Software Engineering 2

github.com/martinopiaggi/polimi-notes

2022-2023

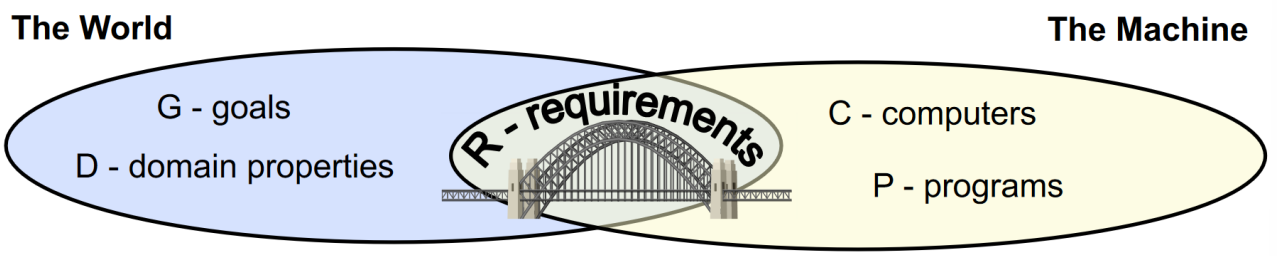
Contents

1 Requirements Engineering **3**
- 1.1 Shared and not shared phenomenas . . . 3
- 1.2 Define use case flow . . . 4

2 Alloy **5**
- 2.1 Syntax . . . 5
- 2.2 Predicates . . . 6
  - 2.2.1 Predicates to model actions . . . 7
  - 2.2.2 Overloading . . . 7
- 2.3 Functions . . . 8
- 2.4 Facts . . . 8
- 2.5 Assertions . . . 8
- 2.6 Super basic commands . . . 8
- 2.7 Useful links . . . 9
- 2.8 Brief List of Design principles and process . . . 9
  - 2.8.1 Design principles . . . 9
  - 2.8.2 Design process . . . 9
- 2.9 Design Document Structure . . . 9
- 2.10 Architectural styles . . . 10
  - 2.10.1 Layered style . . . 11
  - 2.10.2 Client/Server . . . 12
  - 2.10.3 Three-tier architecture . . . 13
  - 2.10.4 Event-based . . . 14
  - 2.10.5 Service-Oriented Architecture (SOA) . . . 15
  - 2.10.6 Microservice architectural style . . . 15
- 2.11 UML diagrams . . . 17
  - 2.11.1 Use Case Diagram . . . 17
  - 2.11.2 Component Diagram . . . 19
  - 2.11.3 Class Diagram . . . 26
  - 2.11.4 Sequence Diagram . . . 28
  - 2.11.5 Deployment Diagram . . . 30

3 Availability **31**
- 3.1 Solve availability in practice . . . 32
- 3.2 Client tier . . . 32
- 3.3 Web tier . . . 33
  - 3.3.1 Stateful and stateless beans . . . 34
  - 3.3.2 Examples of beans methods . . . 34
  - 3.3.3 JMS APIs . . . 35
  - 3.3.4 JPA . . . 35
- 3.4 Testing . . . 36
- 3.5 Analysis . . . 37
  - 3.5.1 <Def-use> analysis . . . 38
  - 3.5.2 Symbolic Execution . . . 38
- 3.6 Schedule planning and Gantt Chart . . . 39
  - 3.6.1 Gantt chart . . . 41
- 3.7 Function points . . . 41
- 3.8 EVA monitoring methodology . . . 44

1 Requirements Engineering



RE is a relatively young domain, there's no consensus on terminology yet, in particular about what is a requirement. The purpose of a RE activity is to:

- to identify the real goals of the project
- to explore alternative ways to satisfy the goals, through alternative pairs (Requirements, Domain assumptions) such that Requirements and Domain assumptions always satisfy G
- to evaluate the strengths and risks of each alternative interfaces between the world and the machine

Some definitions:

- **The machine** is the portion of system to be developed typically, software and hardware.
- **The world** is the portion of the real-world affected by the machine.
- **Phenomena** can be **shared** between world and machine. Shared phenomena can be controlled by the machine and observed by the world, or viceversa.
- **D** = Domain assumptions are descriptive assertions assumed to hold in the world. They are real world properties and they don't depend on the machine.
- **G** = Goals are assertions formulated in terms of world phenomena.
- **R** = Requirements are assertions formulated in terms of shared phenomena.

The requirements are complete if :

1) **R** ensures satisfaction of the goals **G** in the context of the domain properties

2) **G** adequately capture all the stakeholders's needs

3) **D** represents valid properties/assumptions about the world

When the domain assumptions **D** are wrong the software design could lead to disasters. Note that we are not talking about bugs on the code: problems could caused purely because bad design of the software.

Example G-A-R:

- Goal: The spectators want to see who is leading in all parts of the path.

- Assumption: The error of devices in measuring the position of athletes is lower than 1 mt.
- Requirement: The system must display the position of the leading athlete on the map.

1.1 Shared and not shared phenomenas

	Controlled by WORLD	Controlled by MACHINE
SHARED	action initiated by the world that may trigger a reaction from the machine.	action initiated by the machine which is seen from the world

	Controlled by WORLD	Controlled by MACHINE
NOT SHARED	action initiated by the world with not directly observed by the machine	"black-box action" (invisible to the world) required by the machine to function properly

Example:

Phenomenon	Shared	Who controls it
User wants to buy some milk	N	W
User inserts a coin in the machine	Y	W
The machine compares the inserted coin with the last received one	N	M
The machine rejects the inserted coin	Y	M
The machine accepts the inserted coin	Y	M
User inserts a fidelity card	Y	W
The machine checks and accepts the fidelity card	Y	M
The machine sees that amount needed to buy a bottle of milk is reached	N	M
The machine delivers the bottle of milk		
The machine updates the current amount of money	Y	M
The user goes home with the milk	Y	M
The user wants to receive the money back	N	W
The user asks for the money back	N	W
The machine delivers the amount of money to the user	Y	W
The machine resets the money count	Y	M
The operator sets the current number of bottles in the machine	N	M
A milk sensor signals the milk in the machine is finishing	Y	W
The machine decreases the counter of the current number of bottles	N	M
The machine goes out of service	Y	M

1.2 Define use case flow

1) (Trivial) Give a name to the use case: the use case is the flow of events in the system
2) Find the actors

3) Then concentrate on the flow of events using informal natural language:
   - entry conditions
   - exit conditions: use case terminates when the following condition holds
4) Exceptions (Describe what happens if things go wrong)
5) Special requirements (Nonfunctional Requirements, Constraints)

Some notes:

- Note that each use case may lead to one or more requirements.
- We should separate as much as possible different activities. It's better to keep concerns separated. For example a generic use case of an app will probably need the user logged in before posting a useless meme. But in the activity of posting the meme, we will not specify **in the flow of events** the action of 'logging in' , but only in the entry conditions.
- Requirements identification needs to take into account the needs of many stakeholders: identify priorities of requirements is the task of a requirement engineer.
- Implementations concerns are not related to a Requirement Engineer. The RE only focus on 'the world' not the machine.

- the requirement validation is not 'a monolitich static step'. You continue to search error: the earlier you catch the errors, the earlier you can fix them.

2 Alloy

Alloy is a tool for specifying models of systems and softwares with the goal of seeing if they are designed correctly. It is basically a **declarative** OO language with a strong mathematical foundation.

2.1 Syntax

- `sig`: Signatures define types and relationships. Alloy generates instances of each signature, called **atoms**. Since we care about the relationships between the parts of our systems we add **relations** inside of the signature body. Note that the field of each signature (the relations) are first class objects, and can be manipulated and used in expressions.

```
sig sig_name{
    field: one Other_sig,
    field: set Another_sig
}
```

- `extends`: keyword we can make some signatures subtypes of other signatures.
- `abstract` : there won't be atoms that are just the supertype but not any of the subtypes.
- We can make enums in this way: `sig weather{Cloudy, Sunny, Rainy}`.
- `enum` using subtypes:

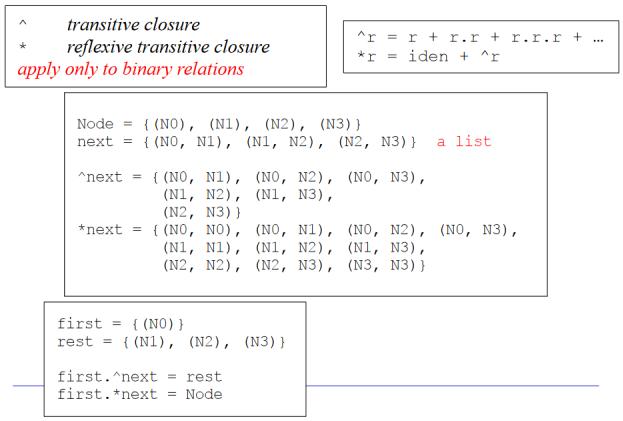
```
abstract sig Color {}
one sig GREEN extends Color {}
one sig RED extends Color {}
```

- Expression quantifiers are also used in expressions and not only in relations:
  - some var: at least one
  - all var
  - one var: exactly one
  - lone var : at most one
  - no var
- `disj` is used to specify distinct instances. It can be append to any expression quantifier. For example: `all disj a,b : S | expres`.
- `none` is the empty set
- `univ` is the universal set
- `iden` is the identity relation
- `Int` is the set of integers that have been instantiated.
- `int` returns the value of an `Int`. You have to explicitly write `int i` to be able to add, subtract, and compare an `Int`.
- `~r`(transpose/inverse)

```
assert symmetric{
friends = ~friends}
```

- `^r` (positive transitive closure)

```
//restutuisci gli ancestori di p
fun ancestors [p: Person]: set Person {
p.^(mother+father)
}
```



- `*r` (reflexive transitive closure)
- `x in A` means that x is an element of the set A
- Dot join: `a.b` combines two or more sets into a single set. Remember that the match column is dropped: ‘(a) in A and (a,b) in F implies A.F=(b)‘ . Also remember a join on the right side returns the set of left side elements, and viceversa. The parenthesis are not always necessary, but in doubt put them in both cases.
- Box join: `a[b]` is equivalent to writing `a.(b)`. It has a lower precedence than the `.` operator.
- Cross product: a $\rightarrow$ b (Cartesian product)
- Intersection: `a & b` (intersection of two sets, not equivalent with `and`)
- Union, difference: `a+b`, `a-b` (between sets)
- Comparison operators: in `=,<,>,=,=<,=>`
- Set comprehensions:

```
{x: Set1 | expr[x]}
{x: Set1, y: Set2, ... | expr[x,y]}
```

- expressions are mix of the last points:

```
expr1 or {
  expr2
  expr3
    ...
}
-- means expr1 or (expr 2 and expr3 and ...)
```

- `let`: often used to simplify expressions.

```
//let example:

// this function returns the set of devices which host a specificsoftware module in a given system config
fun deployedOn [c: DeployedSystem, sm: SoftwareModule] : set Device{
    let res = {d: Device | d in c.devices and sm in d.deployed} |
    res
}
```

- Logical operators (both symbolic and english form) |word|symbol| |:–:|:—:| |`and`|`&&`| |`or`|`||`| |`not`|`!`| |`implies`|`=>`| |`iff`|`<=>`|

2.2 Predicates

A predicate is like a programming function that returns a boolean. Predicates are mainly use to model constraints since they are **reusable expressions**. The most used is `show()` , which is a **built-in Alloy Analyzer pred** to

show us a visual rapresentation of the model specified.

2.2.1 Predicates to model actions

```
pred draw[c, c': Coordinator, num: Number]{
//precondition
not num in c. drawnNumbers
//postcondition
c'.tickets = c.tickets
c'.drawnNumbers = c.drawnNumbers+num
}
```

Note that preconditions and postconditions are just comments, they are not in Alloy syntax. Write pre and post conditions when evaluating the effect of applying an "action" to the objects you are dealing with. Check that given certain initial conditions, after performing a certain operation, certain other properties are true.

```
//adding new post to the social network
pred addPost [ s, s' : SocialNetwork, p : Post ] {
    //precond
    not p in s.posts
    p.creator in s.users
    p.comments = none
    //postcond
    s'.users = s.users
    s'.comments = s.comments
    s'.posts = s.posts + p
}
```

In the predicate's signature, you should use the following syntax: `pred examplePred[a1,a2: A , b1,b2: B, c: C]` where `a2` and `b2` will represent respectively `a1` and `b1` changed. Typical thing I forgot: *when adding an instance to a set, remember to check that it is not already there!*

```
pred deploy[c1,c2: Config, m1: Module, d1,d2: Device]{
    //precond
    m1.tech in d1.supported
    d1 in c1.devices
    not (m1 in d1.modules)

    //post
    d2.location = d1.location
    d2.supported = d1.supported
    d2.modules = d1.modules + m1
    c2.devices = c1.devices - d1 + d2
    c2.modules = c1.modules
}
```

2.2.2 Overloading

```
sig A {}
sig B {}

pred foo[a: A]{
  a in A
}
```

```
pred foo[b: B]{
  b in B
}
```

2.3 Functions

Alloy functions have the same structure as predicates but also return a value. Unlike functions in programming languages, they are always executed within an execution run, so their results are available in the visualisation and the evaluator even if you haven't called them directly. This is very useful for some visualisations, although the supporting code can be disorienting when transitioning from "regular" programming languages.

```
fun name[a: Set1, b: Set2]: output_type {
  expression
}
```

2.4 Facts

A fact is a property of the model. It's a predicate which is always considered true by the Analyzer. Any models that would violate the fact are discarded instead of checked.

```
fact {
    all m: Man, w: Woman | m.wife = w iff w.husband = m
}

//"Nel join" tra tutte le donne e gli uomini (con la relazione wife) non ci deve essere nessun antenato c

fact noCommonAncestorsInMarriage{
    all p1: Man, p2: Woman |
    p1 -> p2 in wife implies ancestors[p1] & ancestors[p2]=none
    }
```

We can also write constraints after the signature declaration, in this way:

```
//this constraint avoid "self requirement"

sig Function {
    reqs: set Function
} {not this in reqs }
```

2.5 Assertions

Assertions are properties we want to check.

```
assert NoSelfFather {no m: Man | m = m.father}
check NoSelfFather
```

2.6 Super basic commands

A *command* is what actually runs the analyzer. It can either find models that satisfy your specification, or counterexamples to given properties.

- run: finds a matching example of the specifications. You can use it with a bound: `run {} for 42 but exactly 4 Bananas, 2 Pears` .
- run {fact} : Finds an example satisfying the ad-hoc constraint in the braces.

- check: **check** tells the Analyzer to find a counterexample to a given constraint.

2.7 Useful links

https://alloy.readthedocs.io/en/latest/ https://alloytools.org/download/alloy-language-reference.pdf # Software Design

2.8 Brief List of Design principles and process

2.8.1 Design principles

- Keep the level of abstraction as high as possible: organizing the design into high-level abstract concepts.
- Divide and conquer: breaking a complex problem into smaller, more manageable parts.
- Increase cohesion where possible: making sure that the different parts of a system work together effectively and efficiently.
- Reduce coupling where possible: minimizing the dependencies between different parts of a system to increase flexibility and ease of modification.
- Design for flexibility: allowing for changes to be made to the system without requiring significant modification or redesign.
- Design for reusability/portability: designing components that can be used and moved to different environments or platforms.
- Reuse existing designs and code: using existing solutions to common problems rather than re-inventing the wheel.
- Anticipate obsolescence: designing with the expectation that technologies and components will become outdated over time.
- Design for testability: making it easy to test and verify the correct operation of the system.

2.8.2 Design process

- Top-down approach: starting with the high-level structure of the system and gradually working down to low-level details.
- Bottom-up approach: starting with reusable low-level utilities and building up to high-level constructs.

2.9 Design Document Structure

1) Introduction
   - Scope: reviews the domain and product, summary of main architectural choices
   - Definitions, acronyms, abbreviations
   - Reference documents
   - Overview
2) Architectural Design
   - Overview: high-level components and interactions
   - Component view: components and interfaces description
   - Deployment view: infrastructure description
   - Component interfaces: more detailed description of each interface
   - Runtime view: dynamics of interactions

- Selected architectural styles and patterns
- Other design decisions

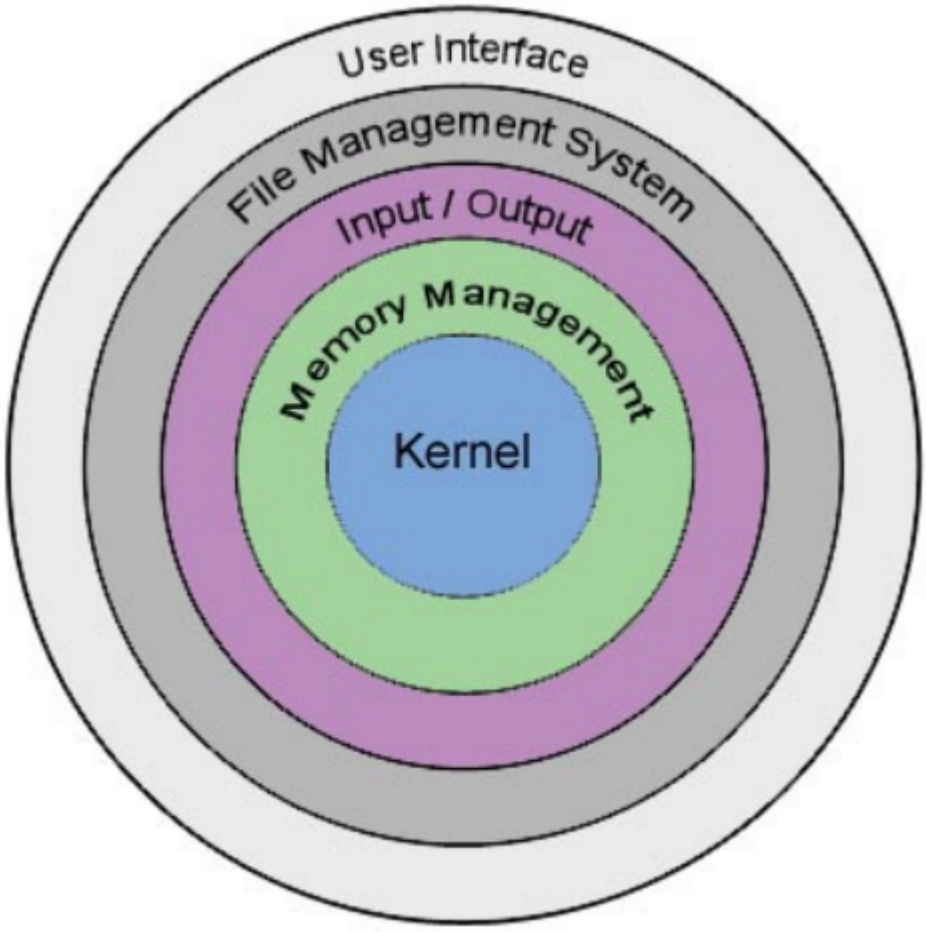
3) User Interface Design: overview of UIs
4) Requirements traceability
5) Implementation, Integration and test Plan: order in which you plan to implement all the stuff
6) Effort Spent
7) References

2.10 Architectural styles

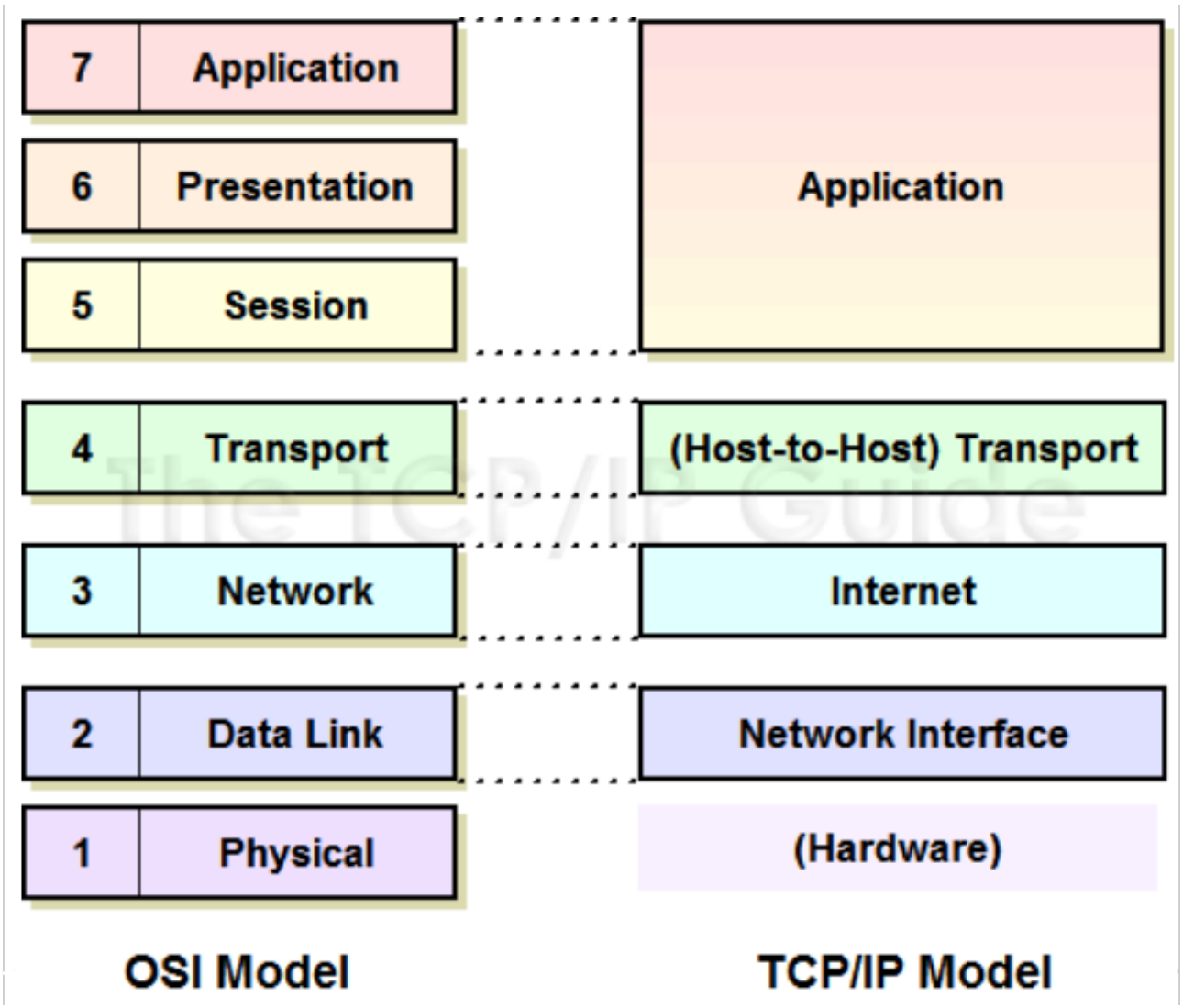
Architecture is a transferable abstraction of a system. The architecture manifests the earliest set of design decisions:

- it introduces design constraints on implementation
- it introduces organizational structure

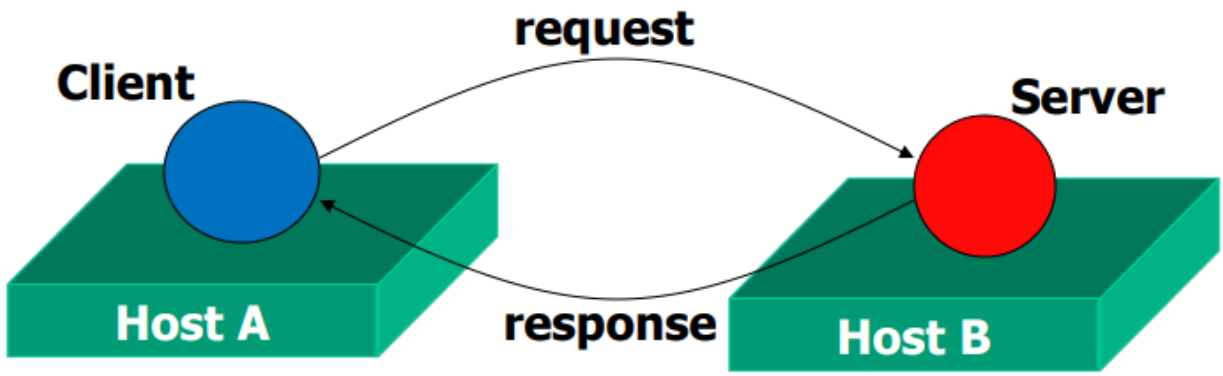
2.10.1 Layered style



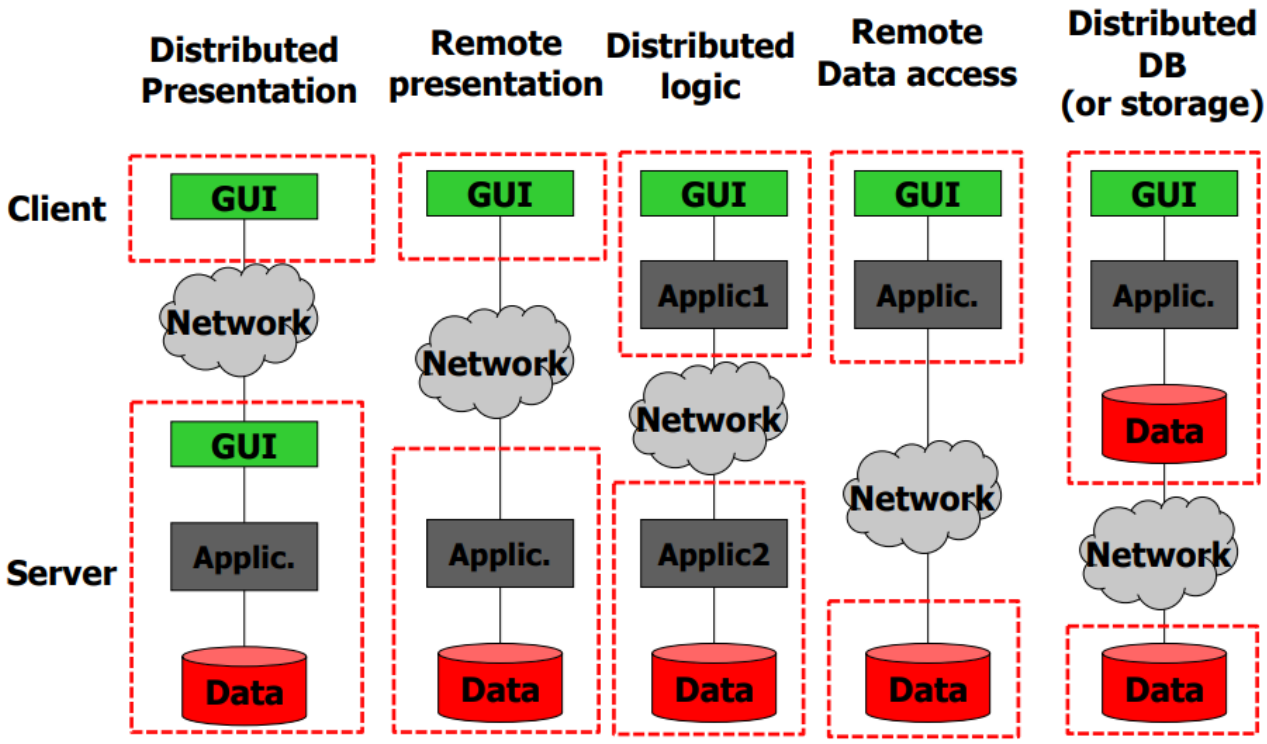
Layers architecture enable separation of concerns, claryfying the main focuses of each layer. Each layer can be considered 'independent' from the others. Layered style can be used when it is possible to identify a specific (bounded) concern for each layer and clearly communication protocol between layers.



2.10.2 Client/Server

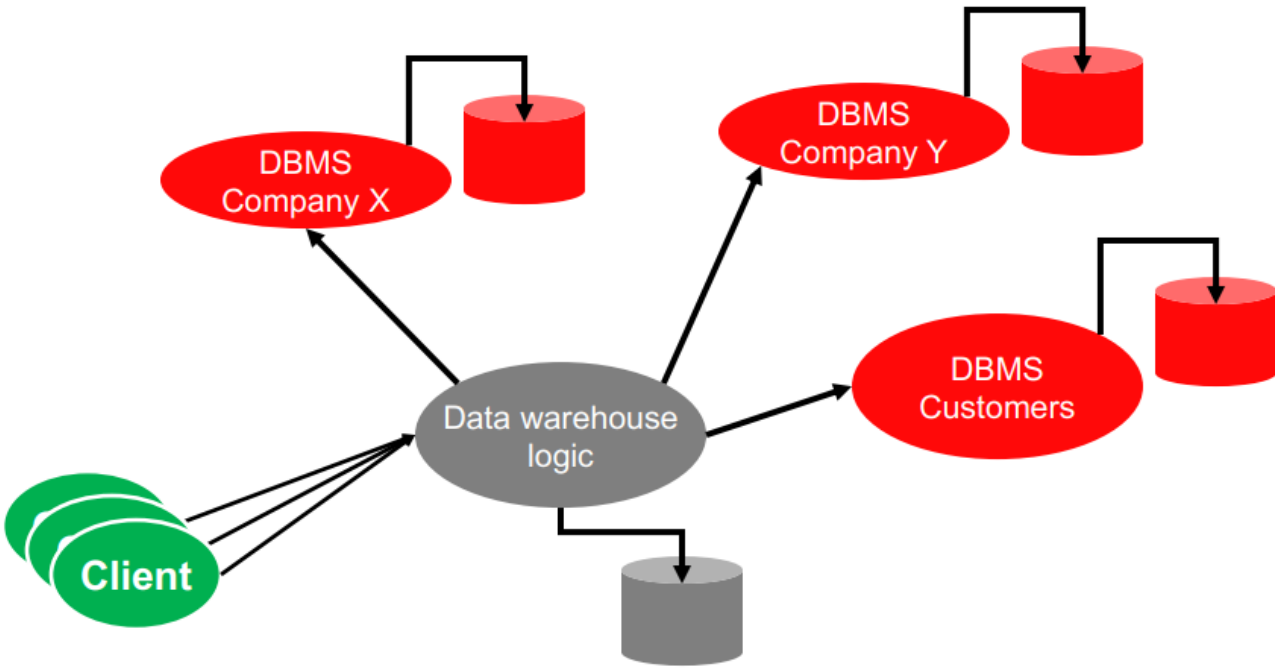


A client-server architecture is a kind of layered architecture with two layers (also called tiers).



2.10.3 Three-tier architecture

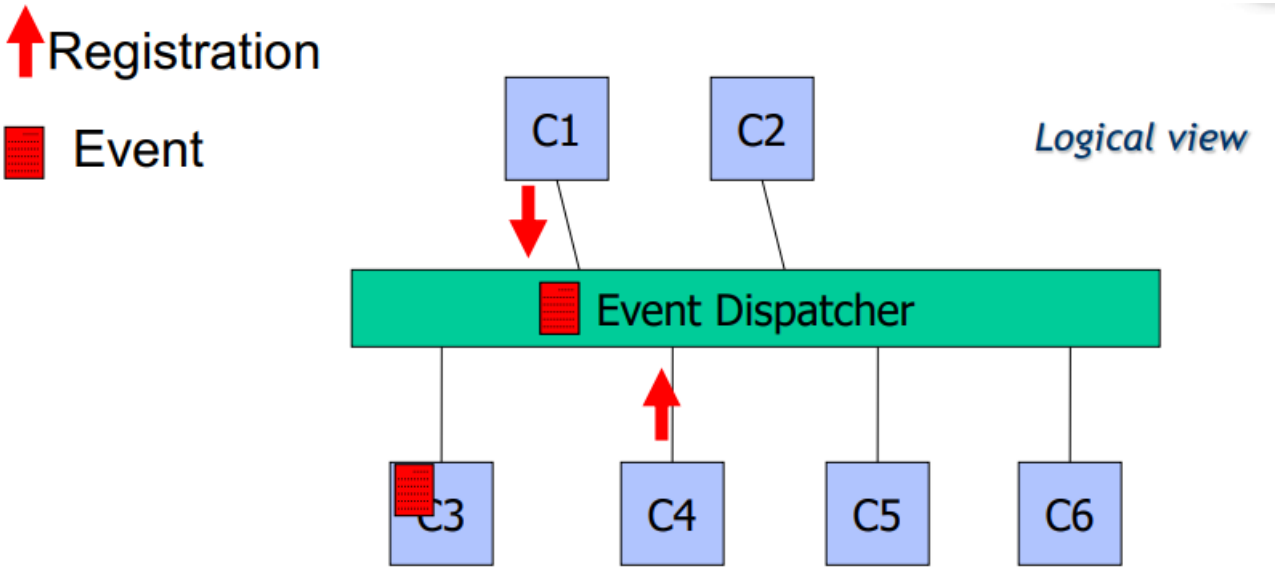
Decoupling of logic and data, logic and presentation using three tiers.



Note how the middle tier plays 2 roles: both server and client.

2.10.4 Event-based

This kind of style is popular in distributed systems (systems heavily distributed and decentralized).

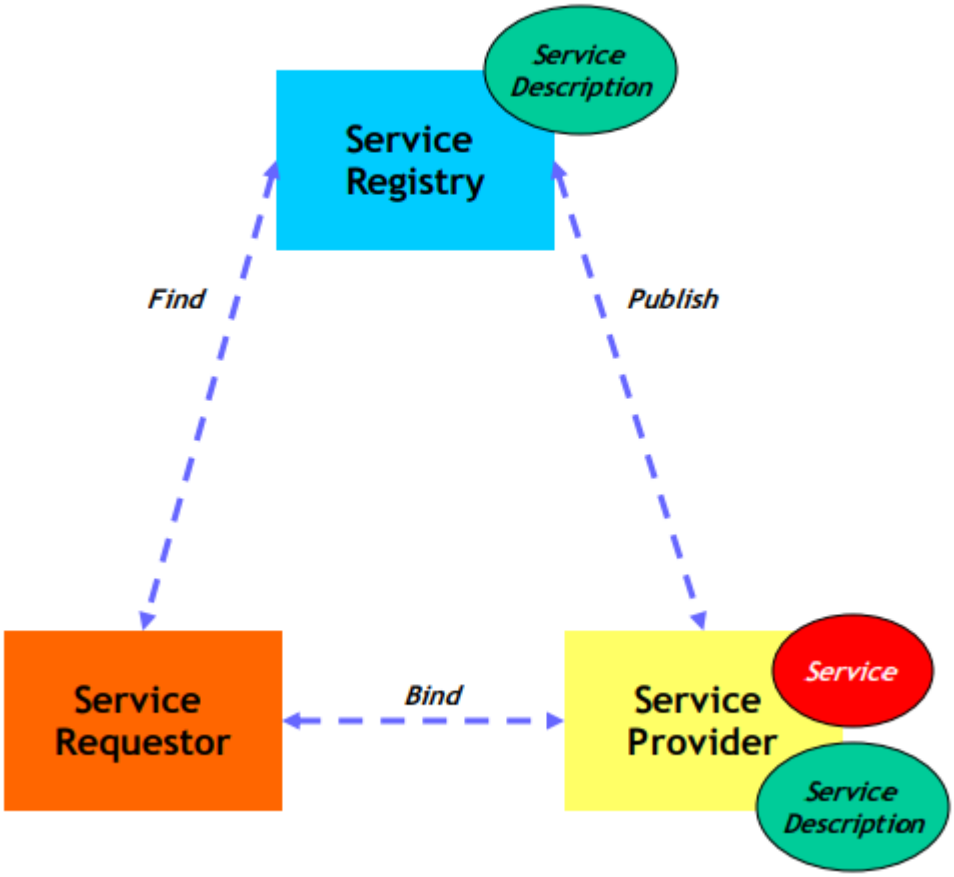


There are producers of events and consumers of evens and everything is coordinated by an event dispatcher:

- components can “register to” or “send events”
- events are broadcast to all registered components by the event dispatcher

Since publishers/subscribers are decoupled, addition/deletion of components is easy. The main problem is the scalability since the event dispatcher may become a bottleneck (under high workloads). Also notice that the architecture has an asynchronous nature so the ordering of events is not guaranteed. In continuous integration and deployment (CI/CD), event-based architecture is very common: for example, in GitHub Actions, a user can set up a workflow that is triggered by a specific event, such as a new commit being pushed to the repository. This and other triggers can make the CI/CD process more efficient and automated.

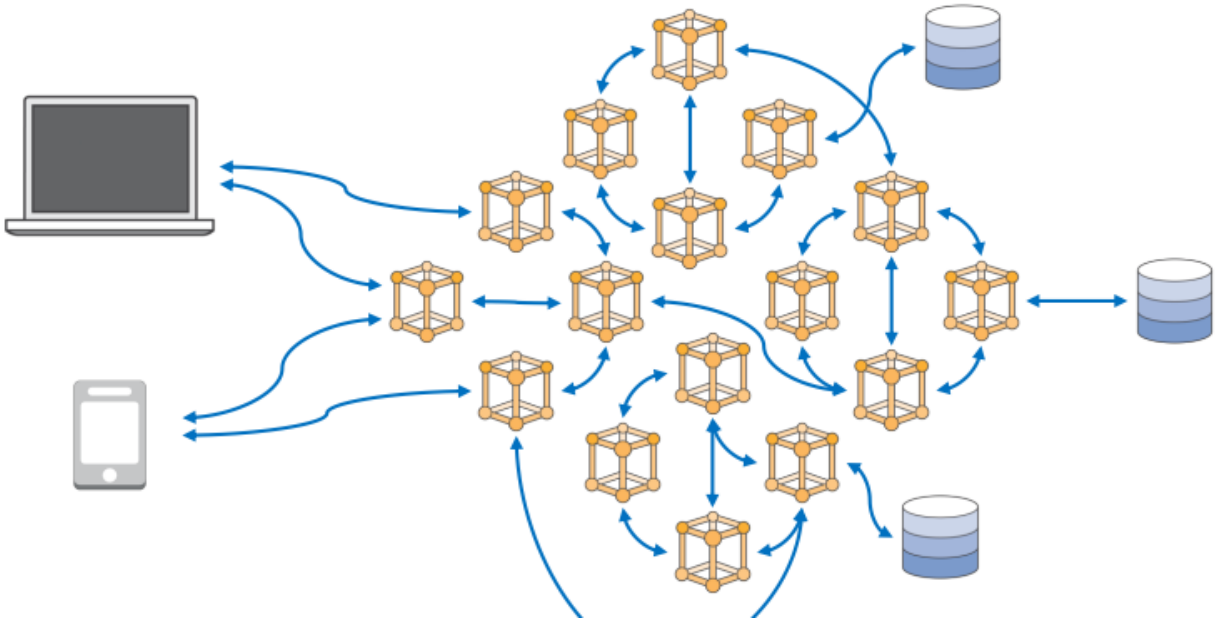
2.10.5 Service-Oriented Architecture (SOA)



The architecture provides many services and each of them are offered through a API.

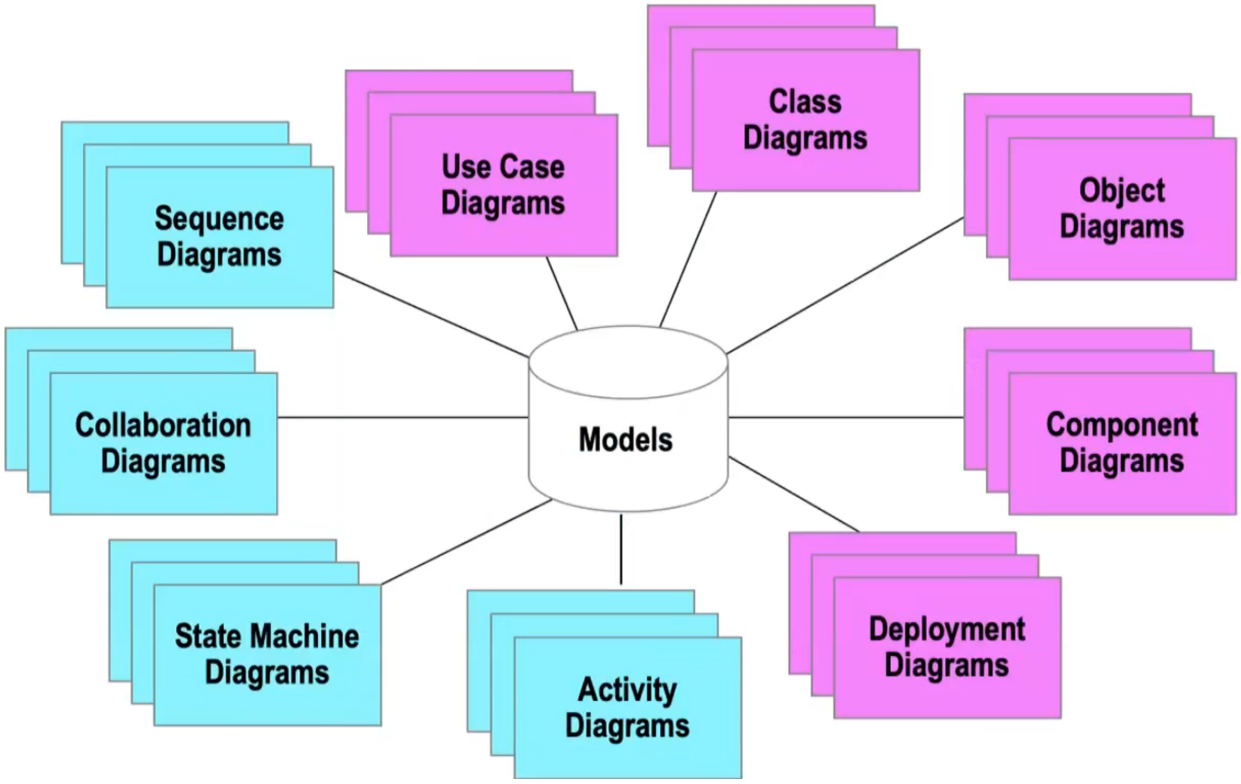
2.10.6 Microservice architectural style

Microservices are an evolution of SOA: they are based on the ideas of decomposing a monolithic application into smaller, independent components (like SOA), but they take this idea further by making the components smaller, more modular, and more autonomous. Indeed, each service uses its own technology stack and it’s isolated.



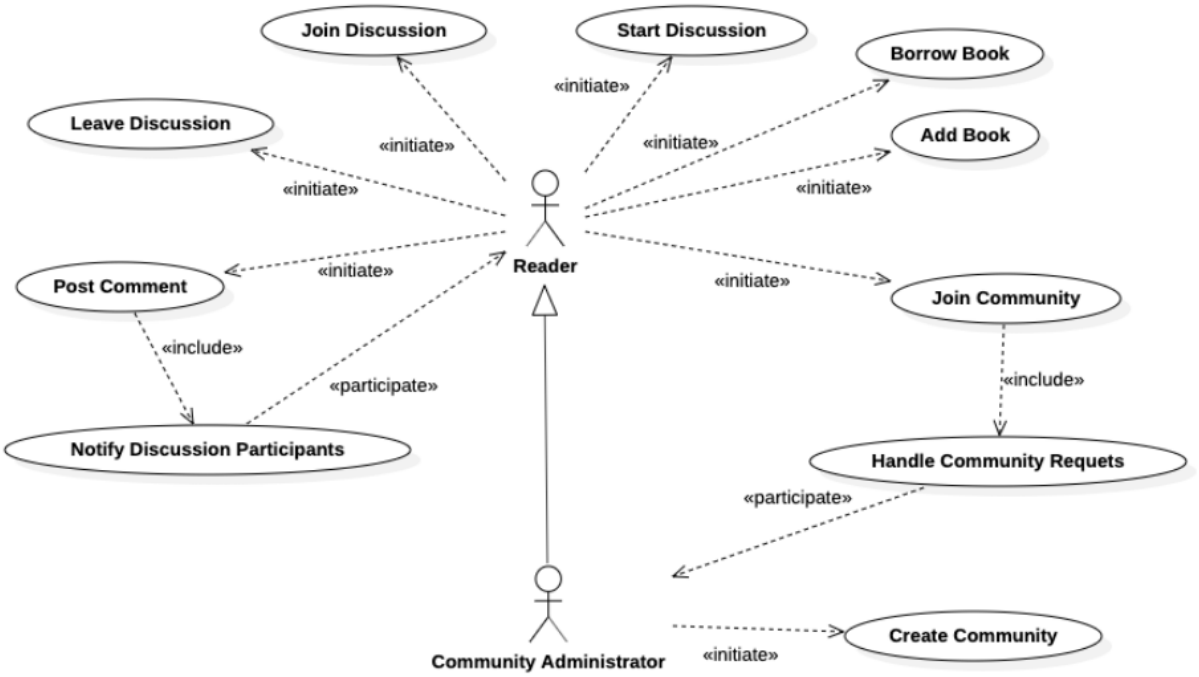
Microservices architecture is very resilience and scalable. Each service has its own software repository and there is no cross-dependencies between codebases. While monolithic systems are often big and complex and their replacement is risky and costly; replacing a microservice implementation is much easier. Microservices are commonly used to build complex, scalable applications: Netflix uses a microservices architecture, Amazon has microservices that handle tasks such as shopping cart management, order processing, and payment processing, Uber, Twitter for tasks such as feed generation, user authentication, and Airbnb.

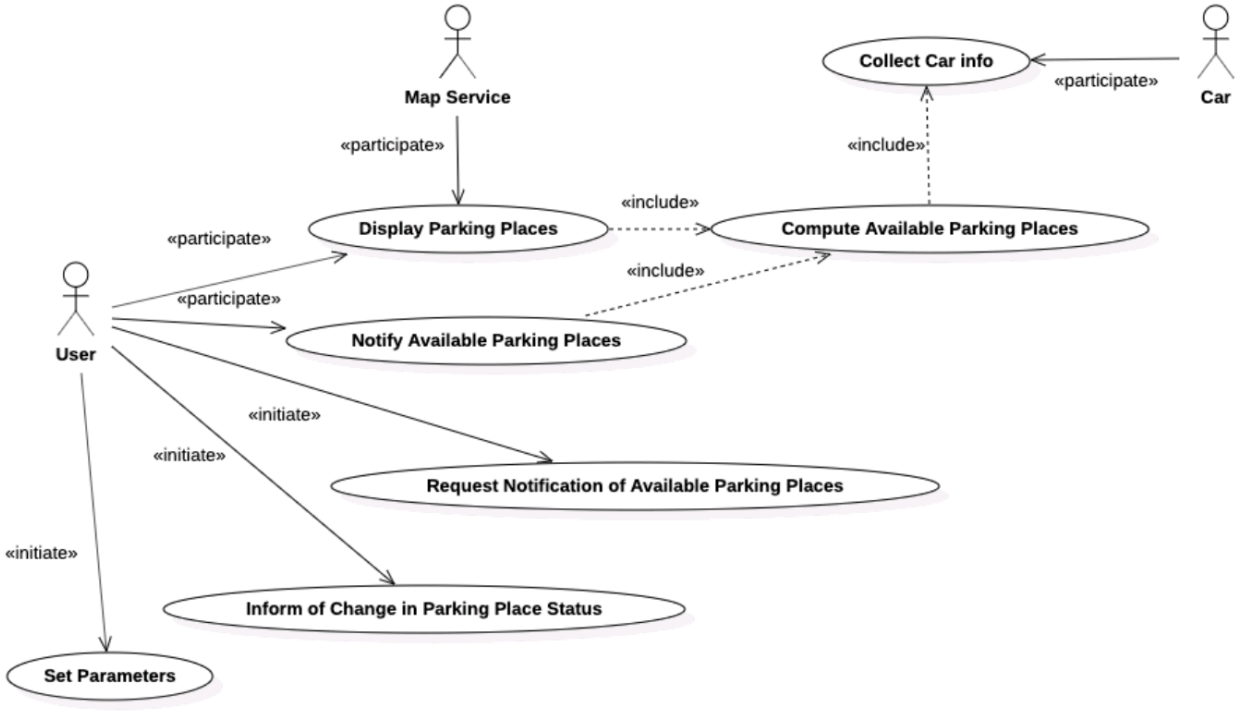
2.11 UML diagrams

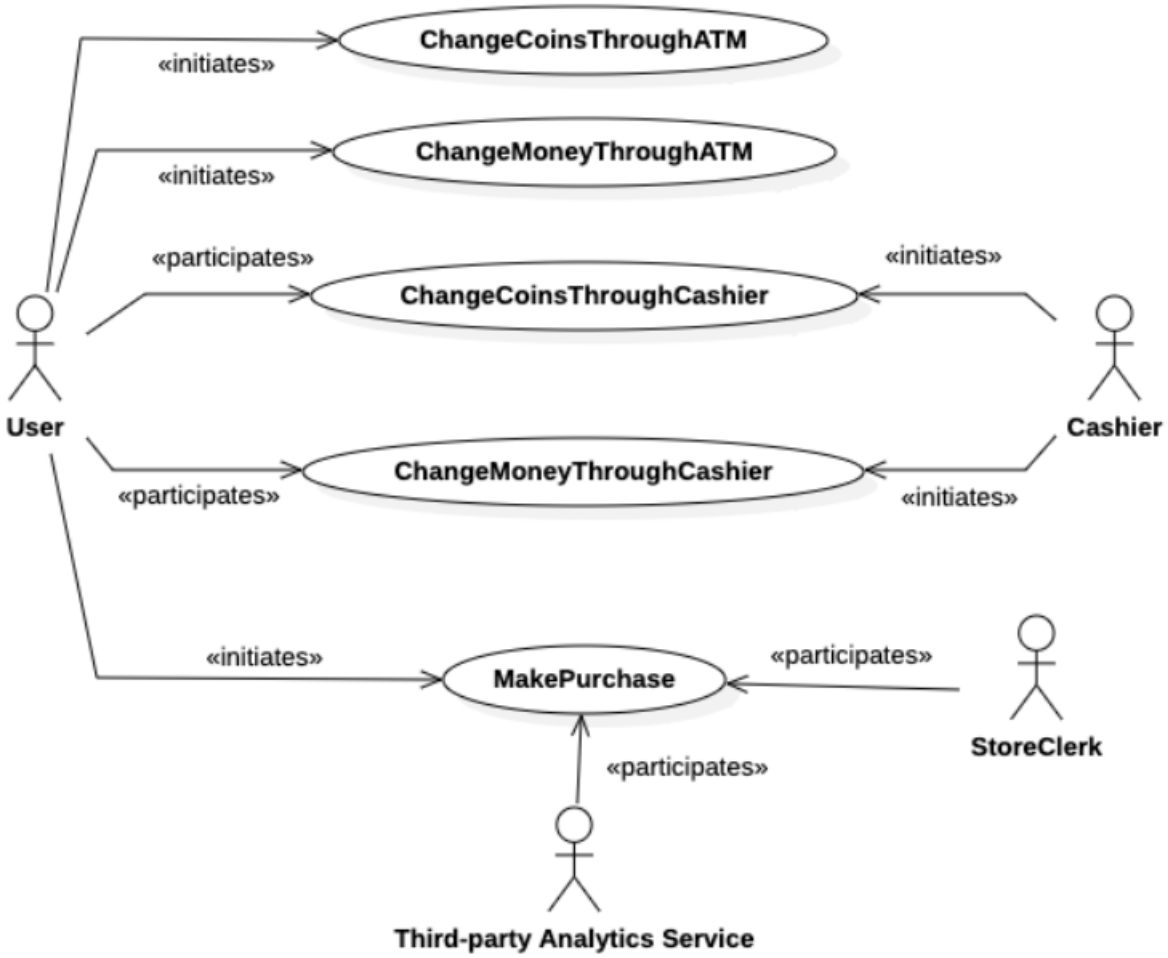


2.11.1 Use Case Diagram

A Use Case Diagram illustrates interactions between actors, the system and the ways it can be used, showing functional requirements and how the system will be used by actors.

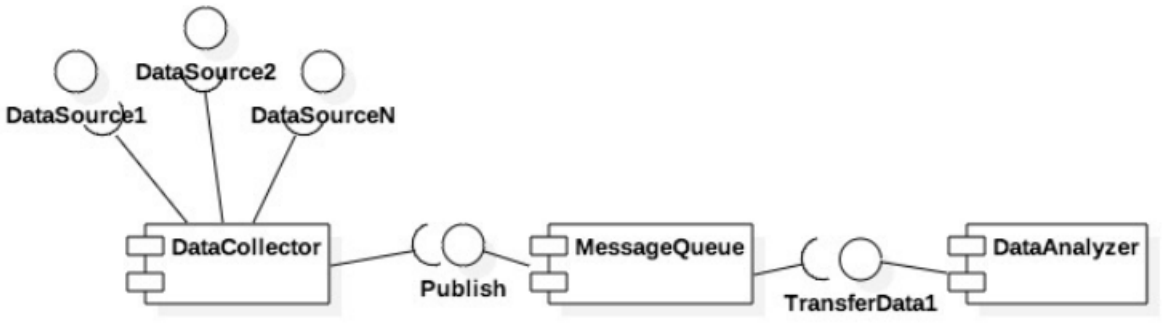

Join Discussion
Start Discussion
Borrow Book
Leave Discussion
Add Book
«initiate»
Reader
Post Comment
Join Community
«include»
«participate»
Notify Discussion Participants
Handle Community Requets
Community Administrator
Create Community


Map Service
Collect Car info
Car
«participate»
«include»
Display Parking Places
Compute Available Parking Places
User
Notify Available Parking Places
«initiate»
Request Notification of Available Parking Places
Inform of Change in Parking Place Status
Set Parameters

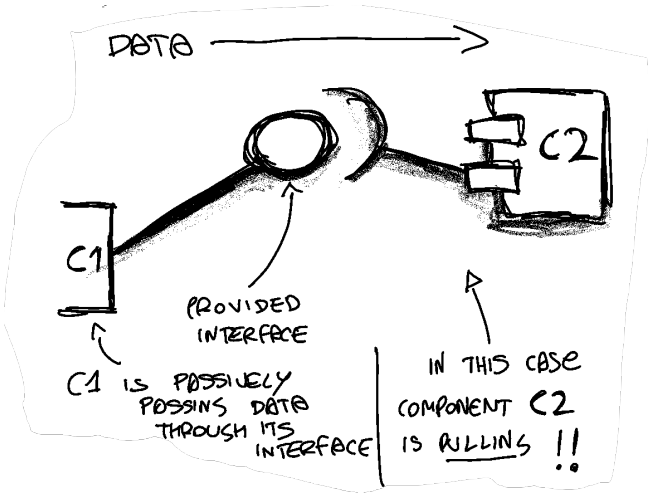


2.11.2 Component Diagram

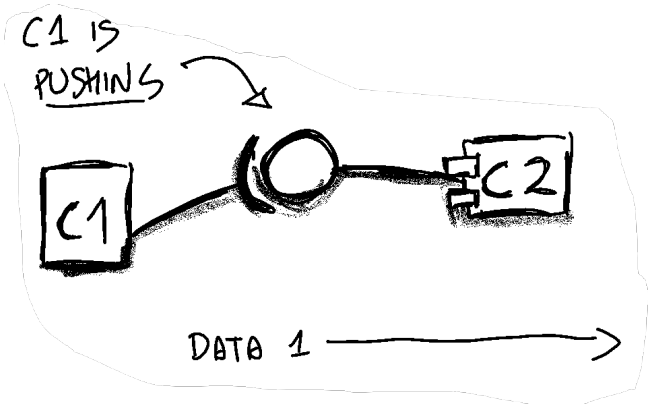
Component Diagram used to show interfaces between components.



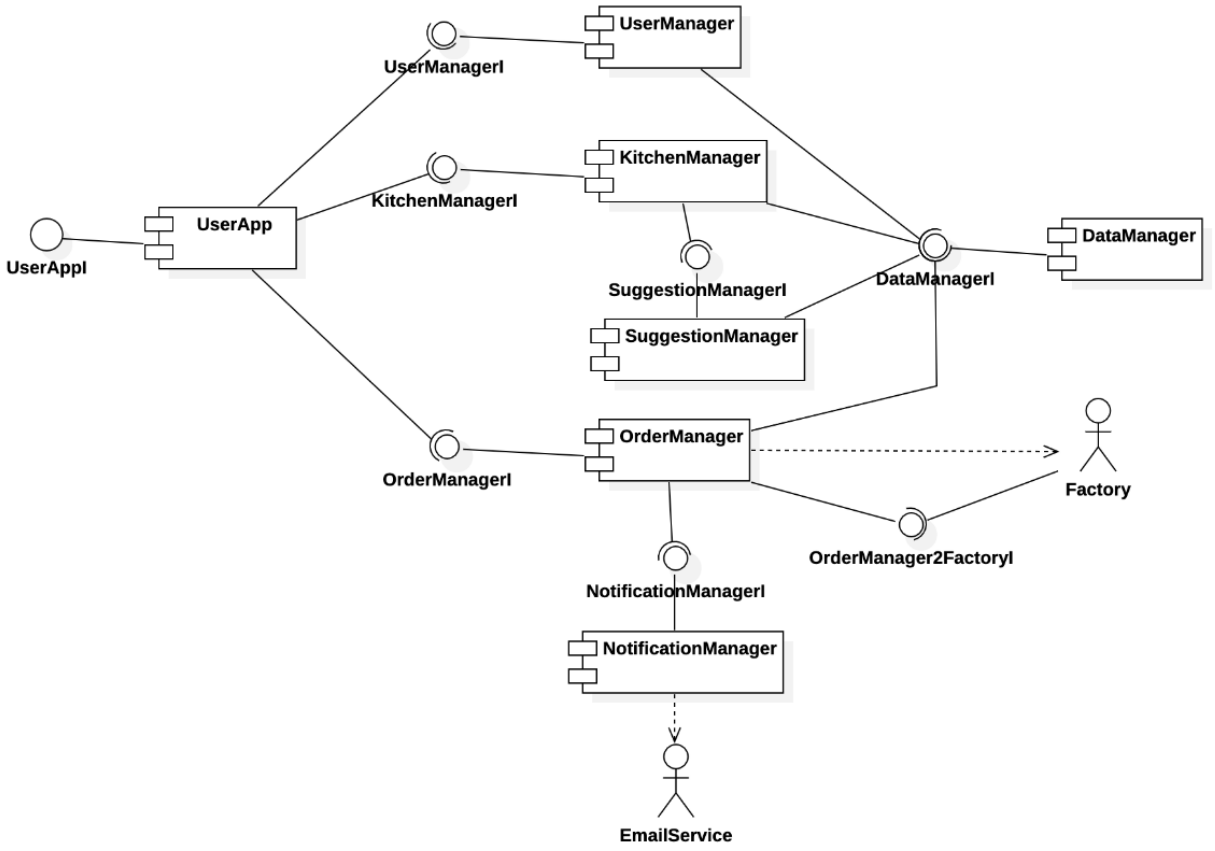
A component can passively pass data or can actively pass it pushing it. In the component diagram this aspect can be visualized trough the “half and full circles notation”. First example:

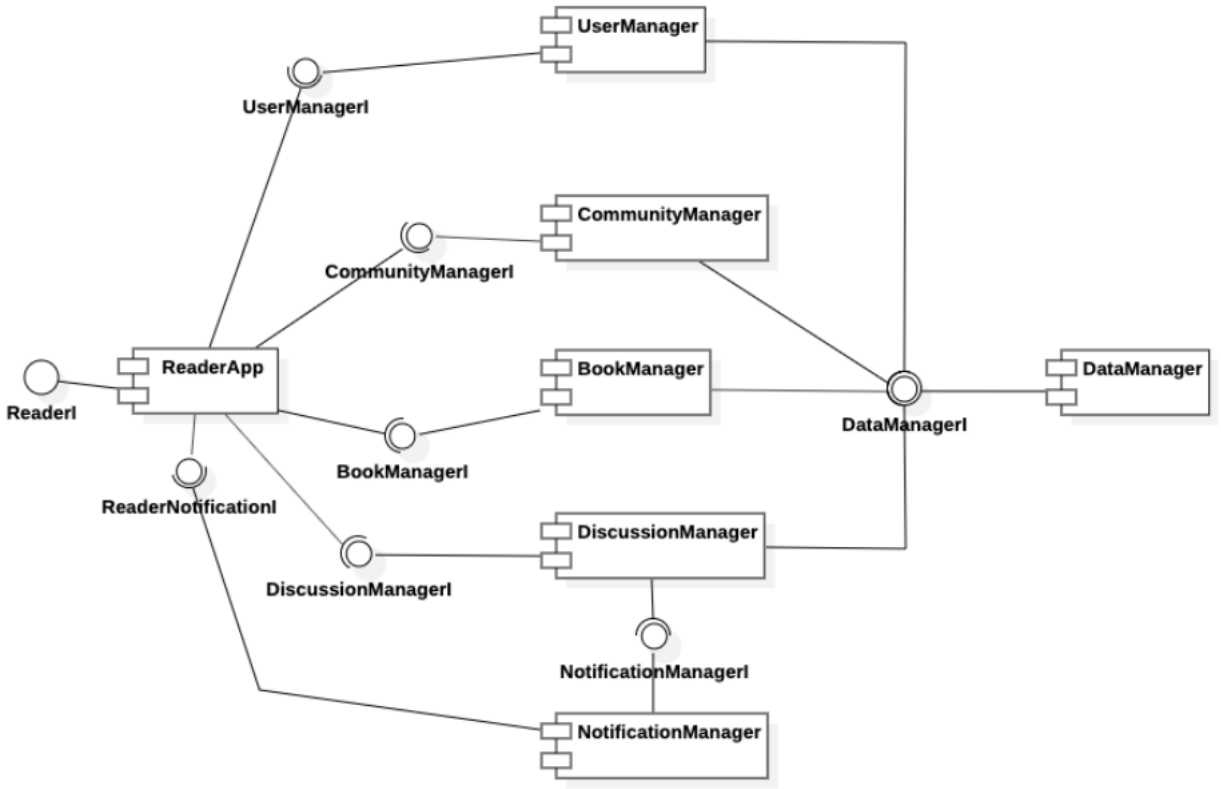


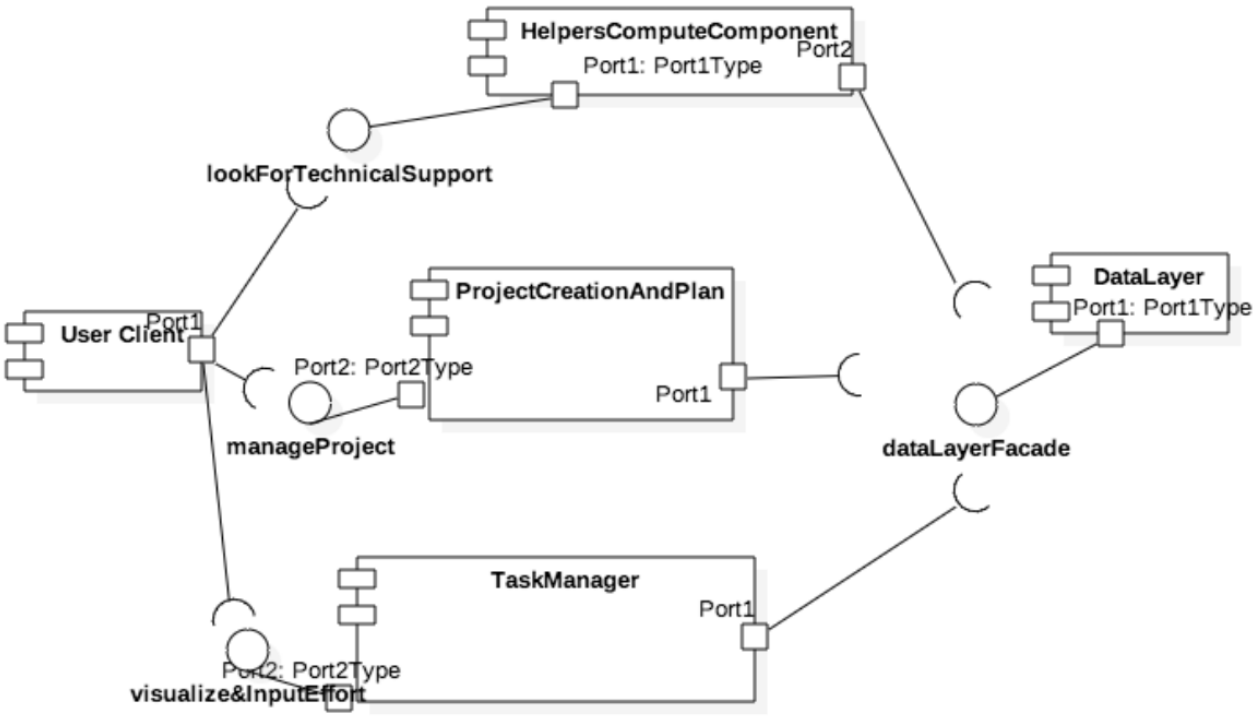
Second example:

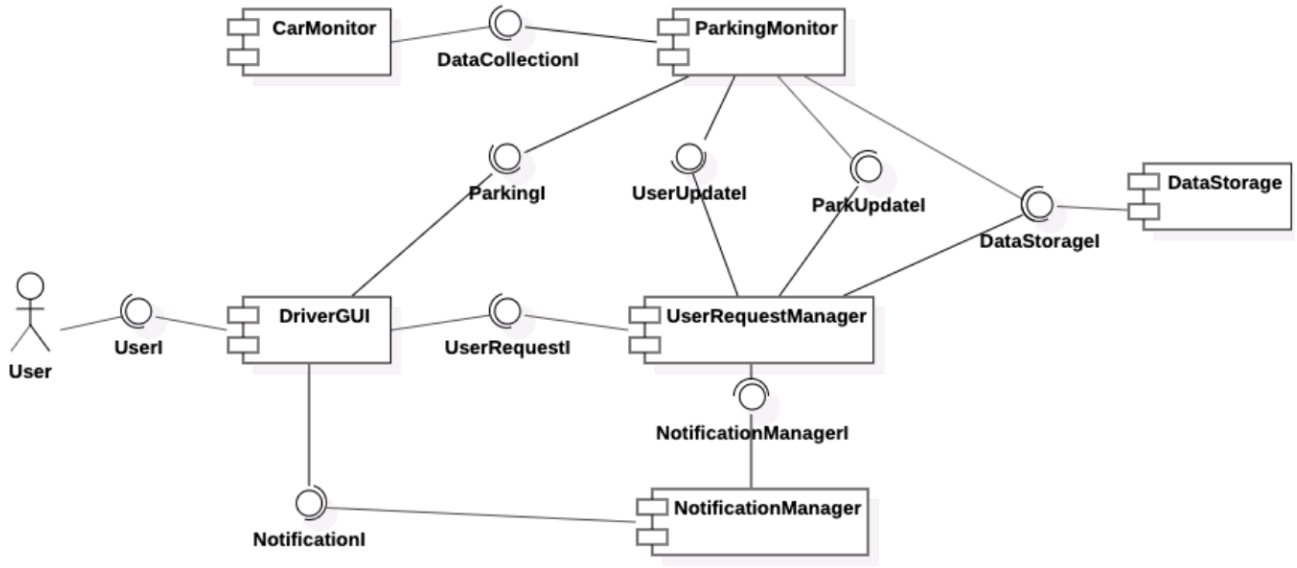


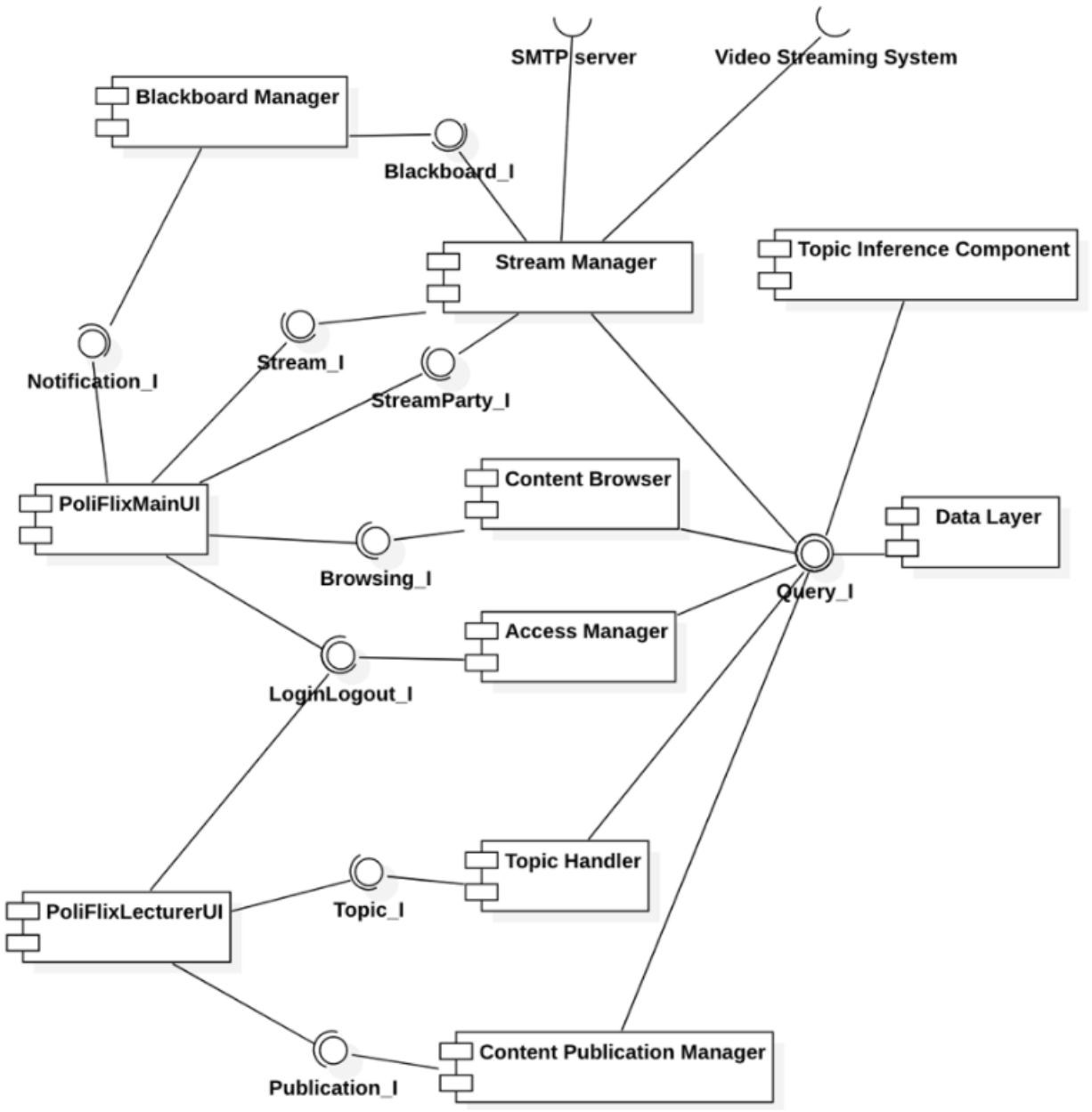
Another example of component diagram with actors:

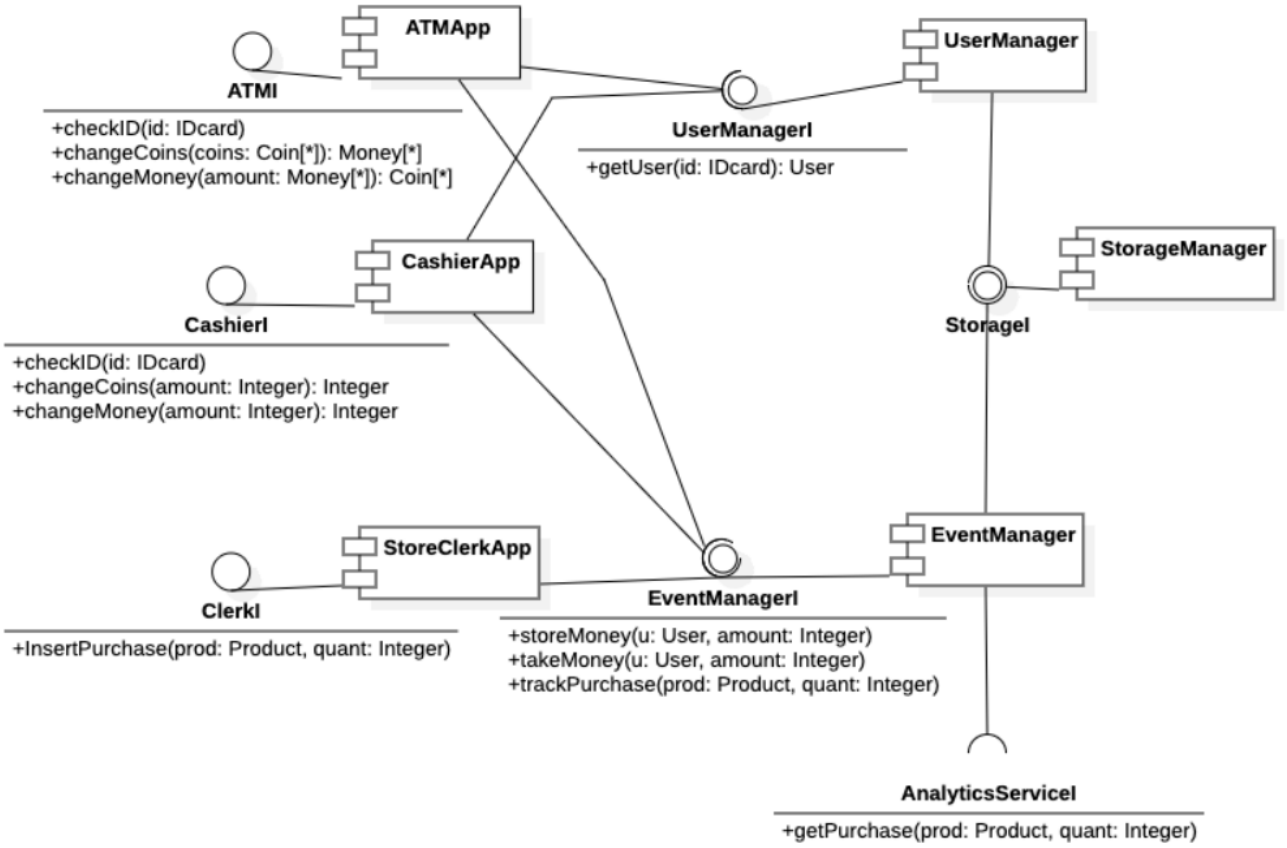

UserManager
UserManagerI
KitchenManager
KitchenManagerI
UserApp
UserAppI
SuggestionManagerI
DataManager
DataManagerI
SuggestionManager
OrderManager
OrderManagerI
Factory
OrderManager2FactoryI
NotificationManagerI
NotificationManager
EmailService


UserManager
UserManagerI
CommunityManager
CommunityManagerI
ReaderApp
ReaderI
BookManager
DataManager
DataManagerI
BookManagerI
ReaderNotificationI
DiscussionManager
DiscussionManagerI
NotificationManagerI
NotificationManager


HelpersComputeComponent
Port1: Port1Type
Port2
lookForTechnicalSupport
ProjectCreationAndPlan
DataLayer
Port1: Port1Type
User Client
Port1
Port2: Port2Type
Port1
manageProject
dataLayerFacade
TaskManager
Port1
Port2: Port2Type
visualize&InputEffort


CarMonitor
DataCollectionI
ParkingMonitor
ParkingI
UserUpdateI
ParkUpdateI
DataStorage
DataStorageI
User
UserI
DriverGUI
UserRequestI
UserRequestManager
NotificationManagerI
NotificationI
NotificationManager


SMTP server
Video Streaming System
Blackboard Manager
Blackboard_I
Stream Manager
Topic Inference Component
Notification_I
Stream_I
StreamParty_I
Content Browser
PoliFlixMainUI
Data Layer
Browsing_I
Query_I
Access Manager
LoginLogout_I
Topic Handler
PoliFlixLecturerUI
Topic_I
Content Publication Manager
Publication_I

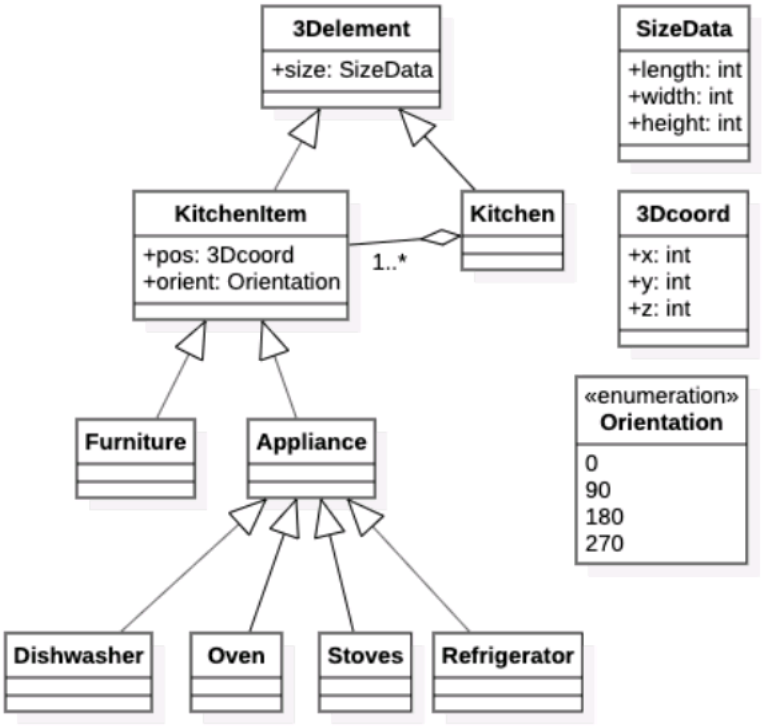


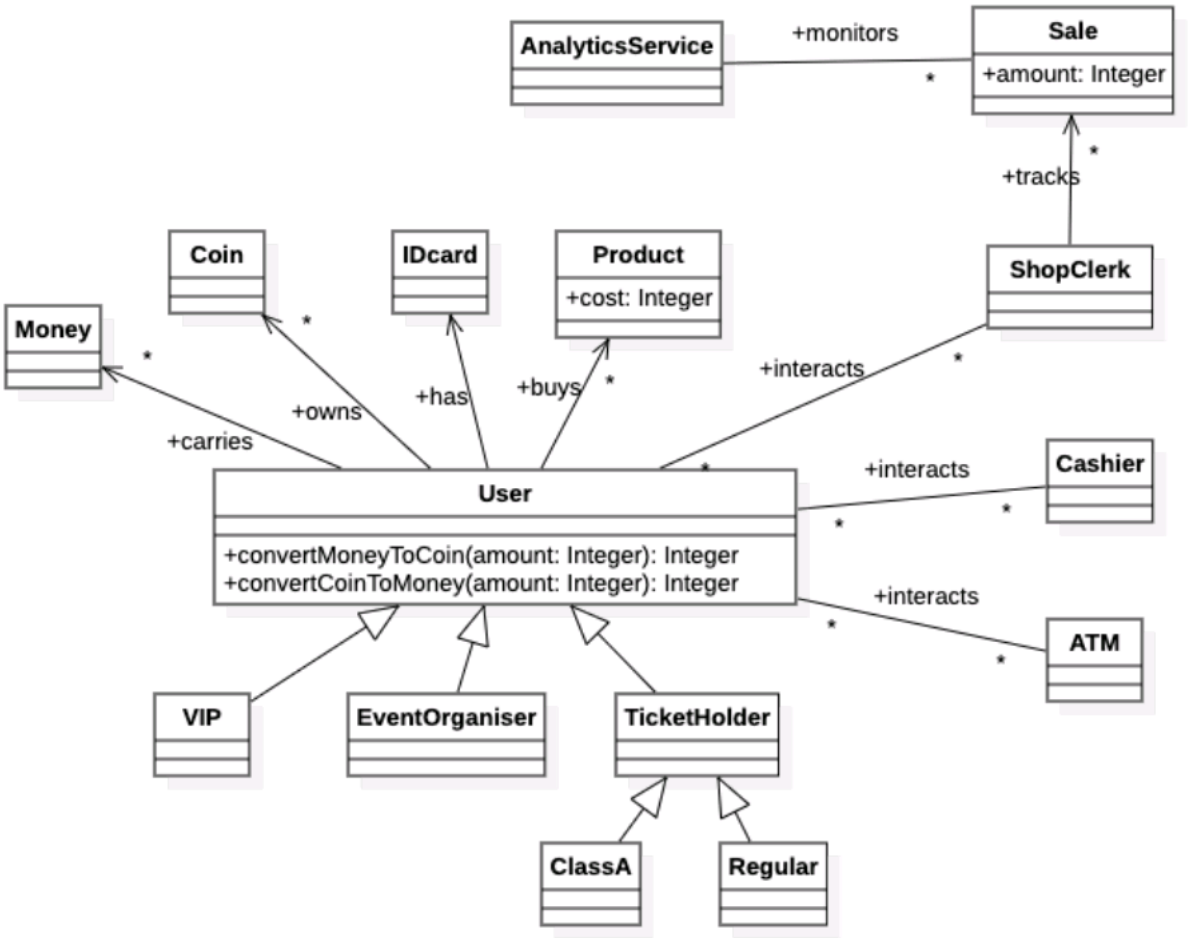
3 different front-end modules. Each one handle the interaction with a different actor through a different interface.

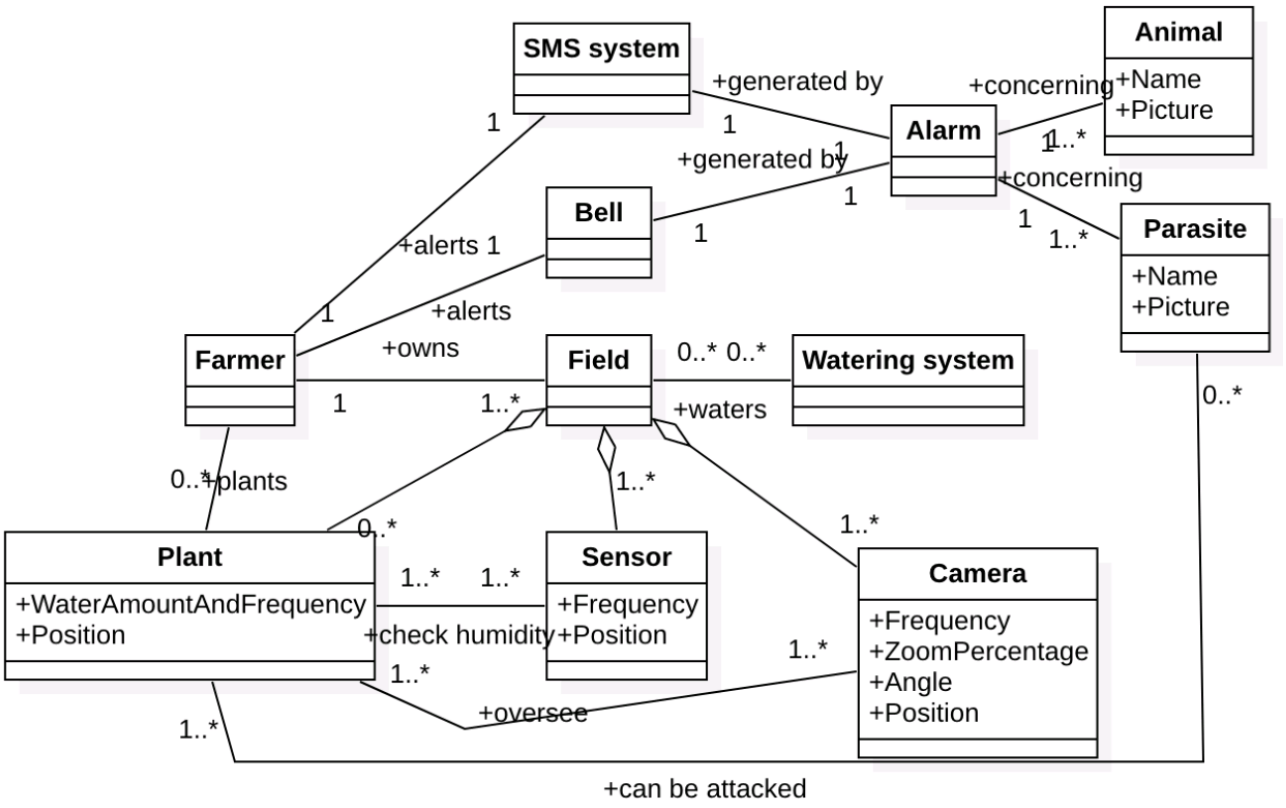
Suggestions:

- Remember the NotificationManager if there are any kind of notifications
- If there are other actors involved (as for example a Factory or and Email Provider or a Cloud provider) represent them!

2.11.3 Class Diagram

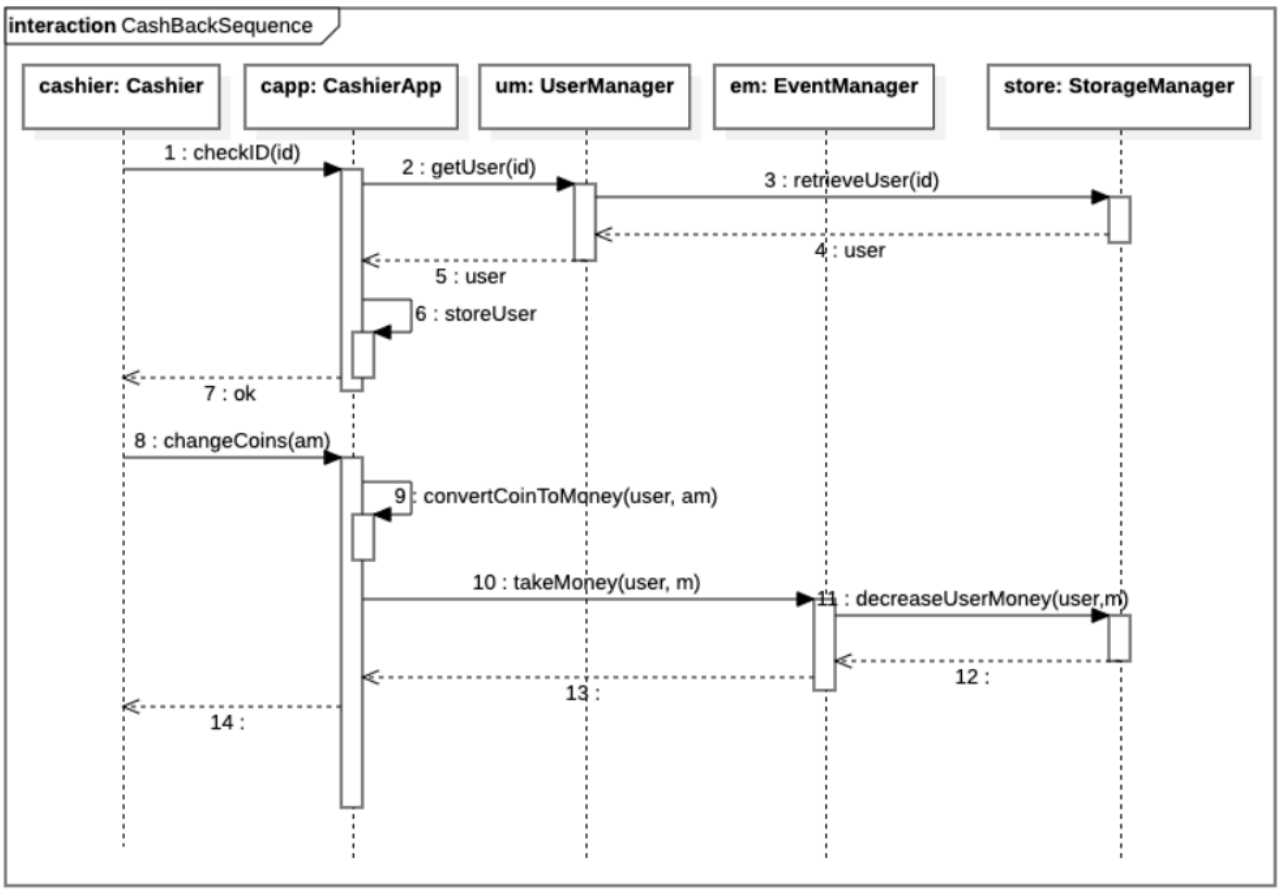


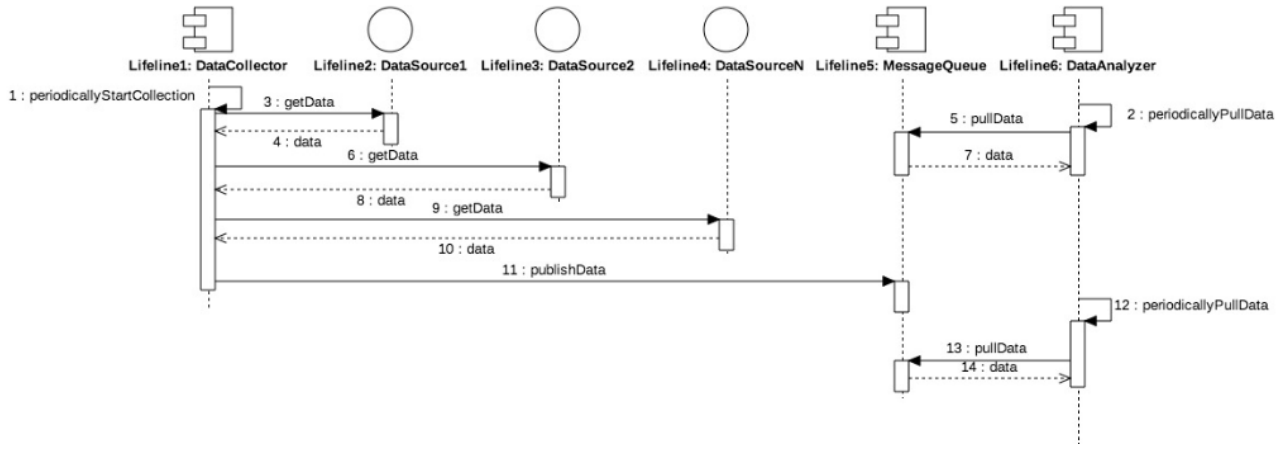

AnalyticsService
+monitors
Sale
+amount: Integer
+tracks
Coin
IDcard
Product
+cost: Integer
ShopClerk
Money
+owns
+has
+buys
+interacts
+carries
User
+convertMoneyToCoin(amount: Integer): Integer
+convertCoinToMoney(amount: Integer): Integer
+interacts
Cashier
+interacts
ATM
VIP
EventOrganiser
TicketHolder
ClassA
Regular

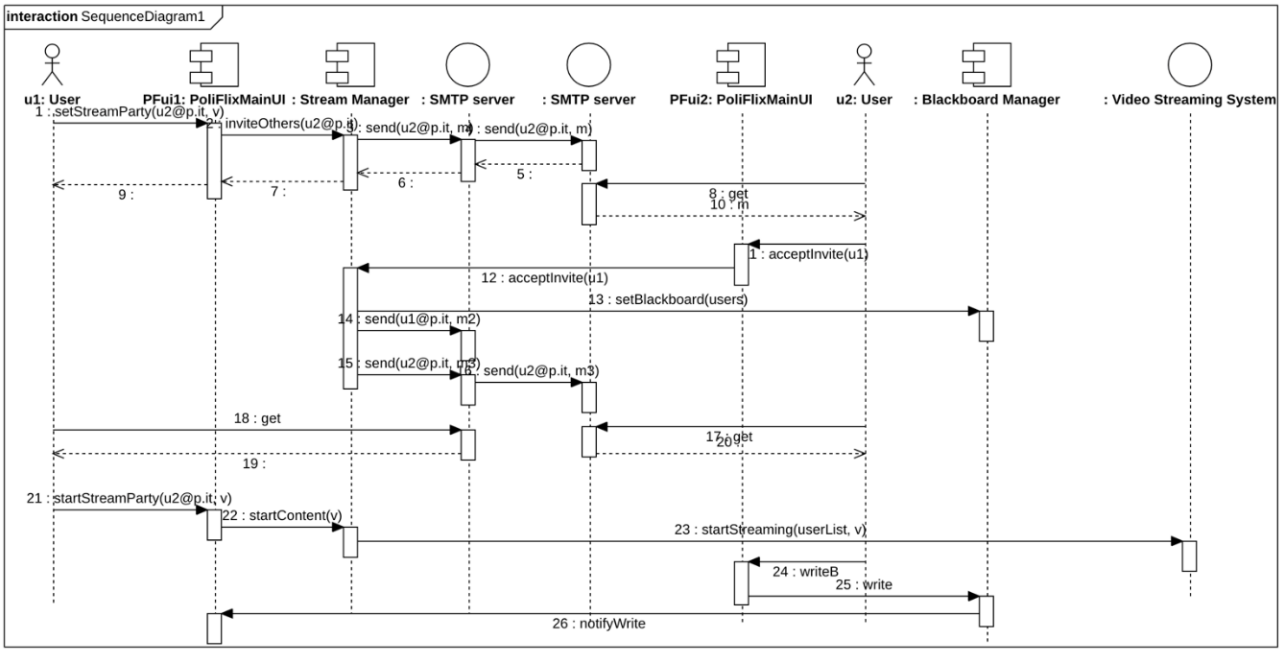


2.11.4 Sequence Diagram

Sequence Diagram used to describe behavior time-oriented. Focused on internal message exchange among components. It is used to show the control flow and illustrate typical scenarios.

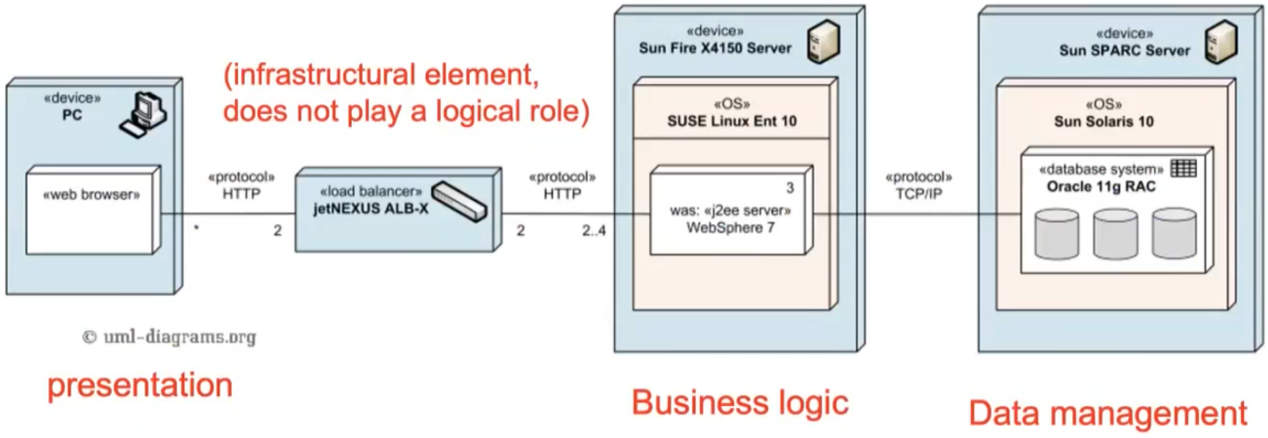

interaction CashBackSequence
cashier: Cashier
capp: CashierApp
um: UserManager
em: EventManager
store: StorageManager
1 : checkID(id)
2 : getUser(id)
3 : retrieveUser(id)
4 : user
5 : user
6 : storeUser
7 : ok
8 : changeCoins(am)
9 : convertCoinToMoney(user, am)
10 : takeMoney(user, m)
11 : decreaseUserMoney(user,m)
12 :
13 :
14 :

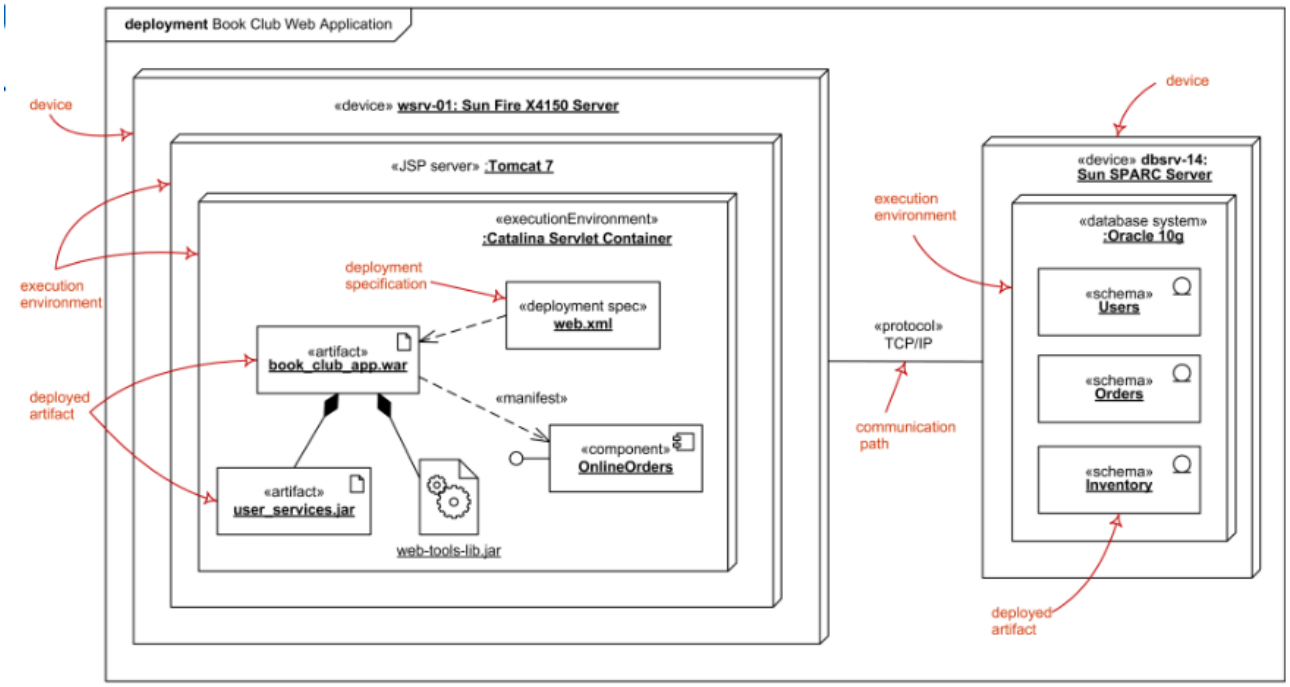

Lifeline1: DataCollector
Lifeline2: DataSource1
Lifeline3: DataSource2
Lifeline4: DataSourceN
Lifeline5: MessageQueue
Lifeline6: DataAnalyzer
1 : periodicallyStartCollection
2 : periodicallyPullData
3 : getData
4 : data
5 : pullData
6 : getData
7 : data
8 : data
9 : getData
10 : data
11 : publishData
12 : periodicallyPullData
13 : pullData
14 : data



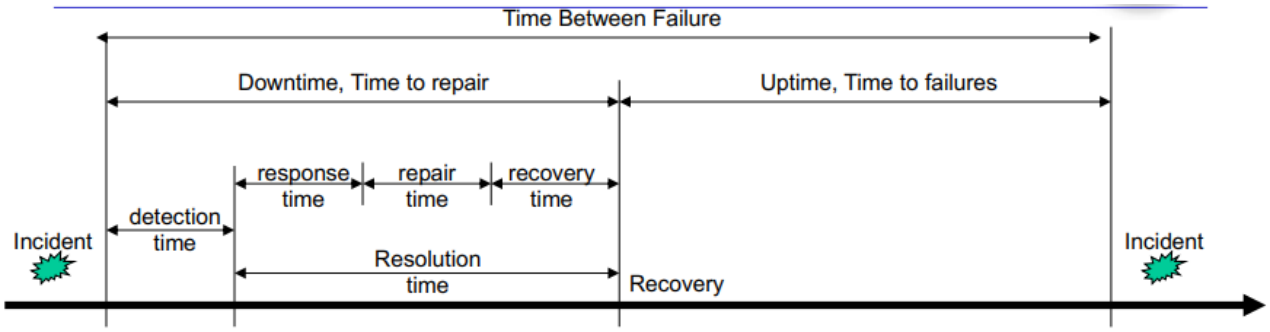
2.11.5 Deployment Diagram

Deployment Diagram captures topology of a system: hardware, software and communications protocol between them. It is used to specify the distribution of components and identify performance bottlenecks.





3 Availability



A couple of parameters:

- Mean Time to Repair ($MTTR$): time between the occurrence of a fault and service recovery, also known as the mean downtime.
- Mean Time To Failures ($MTTF$): time between the recovery from one incident and the occurrence of the next incident, also known as up-time.
- Mean Time Between Failures ($MTBF$): Mean time between the occurrences of two consecutive incidents

Then we can define availability as:

$$A = \frac{MTTF}{(MTTF + MTTR)}$$

The probability that a component is working properly at time (actual uptime).

Reliability: means that the service is available for an agreed period without interruptions (frequency of interruptions).

$$R = e - \lambda t \text{ where } \lambda = \frac{1}{MTTF}$$

Note that a system could be available for 99% of time but have a lack of reliability since it continues to 'crash' and quickly restart.

3.1 Solve availability in practice

Simple rules:

- Components in series, so the total availability is the multiplication of all the components availability.

$$A_{System} = A_1 * A_2 * \cdots * A_N$$

- Components in parallel:

$$A_{System} = (1 - (1 - A_1) * (1 - A_2) * \cdots * (1 - A_N))$$

- The main rule to increase the availability of a system is to add in parallel another component of the one with lesser availability.

- We repeat this process until we obtain the desired availability. # JEE

Java Platform Enterprise Edition (JEE) is a software framework for developing enterprise applications in Java. Enterprise applications are designed to handle large amounts of data and often involve complex tasks such as source control, multi-threading, synchronization, concurrency, transaction processing, distributed systems, messaging, and service management.

JEE includes several components that assist with these tasks:

- Enterprise JavaBeans (EJBs), which contain the main business logic of the application
- Java Persistence API (JPA), which manages database connectivity and data management
- Java Messaging Service (JMS), which provides a common interface for messaging protocols to facilitate communication between distributed systems
- Java Transaction API (JTA), which specifies standard interfaces between a transaction manager and the parties involved in a distributed transaction system.

JEE follows a multi-tier architecture, with tiers for the:

- client
- web
- business
- and enterprise information system (which includes the database management system and other technologies).

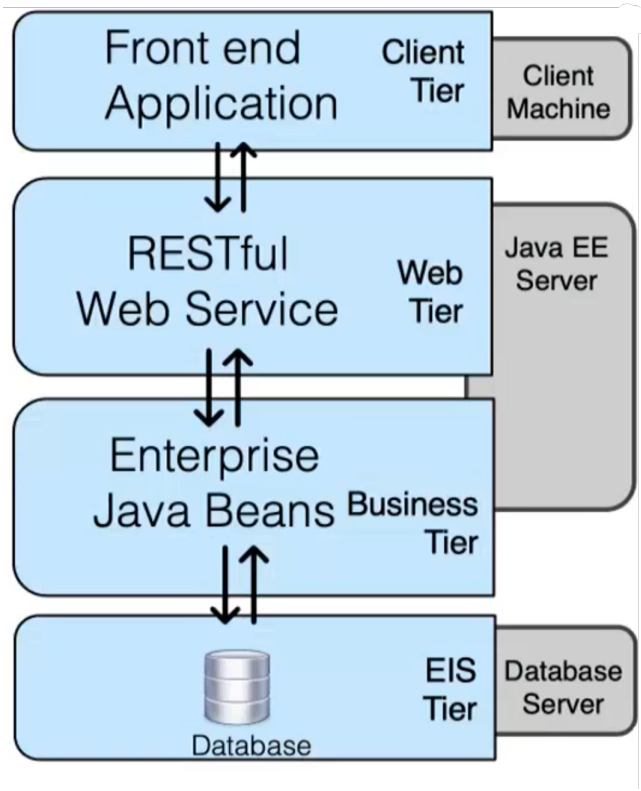
Each tier consists of logical groups of components that serve a specific purpose. Components are self-contained software units that are deployed onto the tiers of an enterprise application.

3.2 Client tier

Java applets, which are small Java programs that are designed to be run on a client computer, were once a popular way to add interactive features to websites. However, they fell out of favor due to security concerns, as applets have the ability to run arbitrary code on the client computer, which could potentially be exploited by malicious actors. As a result, applets are no longer widely used, instead JavaScript and HTML5 are used to add interactive features to websites.

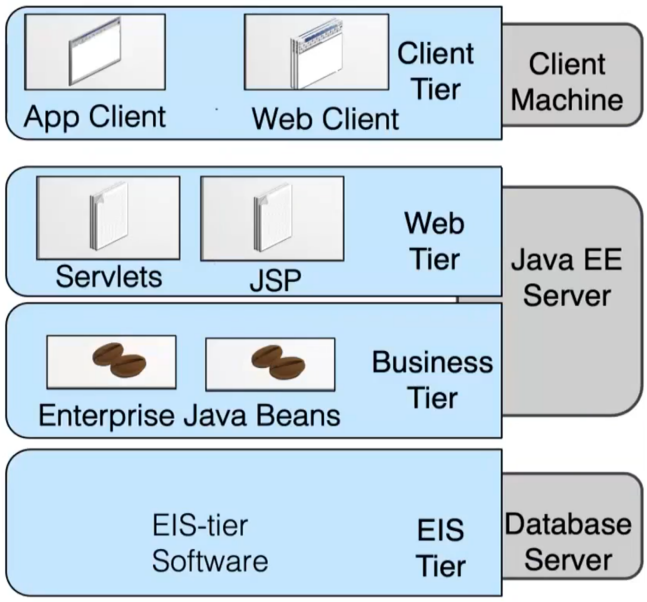
3.3 Web tier

EJBs were once a popular choice for building enterprise applications, but over time, other technologies have become more widely used. In this course we will still study EJB since they are still used as legacy technology.



The Java EE Server of the application is composed of various JSPs containing HTML forms and Servlets to manage them.

- **JSP**: JSP pages are typically used to create the front-end of a web application, which is the part of the application that the user interacts with, for example: a page used to registration.
- **Servlets**: Java classes that run on the server and handle requests and responses between the client and server. Generally Servlets receive user input from the JSP front-end and forwards it to the corresponding bean.
- **Bean**: a bean is a Java class that is used to store and retrieve data, often for the purpose of passing data between different layers of an application.



3.3.1 Stateful and stateless beans

Session beans are of three types:

- **Stateful Session Beans**: each bean has only one client and there is a ''conversational state", which maintains across method invocations.
- **Stateless Session Beans**: there is no maintain any kind of ''conversational state" , there is just a reply to a a request.
- Singleton Session Beans: unique bean, generally used to keep system level configurations.

Examples of stateful session beans applications:

- A shopping cart for an online store that maintains a list of items a customer has added to their cart.
- A customer service chat application that maintains the conversation history with a customer.
- A social media application that maintains a user's friends list and activity feed.
- A financial management application that maintains a user's budget and expenses.
- A fitness tracker that maintains a user's workout history and progress.

Examples of stateless session beans applications:

- A currency conversion service that performs conversions between different currencies.
- A weather information service that provides current and forecasted weather data for a location.
- A calculator that performs mathematical calculations.
- A search engine that provides search results for a query without any tracking information.
- A translation service that translates text from one language to another.

3.3.2 Examples of beans methods

Methods provided by UserBean:

1. `UserEntity registerNewUser(String username, String password)` : creates and persists a new user in the database, returns the persisted UserEntity, throws an exception if the user already exists.
2. `boolean loginUser(String username, String password)` : queries the database for a user, compares passwords and returns true if match, otherwise false.

3. `boolean existsUser(String username)` : queries the database and returns true if user exists, otherwise false.
4. `UserEntity getUserById(String username)` : queries the database and returns the corresponding UserEntity if exists, otherwise null.

Methods provided by ArticleBean:

1. `ArticleEntity insertNewArticle(String title, String body, UserEntity author)` : creates and persists a new article in the database, returns the persisted ArticleEntity
2. `void deleteArticle(ArticleEntity article)` : Removes an article from the database.
3. `ArticleEntity getArticleById(Integer articleId)` : queries the database and returns the corresponding `ArticleEntity` if exists, otherwise 'null.
4. `List<ArticleEntity> getAllArticles()` : Queries the database for a list of all persisted articles.

3.3.3 JMS APIs

The Java Messaging Service (JMS) is a messaging system that consists of a JMS provider, JMS clients, and messages that are exchanged between components. JMS supports point-to-point (P2P) and publish/subscribe messaging models. In the P2P model, messages are sent to specific queues and are received by receivers, while in the publish/subscribe model, messages are broadcast to subscribers using topics.

3.3.4 JPA

JEE specification and API that simplifies the interaction with back-end databases providing support for relational/SQL/NoSQL databases. JPA provides an object-oriented view of the DB tables (entities in the ER model):

- A table in the DB is represented by a class
- Table rows are instances of this class Columns map to fields (attributes)
- Relations between two tables are defined by properties that are common in the two classes
- Multiplicity of a relation is declared using annotations

3.3.4.1 Basic JPA stuff

`@Entity` annotation is mandatory while `@Id` indicates the primary key.

```
@OneToOne
@JoinColumn ( name  = "mailID")
private MainlingAddress address;

//in MainlingAddress there the attributed annotated as ''@Id" is called "mailID"
```

Main entity relationship types of JPA are (for example, `Customer` is in `OneToMany` with Order):

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

Entities are managed by `EntityManager` :

```
@PersistenceContext
private EntityManager em;
```

It can be used to make queries like for example this selection:

```
public Product findProduct(int productid) {
    return em.find(Product.class, productid);
    }
```

Mainly there are two EM patterns:

- direct access
- DAO (Data Access Object): better separation and better design for change. It works like an interface so that you can have for example multiple database backends that are encapsulated by the DAO. # Verification and Validation

Two main approaches:

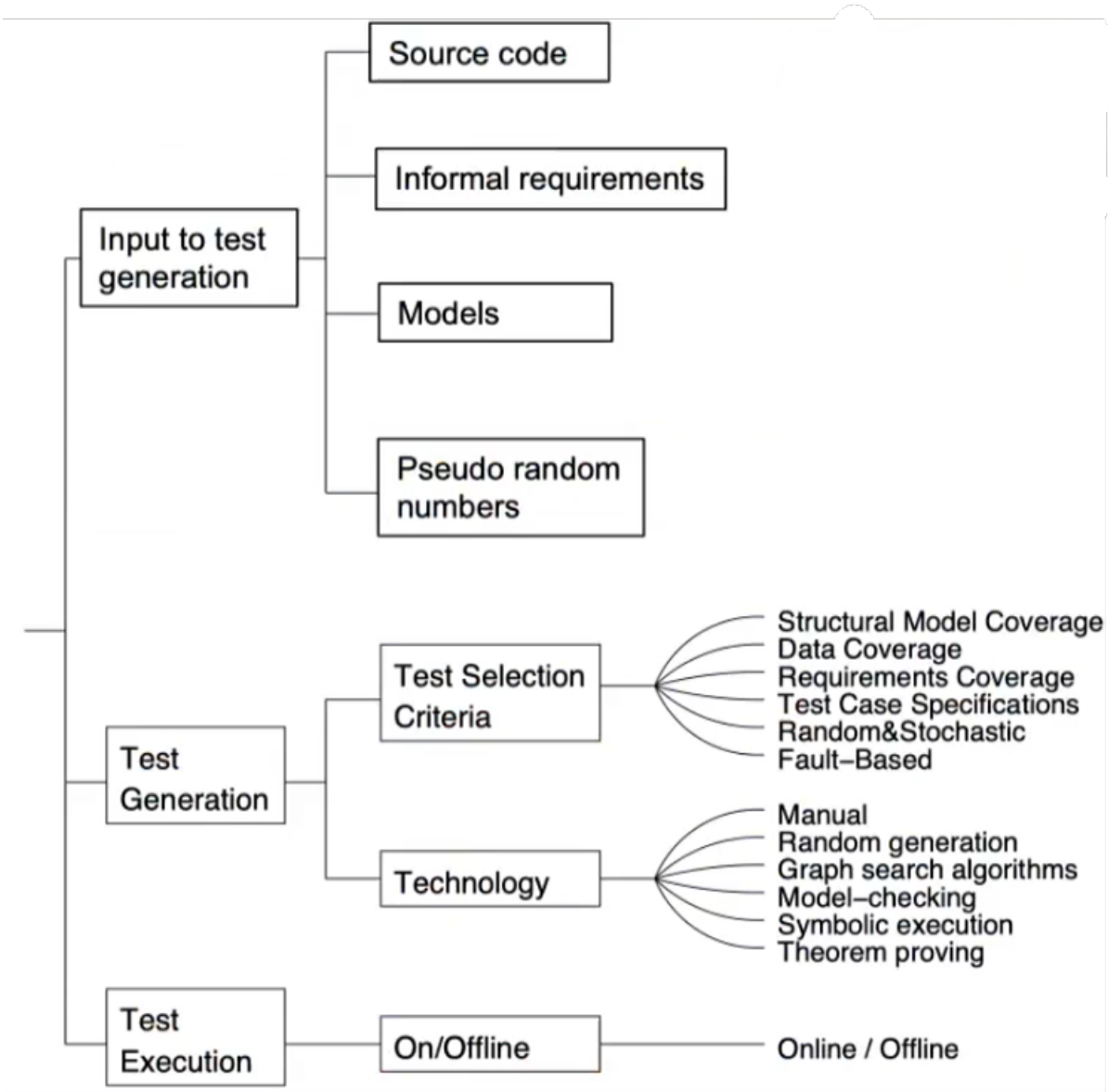
- ANALYSIS (static) analytic study of properties
- TESTING (dynamic) experimenting with behavior of the product sampling behaviors and finding counterexamples

3.4 Testing

Testing(../../../BSc([/Ingegneria%20del%20Software/src/10.Testing.md)

Some def related to testing:

- Random testing is a testing approach where test cases are generated randomly, without considering the structure or behavior of the software being tested.
- Systematic testing is an approach that uses characteristics or structure of the software, or information about its behavior, to guide the selection of test cases.
- Unit testing is a form of testing that is conducted by developers to test individual units of code, and integration testing is a form of testing that focuses on exercising the interactions between different modules or components of a system.
- System testing is a form of testing that is conducted on the complete system, once it has been integrated, and is designed to identify bottlenecks and other issues.
- Scaffolding is temporary support structures used to assist in testing
- black box testing is a form of testing that focuses on the external behavior of a system, rather than its internal structure or implementation. When testing a system that acts like a state machine, two common criteria for coverage are state coverage and transition coverage.



3.5 Analysis

A family of techniques which don't involve the actual execution of software. Two main approaches:

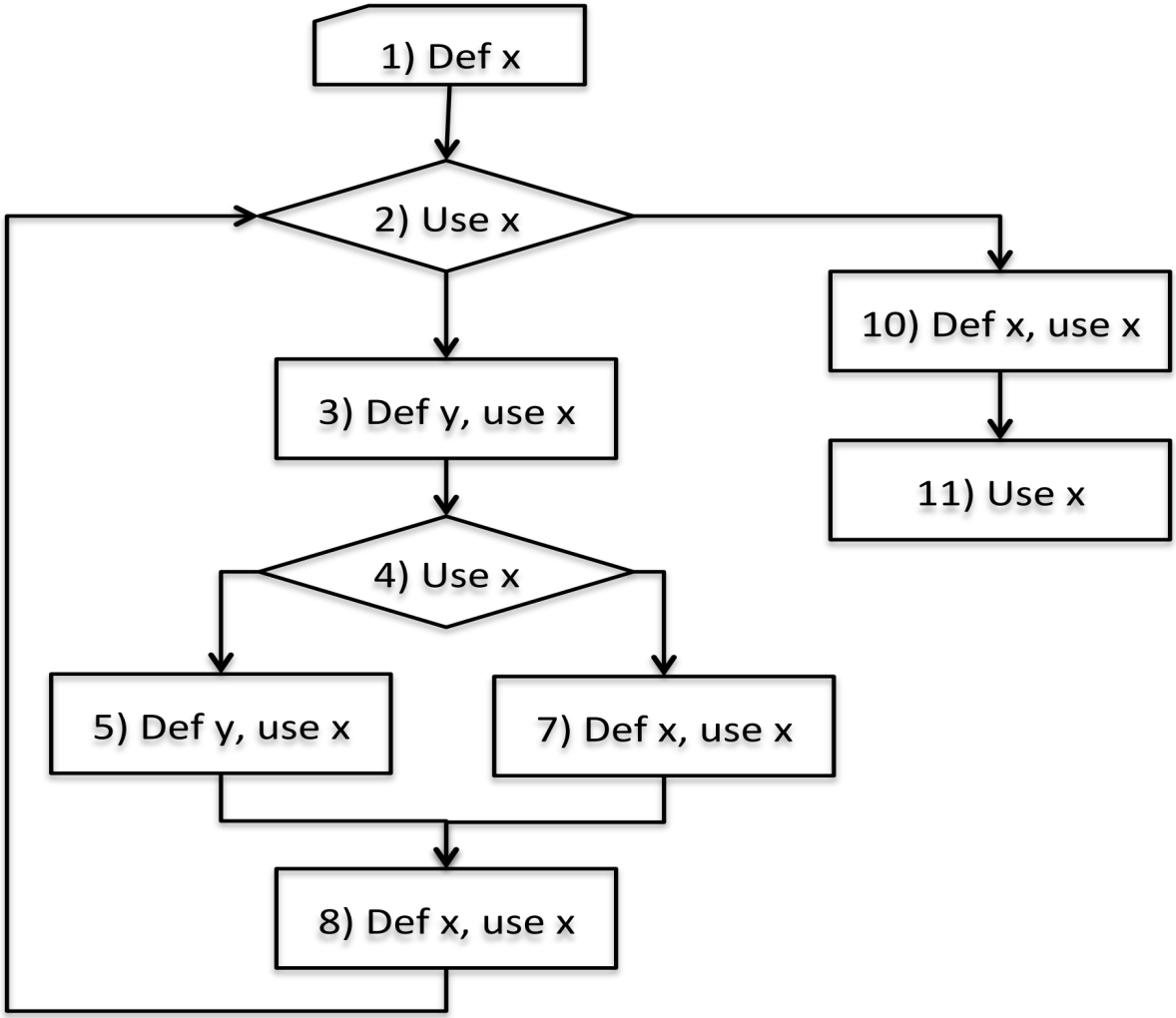
- **Manual inspection**: (informal) Review is an in-depth examination of some work product by one or more reviewers. Product is anything produced for the lifecycle, i.e., requirements, plans, design, code, test cases, documentation, manuals, everything.

- **Algorithmic static analysis**: a formal evaluation technique (systematic) in which software requirements, design, or code are examined in detail. Two different types:
  - **<def-use>** analysis or **data flow analysis**: focused on usage of variables and possible errors related to them. Typically used by compilers to find symptoms of possible errors:
    - a variable is uninitialized when used
    - a variable assigned and then never used

    - a variable always get a new value before being used
- **Symbolic execution**: is a technique which involves representing values in a program as **symbols**, and executing statements to compute new expressions.

3.5.1 <Def-use> analysis

1) Derive the control flow diagram
2) Identify points where variables are defined and used.
3) Using basically algorithms that explore the graph and check the variables, analyzing the pairs



3.5.2 Symbolic Execution

The symbolic execution can, using mathematical and logical inference on the variables symbolic value, to highlight problems and bugs which are not possible to see using def-use analysis. It is often used in conjunction with def-use analysis to identify potential errors and optimize code.

```
1  main() {
2    int a, h, f, q;

3    scanf("%d", &a);
4    scanf("%d", &q);
5    h = q - 2;
6    while (a > 0){
7      if (q == h + 2)
8        f = a;
9      else if (a > f)
10       f = a;
11     scanf("%d", &a);
12     h = h+1;
13   }
14   printf(%d", f);
15 }
```

The paths leading to the potential uses of f before its initialization are the following: 3, 4, 5, 6, 14 and 3, 4, 5, 6, 7, 9.
Symbolic execution through path 3, 4, 5, 6, 14 gives the following result:
3: a = A
4: q = Q
5: h =Q-2
6: A ≤ 0
Hence, the path is indeed feasible with condition A≤0, so this is a real potential problem that can occur in the program.

Path 3, 4, 5, 6, 7, 9, instead, is not feasible. In fact, symbolic execution gives the following result:
3: a = A
4: q = Q
5: h =Q-2
6: A > 0
7: Q ≠ Q-2+2
which yields the contradictory condition Q ≠ Q. Hence, this is a false positive that def-use analysis produces.

Project Management

Main steps of Project Management are:

1. INITIATING:
   - Define the project
   - Define initial scope
   - Estimate cost and resources
   - Define the stakeholders
2. PLANNING:
   - Scope management plan
   - ***Schedule planning*** <- (Gantt chart)
   - ***Cost and Effort estimation*** <- (Function Points)
   - Quality management plan
   - Change management plan
   - Communication management plan
   - Risk management plan
3. EXECUTING
4. ***MONITORING AND CONTROLLING*** <- EVA
5. CLOSING

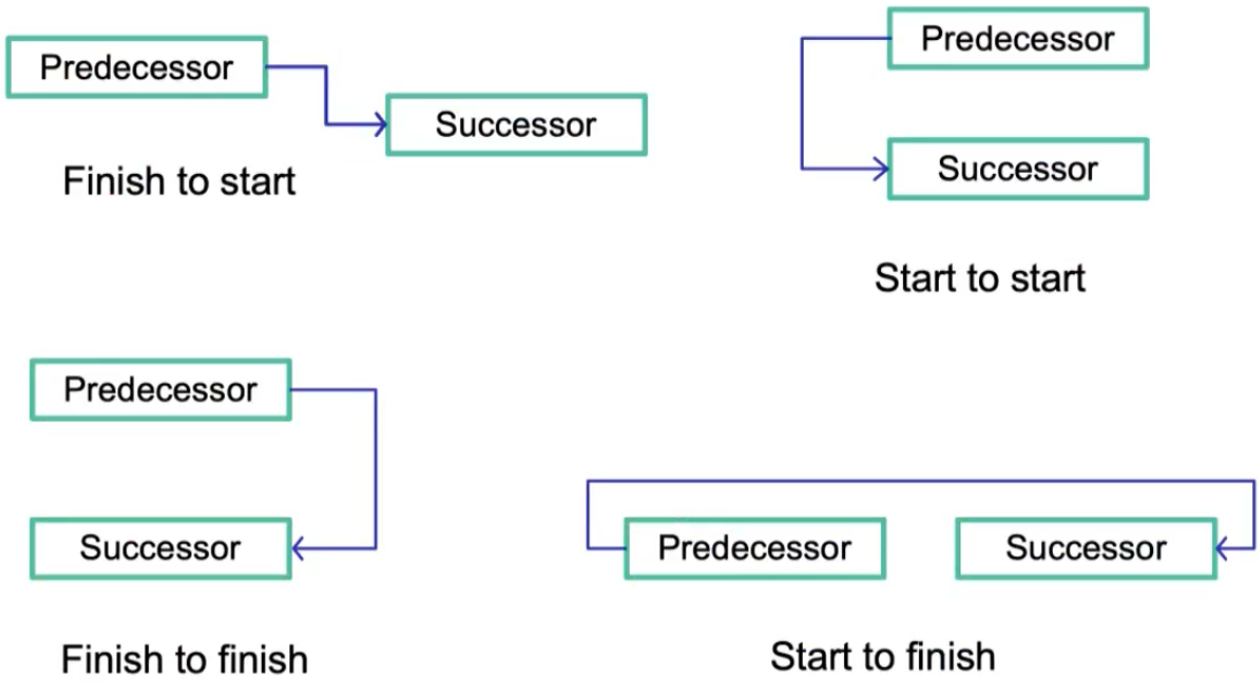
3.6 Schedule planning and Gantt Chart

- Tasks are activities which must be completed to achive the project goal.
- Milestones are points in the schedule where progress can be assessed.
- Deliverables are work products delivered to the customer (e.g documents).

How schedule is developed, managed, executed and controlled?

- Break down project in tasks
- Define dependencies between tasks
- Define lag time between dependencies (even negative).

There can be different dependencies between tasks:



The **critical path** is a sequence of tasks that runs from the start to the end of the project:

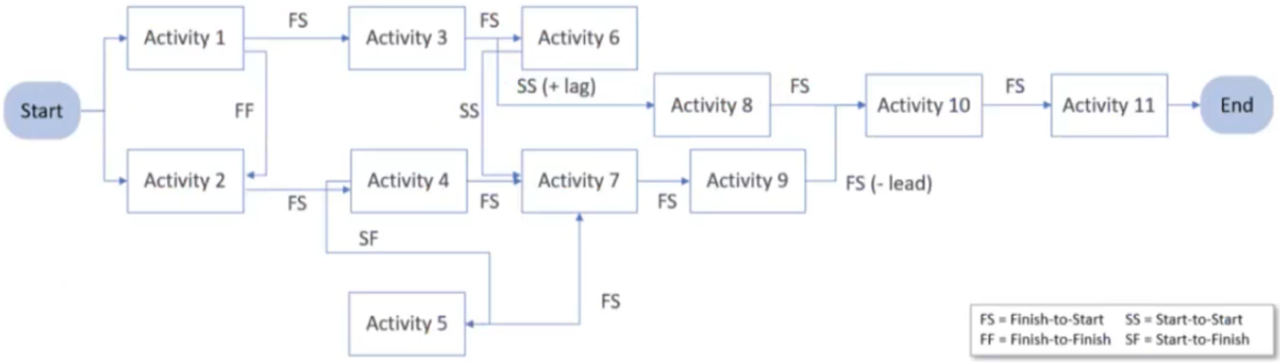
- Changes to tasks on the critical path changes the project finish date.
- A task is critical if it cannot float earlier or later.

The dependencies can be:

- mandatory
- discretionary
- external: outside project's team control (ex: waiting 3rd party component completion before integration)
- internal

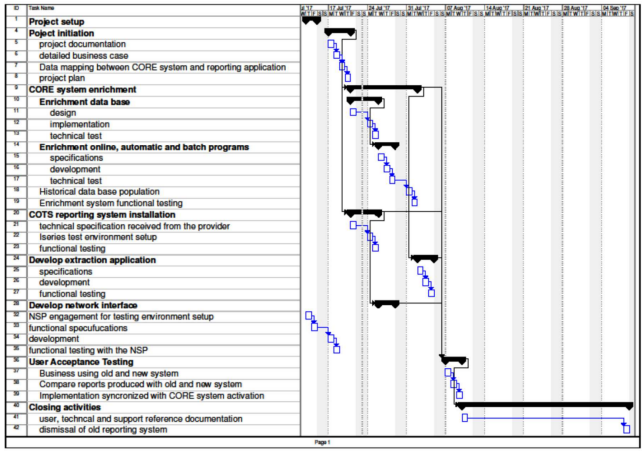
and also:

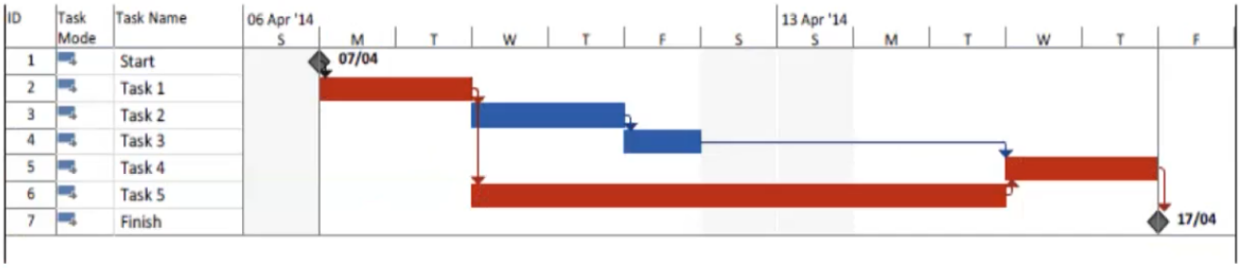
- Flexible: dynamic deadline (as soon as possible ..)
- Partial flexible: bounded deadline (start no earlier than / finish no later than)
- Inflexible: (must occur on specific time interval)



3.6.1 Gantt chart

It can be detailed and automatically generated:





{width=75%}

Some critical concepts:

- The **critical path** tasks in a Gantt chart are the tasks that determine the duration of the project as they cannot be delayed without delaying the entire project. These tasks are on the main branch starting from the initial point and not the parallel tasks. Note that they are also referred to as dependent tasks, as they depend on the completion of previous tasks to start.
- **Fast-tracking** consists in pushing tasks to occur faster than they would: no cost increase but higher risk.
- **Crashing** is the practice to reduce the time dedicated to the tasks on the critical path allocating additional resources to work in parallel with the existing resources working on the task (increasing costs). Crashing is typically used on critical path tasks that take multiple days to complete, by doing multiple activities in parallel. Tasks that have lower costs are usually prioritized for crashing.

3.7 Function points

There are two main techniques for estimating the cost of a software development project:

- experience-based techniques, which are based on the cost of past projects
- algorithmic cost modeling, which involves using mathematical functions to estimate the cost of a project.

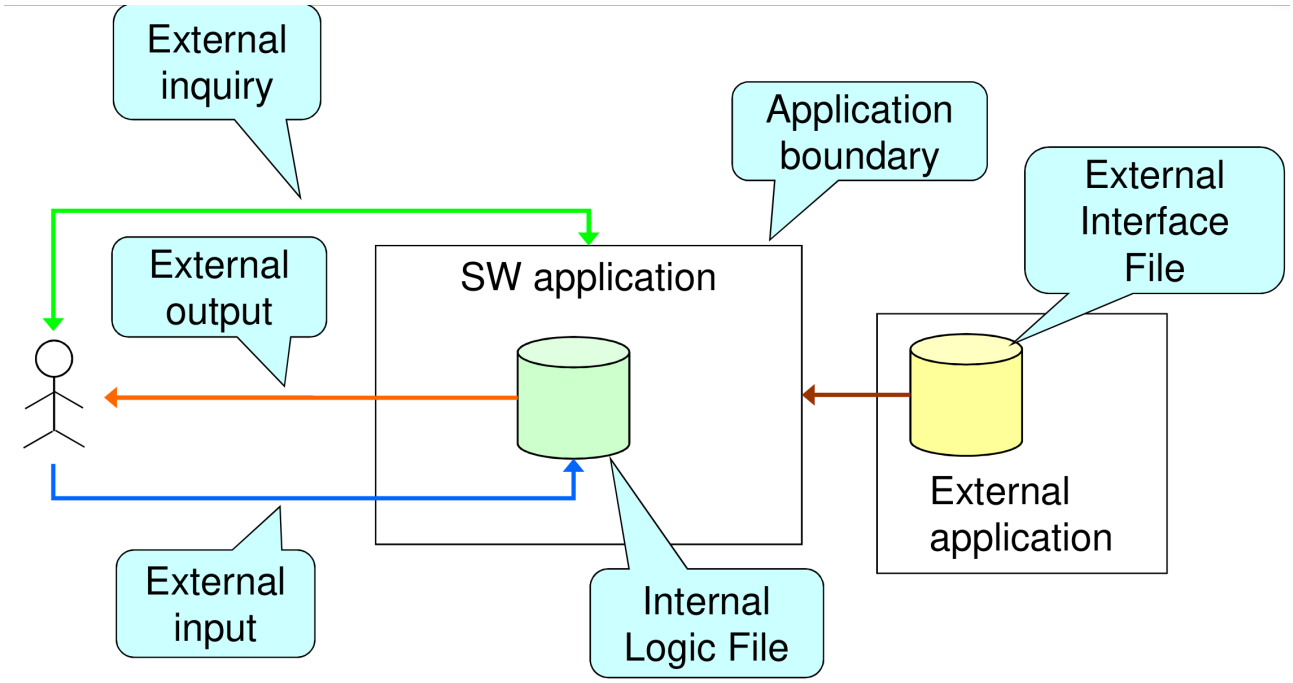
Function points and **COCOMO II** are examples of specific algorithmic techniques that can be used for estimating the cost of a project.

In particular, function points are a measure of the functionality provided by a software system. They are calculated by identifying the number and complexity of five types of software components:

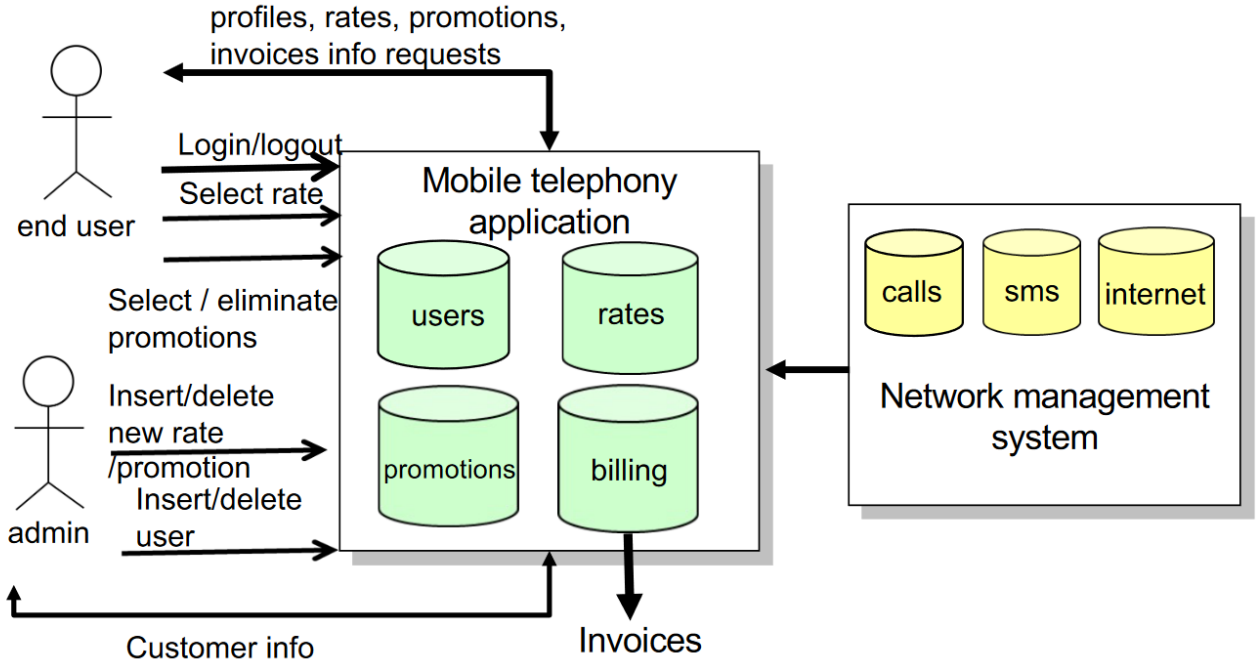
Function Point	wtf is?	Example	Complexity
External Interface Files (EIF)	interface necessary to RECEIVE from external actors	data **received** from an external server	Simple if it's just a simple stream of data, complex if more sophisticated ways
Internal Logic File (ELF)	data/classes/variables used by the app	all the classes to represent the actors involved	the number of fields for each entity and the numbers of entity
External Output	elementary operation that generates data for the external environment (not only the main user but any actor)	return the money change or show the current credit. The actors could be a customer or another company	Depends on number and complexity of outputs and the use of external services
External Input	elementary operation to elaborate data coming from the external environment (not only the user)	The user insert money or spinge a button	use of external services increases the complexity
External Inquiry	elementary operation that involves input and output without significant elaboration	Typical example is a query	it depends on the number of results and the complex of the objects involved

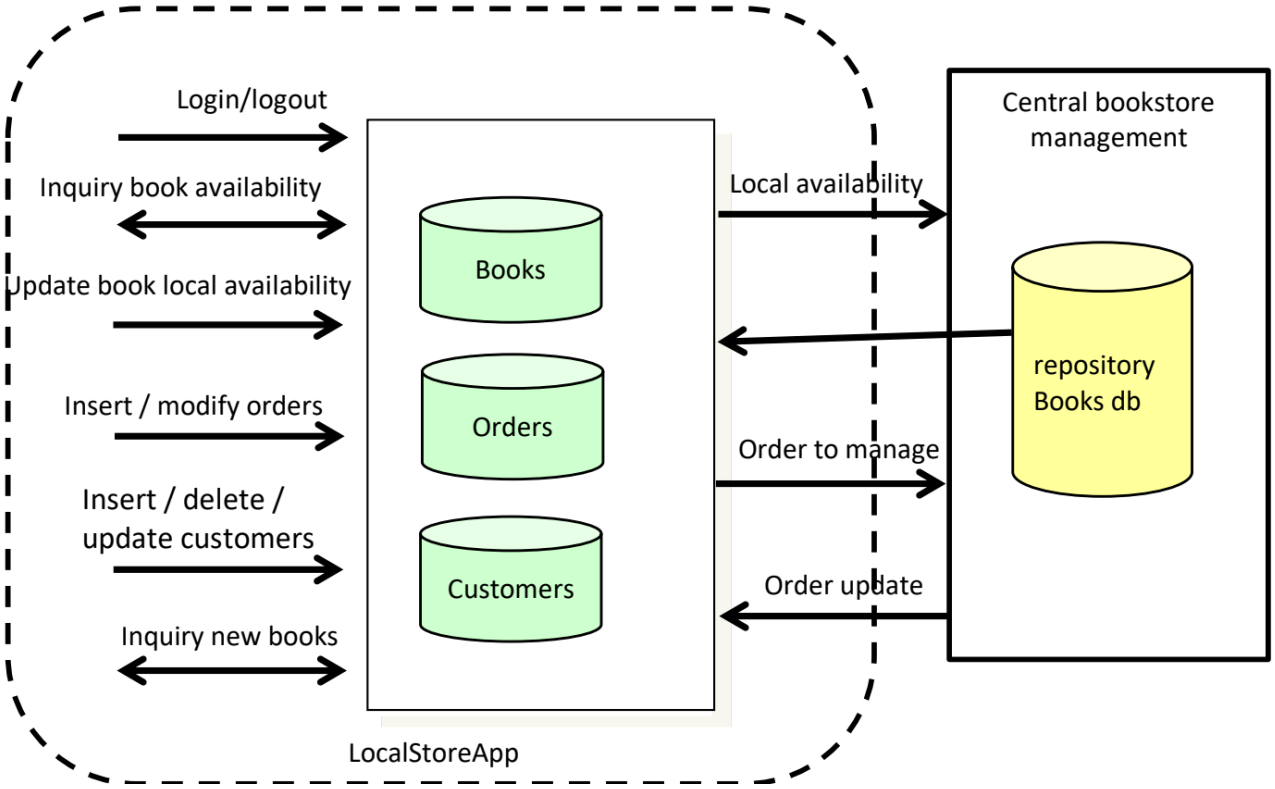
Internal Logic Files examples with segnature: - Topics(topicID, description) - Events(eventID,topicID,body) - Subscriptions(subscriptionID, topicID,componentEndPoint)

Function points are used to estimate the size of a software system, and can be used in conjunction with productivity rates to estimate the time and resources required to develop the system. A generic visualization of Functions Points could be this:



Other examples:





3.8 EVA monitoring methodology

Monitoring involves collecting data on the current status of a project to identify any deviations from the initial plan. **EVA** (Earned Value Analysis) is a project management technique that allows you to track the financial progress of a project by comparing the actual work completed to the work that was planned and budgeted.

Main stuff:

- **BAC**: the budget at completion, which is the total budget for the project.
- **PV**: the planned value, which is the budgeted cost of work planned at a specific time of the project.
- **EV**: the earned value, which is the budgeted cost of work performed at a specific time of the project.
- **AC**: the actual cost, which is the actual cost for the completed work.
- **SV**: the schedule variance is calculated as $EV - PV$.
  - If $SV < 0$ means that the project is **running behind schedule** because the value produced is less than the value planned.
- **SPI**: the schedule performance index is calculated as $\frac{EV}{PV}$.
- **CV**: The cost variance is calculated as $EV - AC$.
  - If $CV < 0$ means that the project is **running over budget** because the value produced is less than the cost.
- **CPI**: The cost performance index is calculated as $\frac{EV}{AC}$.
- **EAC** : the expected cost at completion can be computed in three different ways, based on three different assumptions that can be used:
  - $EAC = \frac{BAC}{CPI}$ continuing to spend at the actual/same rate.
  - $EAC = AC + (BAC - EV)$ continuing to spend at the original rate.
  - $EAC = \frac{(AC+BAC-EV)}{CPI*SPI}$ both the CPI and SPI influencing the remaining work.