. You should now see something similar to the following in the console where you ran the task.

```
2019-01-23 16:03:12.384  INFO 2584 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-01-23 16:03:12.388  INFO 2584 --- [           main] com.example.demo.MyApplication           : Started MyApplication in 2.085 seconds (JVM running for 2.507)
```

Figure 81 Launch console output

To test if the defined endpoints have been configured correctly we will use a program called 'Postman' (available at https://www.getpostman.com/downloads/). It enables us to execute HTTP requests in an easy way. Inside Postman create two `GET` requests with the paths `localhost:8080/courses` and `localhost:8080/courses/:id` where, for the second request, an ID must be specified in the Params-tab. Create a new collection and store the created requests therein. Collections are a collection of requests which can share tests, variables, scripts and authentication and are a central element to Postman. When developing a new API, it is generally good practice to create a new request for each new endpoint and to use a shared workspace when working in a team. The requests can now be executed by pressing 'send'.

Overview of Postman's interface (yes, this address really exists!):

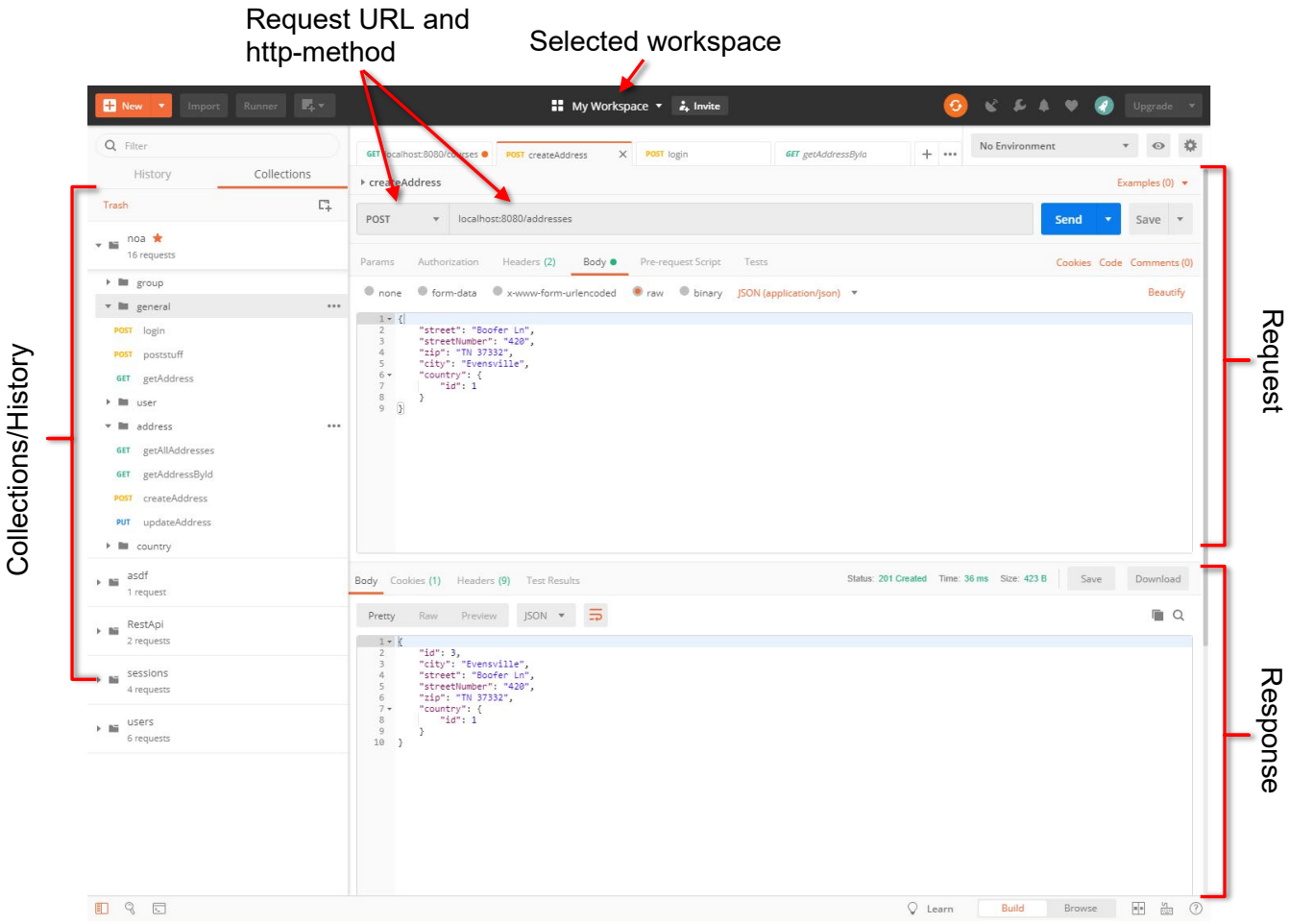


Figure 82 Postman GUI

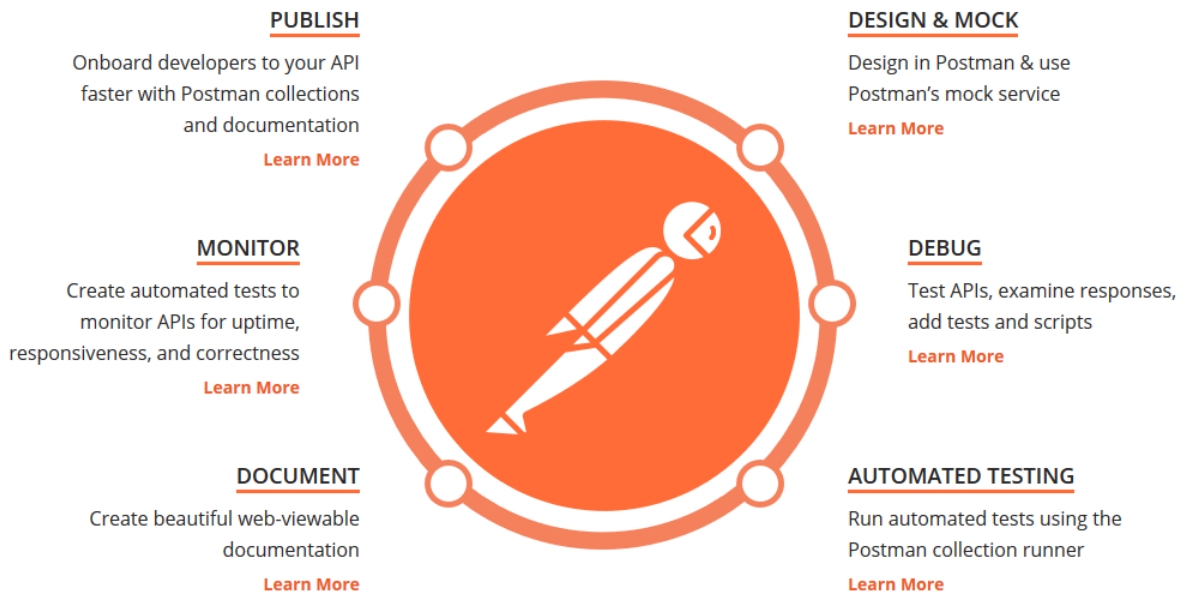


Figure 83 Postman Overview

Postman can actually do much more than just simply executing HTTP-Requests (we have curl for that). It enables developers to share their requests inside their teams using shared workspaces, to use of pretty powerful variables, to create automated tests and environment scripts, to automatically generate documentations and to monitor APIs in production with automated tests. If you want to learn more about postman the postman learning center located at https://learning.getpostman.com/ is a great place to start.

// TODO chapter more about DB: connection pooling, transactions

// TODO show SDJ controller integration [described in SDJ reference doc #4.8.2]

// TODO [CHECK] DTO <-> Model stuff → look in architecture chapter

// TODO unit testing (mocking), Hamcrest <-> AssertJ, (Integration testing) [maybe v2]

// TODO Authentication <->Authorization, JWT (Scheduled for security chapter)

11 List of tables and figures

12 List of references

https://code.snipcademy.com/tutorials/git/introduction/how-version-control-works
https://git-scm.com/book/en/v2/
https://olat.zhaw.ch PROG2 IT17ta_ZH
https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified#_git_reset
https://en.wikipedia.org/wiki/List_of_build_automation_software
https://www.baeldung.com/gradle
https://www.bignerdranch.com/blog/introduction-to-gradle/
https://semaphoreci.com/community/tutorials/introduction-to-gradle
https://en.wikipedia.org/wiki/Apache_Groovy
https://www.scaledagileframework.com/domain-modeling/
https://searchsoftwarequality.techtarget.com/definition/use-case
https://sourcemaking.com/uml/modeling-it-systems/external-view/the-elements-of-view/use-case-diagram
http://agilemodeling.com/artifacts/classDiagram.htm
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/
https://www.uml-diagrams.org/sequence-diagrams.html
https://www.visualparadigm.com/support/documents/vpuserguide/3563/3564/85378_conceptual.html
https://www.1keydata.com/datawarehousing/conceptual-data-model.html
https://www.1keydata.com/datawarehousing/logical-data-model.html
https://www.1keydata.com/datawarehousing/physical-data-model.html
https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/
https://searchsoftwarequality.techtarget.com/definition/use-case
https://sourcemaking.com/uml/modeling-it-systems/external-view/the-elements-of-view/use-case-diagram
https://herbertograca.com/2017/07/28/architectural-styles-vs-architectural-patterns-vs-design-patterns/
https://www.linkedin.com/pulse/architectural-vs-design-patterns-software-engineering-asher-toqeer
https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/
https://searchsoftwarequality.techtarget.com/definition/3-tier-application
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/
https://en.wikipedia.org/wiki/Data_transfer_object
https://en.wikipedia.org/wiki/Representational_state_transfer
https://www.uml-diagrams.org/realization.html
http://www.cs.utsa.edu/~cs3443/uml/uml.html
https://www.webopedia.com/TERM/N/normalization.html
https://www.guru99.com/database-normalization.html
https://www.lifewire.com/database-normalization-basics-1019735
https://www.vividcortex.com/blog/what-is-cardinality-in-a-database
https://www.techopedia.com/definition/18/cardinality-databases
https://www.scribd.com/doc/53520296/Class-Diagram-Exercises-and-Solutions
https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms
https://docs.spring.io/spring-framework/docs/current/javadoc-api/

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
https://docs.spring.io/spring/docs/current/spring-framework-reference/
https://stackoverflow.com/questions/39890849/what-exactly-is-field-injection-and-how-to-avoid-it
https://softwareengineering.stackexchange.com/questions/300706/dependency-injection-field-injection-vs-constructor-injection

http://olivergierke.de/2013/11/why-field-injection-is-evil/
https://learning.getpostman.com/docs/postman/
https://searchsoftwarequality.techtarget.com/definition/requirements-analysis
https://www.evoketechnologies.com/blog/functional-non-functional-requirements-sdlc/
https://www.mountaingoatsoftware.com/blog/agile-needs-to-be-both-iterative-and-incremental
https://blog.easyagile.com/the-difference-between-a-flat-product-backlog-and-a-user-story-map-93cff02f23a3
https://reqtest.com/requirements-blog/understanding-the-difference-between-functional-and-non-functional-requirements/