Software Engineering Notes

Giuliano Abruzzo

September 15, 2019

Contents

1 Cap 1: Software Process Standardization

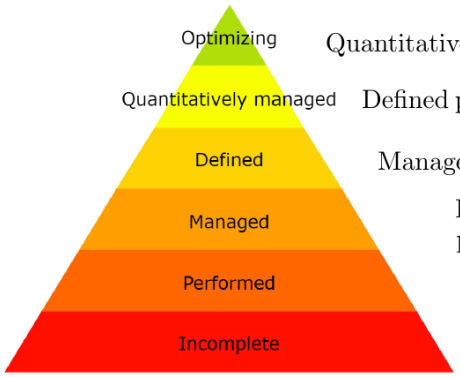
A **process model** is a *structured collection of practices* that describe the characteristics of effective *processes*. A *process model* is used:

- To set process *measurable objectives* and *priorities*;
- To ensure *stable, capable*, and *mature processes*;
- As a *guide* for improvement of *projects* and *organizational processes*;
- To diagnose/certify the *state* of an *organization's current practices*;

Capability Maturity Model Integration or **CMMI** is a *process improvement approach* that provides organizations with the *essential elements* of *effective processes*. It is a *constellation* of components which generate *process model* called **CMMI best practices**. These *constellations* are:

- **CMMI-DEV**: which provides guidance for *managing, measuring* and *monitoring development processes*;
- **CMMI-SVC**: which provides guidance for *delivering services* within organizations and to external customers;
- **CMMI-ACQ**: which provides guidance for *applying CMMI best practices* in an acquiring organization;

There are several **capability levels** of a *process*, and they can be described as a *pyramid*:



Quantitatively managed process improved based on variations;

Defined process controlled using statistical and other quantitative technique;

Managed process tailored from the organization's set of standard process;

Performed process with the basic infrastructure to support the process, according to policy;

Satisfies the specific goals of the process area;

One or more of the specific goals of the process area are not satisfied and no generic goals exist;

Process management, **Project management**, **Engineering** and **Support** are the 4 groups of *process areas* related to **CMMI framework**.

ISO 12207 *standard* for *software lifecycle processes* defines and structures all *activities* involved in the **Software Development Process**. Is based on two principles:

- **Modularity**: *processes* with *minimum coupling* and *maximum cohesion*;
- **Responsibility**: establish *responsibility* for each *process*;

ISO 9000 *standard* addresses *quality management*, used by any organization which *design, develops, manages* any *product* or provides any form of *service*, it achieves customer satisfaction and continual improvement. The *ISO 9000* fundamental **building blocks** are:

- **Quality management system**;
- **Management responsibility**;
- **Resource management**;
- **Product/Service realization**;
- **Measurement, analysis, and improvement**;

2 Cap 2: Software Processes

Since *software products* are not *tangible*, we have to use **special methods** in order to manage them, especially during their *development*. The *monitoring* is based on the explicit definition of *activities* to be performed and **documents** to be produced. *Documents* allows to *monitor* the *evolution* of the *process* so we can evaluate its *quality*. There are several **software process models**:

- **Waterfall model**:
  - It provides several stages: *analysis, design, implementation and unit test, integration and system test, maintenance*. This can be a **problem**, cause discovering *errors* at the *end* is not good, the *end user* has not a clear idea about the *project* until it's finished, and *changes* are difficult to apply after the *process* is started;
- **Prototypal model**:
  - In which the *customer interaction* is *contiguous* and we present to it every time the *prototype*. The goal is to collect and disambiguate requirement involving the *customer*. The *prototype* are not full working, but since we work with the *customer*, he has an idea about the *project*;
- **Incremental model**:
  - Similar to the previous one but all the *intermediate version* of the *product* are fully working ones, so it allows a more accurate design.

These two *models* were part of the **incremental development**, where for each versions a part of the required functionality is delivered. With these approach, *system failures* are less probable and the *customer* soon has an idea about the *project* and it is also good for *changes*, obviously;

- **Formal models**:
  - They comprise every *method* where there are *formalism* for all step of the process. They are highly precise, but difficult to understand by a human being;

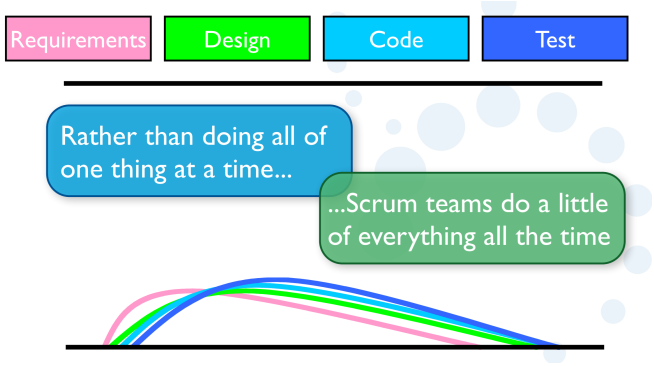
- **Spiral model**:
  - There is no sequence for *activities* to be done, but all is represented as a *spiral* where each loop is a *phase*. There are no fixed *stages* like in previous *models*, every *loop* is defined depending on what is required. We have different *sectors* like: *objective setting, risk assessment and reduction, development and validation, planning;*

The main involved **activities** in *software processes* are:

- **Software specification**: the *process* of establishing what *services* are required, and the *project's constructs*. It's a requirement engineering process;
- **Software design and implementation**: the *process* of converting the *system specification* to an *executable system*;
- **Software validation**: the *process* in which is intended to show that the *system* conforms to its *specification* and meet *customer requirements*, *system testing* belongs to this *activity*;
- **Software evaluation**: the *process* in which new *features* and *changes* are implemented;

3 SCRUM

Scrum is an *agile process* that allows us to focus on delivering the highest *business values* in the shortest time. The *business* sets the priorities and teams self-organize to determine the best way to deliver high priority *features*. *Inspections* is done every two weeks (to one month) so anyone can see *real working software* and decide to release it as is or continue to develop it. The procedure is divided in **sprints** of *software development* (2-4 weeks). *Requirements* of the *project* are captured in a **product backlog**.



This sequence is repeated in each **sprint**, and no *changes* are made during a *sprint*, so there are no problems in *development process*. *Sprint* duration depends in how much time is needed to complete a *feature*, indeed for every *sprint* there is a potentially *shippable product* increment.

The **SCRUM framework** is composed by:

- **Roles**:
  - **Product Owner**: defines *product features*, decide the *release date* and content and rejects/accept the work result;
  - **Scrum Master**: represent *management* to the *project*, ensures that the *team* is fully functional and productive, enable close cooperation across all roles and functions and shields the *team* from *external interference*;
  - **Team**: typically composed by 5-9 people, it is *cross-functional* and *members* should be full-time. Membership should change only between *sprints*;
- **Ceremonies**:
  - **Sprint planning**: the phase where *team* selects items from the *product backlog* to commit completing, so *sprint backlog* is created, giving an estimated time for each *task* and this is done collaboratively;
  - **Daily Scrum**: where the *team* talks about things to do and so on;
  - **Sprint review**: the team presents what it accomplished during the *sprint*;
  - **Sprint retrospective**: the *team* take a look at what is and is not working;
- **Artifacts**:
  - **Product Backlog**: contains *requirement* and a list of all desired work on the *project*. It's ideally expressed such that each *item* has value to the *users* or *customers* of the *product*;
  - **Sprint Backlog**: is isolated to a single *sprint* and contains *goals* for it. It can be manipulated by every *member* of the *team*;

4 User Stories

User stories are sentence in everyday language which represent *backlog items*. They fit on 3" x 5" *index card* and have a fixed *structure* like:

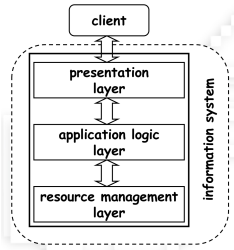
As a *USER*, **So that** *I CAN ACHIEVE GOAL*, **I want to** *DO SOME TASK*.

They can be formulated as *acceptance test* before code is written. As they are *product backlog items*, they have a **priority**. *User stories* can have an **achievement score** (*points* in integer scale) depending on their *complexity*, and this allows to check *productivity* calculating number of *points/week*. *Trackers* allows attaching *documents* to *User stories*. In order to understand if we have a good *user story* we use the *SMART stories system*, which stands for **S**pecific, **M**easurable, **A**chievable, **R**elevant, **T**imeboxed. Sometimes *User stories* need **UI**, so in this case we need to attach a *document* to the story: pen and paper drawings or **Lo-Fi UI**. **Storyboard** show how *UI changes* are based on *user actions*.

5 Distributed Programming

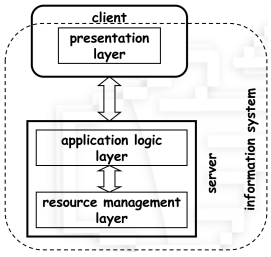
We can have different types of **tier architecture**:

1-Tier Architecture:



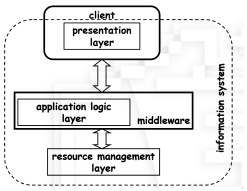
Client is any *user* or *program* that wants to perform an *operation* over the *system*, interacting with the **presentation layer**. The **application layer** represents whats the *system* actually does and it can have several forms: *programs, constraints, business processes*. The **resource manager layer** is the *storage* of the *data* necessary for the *information system* to work. In this design all is centralized and *managing* and *controlling resources* is easier.

2-Tier Architecture:



Client are divided and there is the concept of API at a client level, so there are no client connections/sessions to maintain, as the resource manager interacts with application layer only.

3-Tier Architecture:



Middleware is just a level of *indirection* between *clients* and other *layers* of the system. It introduces an additional layer of *business logic* encompassing all underlying systems. The **N-tier architecture** is just a generalization of this schema.

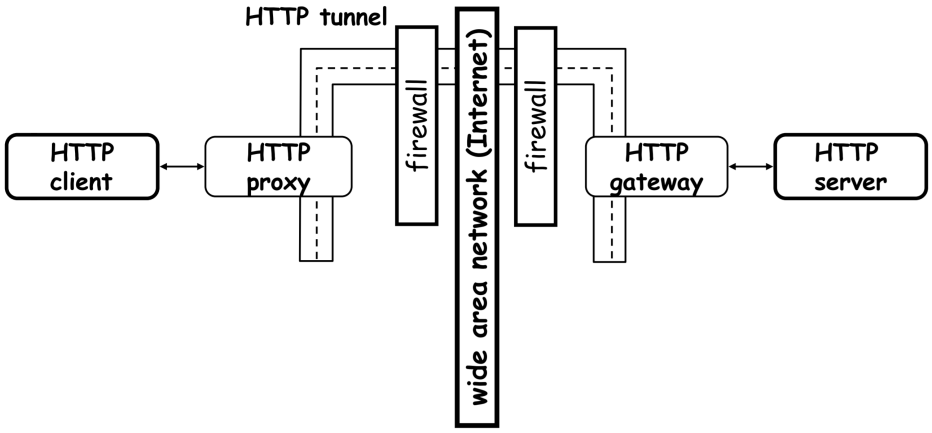
In *Middleware* there is extensive use of **communications**. We know that there are two types of *communication*, *asynchronous* and *synchronous*. *Synchronous* communication requires both the parties to be online, with connection overhead and difficulty to recover from *failures*, in fact *failures* are often handled through *timeouts* and similar technique. To solve this there are two technique:

- **Enhanced Support**: *service replication* and *load balancing* are used in order to prevent the service to be unavailable;
- **Asynchronous Interaction**: with the use of technologies like *non-blocking invocation* or *persistent queues*;

As **Middleware** is a *technique* to hide some complex mechanisms of *information system*, *programming abstraction* can be considered an example. As the *programming abstraction* reach higher and higher levels, the *underlying infrastructure* has to grow accordingly, and it is also intended to support *additional functionality* that makes *development, maintenance* and *monitoring* easier and less costly.

Remote Procedure Calls or **RPC** is a *point-to-point protocol* that supports the interaction between two *entities*. **RPC calls** are treated as independent ones even if the are more *entries* interacting each other. The programmer uses an **RPC system** instead of implementing a different one for each *distributed system*. An *RPC system* hides low-level operations, provides an interface definition language to describe the services, and generates all the code necessary to made the RPCs work. **Publish/Subscribe** is a type of *many-to-many communication* where the participants are divided in two groups, *publisher* and *subscriber* and the *communication* takes place through a *central entry*, that is often a *queue* and *infos* can be *topic-based* or *content-based.* The notification to a *subscriber* can be done in two ways: **push** where *subscribers* are invoked in callback, or **pull** where *subscribers* poll the central entry when they need messages.

This is an example of **message-oriented Middleware** on internet, where *proxy* and *firewalls* are a way to have *Middleware* on the internet and they work as in this schema:



6 Web Services

A **Web Service** is a *programmatically available application logic* exposed over the *internet*, such that *clients* can access *network-available services* instead of invoking available *applications* to accomplish some *task*. **Web services** perform *encapsulated business functions* such as:

- A *self-contained business task*;
- A *full-fledged business process*;
- An *application*;
- A *service-enabled resource*;

Services offered can change depending on *pricing*, in fact there is also a *billing model* which can be different, depending on the *service type* offered *services* can mixed/merged to create *extensions*

and so on. This *application logic* appeared at the beginning with **ASP** or **Application Service Providers**, an *ASP* "rents" applications to subscribers. The **ASP** model introduced the concept of **software-as-service** but suffered from several limitations like inability to *integrate, customize* and *develop highly interactive* applications, instead the new architecture of *web services* makes *communications* and *access* to *applications* on the internet easier. *Services* can be used within an enterprise to accomplish some *tasks*, but also between enterprises. **Services** can mainly be of two types:

- **Informational services**: which provide *access* to *content* interacting with an *end-user* by *request/response sequences*;
- **Complex services**: which invoke the assembly and invocation of many *pre-existing services* in a unique result;

Services can be **functional** or **non-functional**, *functional* if the *service* describe his *overall behavior*, *non-functional* if it describe only some *targets service quality attributes*. *Services* can be **stateless** and **stateful**, *stateless* if they can be invoked repeatedly without maintaining a *state* or a *context*, *stateful* if they require their *context* to be preserved from one invocation to the next.

The **service model** allows for a clear distinction to be made between:

- **Service providers**: who provides the *service*;
- **Service clients**: who uses the *service*;
- **Service registry**: the *directory* where *service descriptions* are published and searched;

SOAP is the *standard messaging protocol* used by *web services*. *SOAP's application* is *inter application communication* through *XML objects* and using *HTTP* as a means for transport. *SOAP* supports two types of *communication* styles:

- **RPC**, where *clients* express their *request* as a method call with a set of arguments and which returns a response containing a *return value*;
- **Document-style**, where there is an *XML document fragment* in the body;

As we said before, **HTTP methods** are used for their *request/reply communication*.

A **service description** is always needed cause it allows the *client* to know how to use the *service* properly, in particular how to handle the precise *XML structure* of the *web service*, such that the *communication* can happen correctly. This *description* is done in a **WSDL file** which describes *service interfaces*, and represent a **contract** between *service requester* and *service provider*. It can be separated into distinct sections:

- **Service-interface definition**: that describes the *Web Service structure*;
- **Service implementation part**: that binds the *abstract interface* to a concrete *network address, protocol* and *data structures*;

These two contains sufficient information in order to *invoke* and *interact* with the *Web Service*. *WDSL interfaces* support four types of *message exchange* patterns:

- **One-way messaging**: from sender to receiver;
- **Request/Response messaging**: from sender to receiver and reverse;
- **Notification messaging**: from receiver to sender;
- **Solicit/Response messaging**: from sender to receiver and reverse;

UDDI or **Universal Description, Discovery and Integration** is a registry for *Web Service* description and discovery, which enables a business to **describe** its business and its services, to **discover** other business that offer desired services, and to **integrate** with these business. So *UDDI* is a usefully *registry* that a *requester* can use in order to find the *service* needed, and the *client* doesn't need to directly contact the *provider* of the service.

Services can also be **mashed-up** in order to obtain a *web application* that combines *data* from **more sources** in a **single integrated tool**, like *Google Maps*, which merges cartographic data and data on traffic, real estate data, weather.

REST refers to simple application interface transmitting data over *HTTP* without additional layers as *SOAP*. Resources are simply organized through *URIs* and they are directly accessible through operations like *GET, POST, DELETE*. The most important principles of *REST* are: **addressability** (*resources on URLs*), **uniform interface** (HTTP GET, PUT, ...), **stateless interactions**, **self-describing messages** and **hypermedia**.

E-service is the provision of a *service* via the Internet, instead **Web Service** is a *software component* available on the Web, to be invoked by a *client app/component*.

7 Microservices

Microservices are a *software development technique* (variant of *service-oriented architecture*), where an *application* is structured as a collection of **loosely coupled services**, so this means the various components of the system know nothing about the other ones. Services are **fine-grained**, so an *application* is decomposed in several *smaller services* in order to improve the **modularity** of the *application*, this means that *smaller services* can be used almost **independently**. This makes the *application* easier to *understand, develop, test* and *recover* in case of *failures*. The development is *parallelized*, every *service* is deployed and scaled independently. *Microservices* solves problems like *scaling difficulty, long development cycles* and *failure recovery difficulty*. Services in **MSA**, or **Microservices architecture**, are often capable to cooperate and communicate each other via Internet in order to achieve a specif goal. *Services* can be implemented using *different programming language, database* depending on what fits best. *Microservices* only communicate through *public API* of the others, such that each of the **primitive components** can hide its data and in order to secure the systems every team can decide to implement security measures.

8 Measures and Statistics

Measures and statistics are used in *software development processes* to validate effects of strategies applied to it for improving their quality. We consider five **measurement scales**:

- **Nominal Scale**: which classifies persons or objects into two or more categories, it use a *pre-defined non ordered set* of distinct values, with operators: $=, !=$, this scale is used to check the frequency by which certain measures fall into certain categories;
- **Ordinal Scale**: which is a rank on how much an item has a characteristic, with operators: $=, !=, >, <$;
- **Interval Scale**: which allows us to rank the order of the items that are measured and to quantify and compare the sizes of differences between them, with operators: $=, !=, >, <, +, -$, an example is the temperature in C° or F°;
- **Ratio Scale**: like the previous one, but with a fixed zero point, not arbitrary, with operators: same as before plus $..., *, /$
- **Absolute Scale**: it is a ratio scale ranging on non negative integers;

The choose of a *scale* depends on the *attribute* to be measured. There are several types of **measures**:

- **Ratio**: a division between two values of two different domains, with values $+/-1$;
- **Proportion**: a division between two values where the dividend contributes to the divisor like: $\frac{a}{a+b}$
- **Percentage**: a proportion or fraction normalizing the divisor to 100;
- **Rate**: a value associated with the dynamics of a phenomenon, like the change of a quantity with respect to another quantity.

In *software engineering*, we can cite as an example the rate of $Errors/KLOC$ where **KLOC** is "*thousands of line of code*" and we use it to determine what's the **quality** of a *software product* when testing it at the end of his *lifecycle*. A *measure* is **reliable** if it gives always or almost always the same values when *measuring* the same thing under the same condition (the less is the **variance** the more is *reliable*), and is valid if it's a correct way of measuring such a thing.

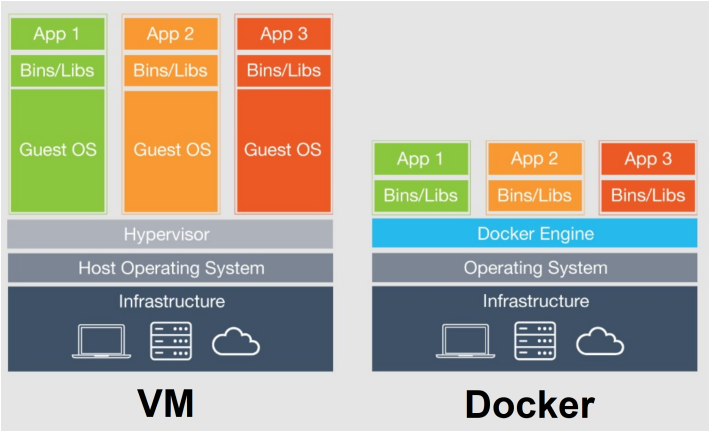
$$M = T + E_T \qquad E_T = E_{systematic} + E_{random}$$

Where M is the *measure*, T is the *true values*, and E_T is the *total error*, and $E_{systematic}$ influences *validity* and E_{random} influences *reliability*.

In **inferential statistics** we have random samples so we calculate the probability that, for example, under a *normal distribution*, the *mean* of a population M is within an interval centered on the *mean* of a sample of N elements of such a population.

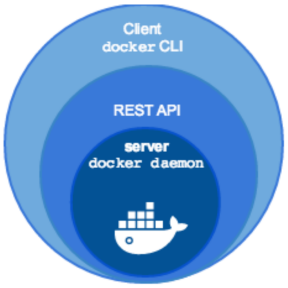
9 Docker

In a **VM** we find everything necessary to run an *OS*, instead in a **Container** (*Docker*) there is the *application* with other *dependencies*, but with a *kernel* shared with other *containers*. They are not tied to a specific *infrastructure*, and run an *isolated process* in userspace on the *host OS*. So, **Docker** *containers* running on a single machine all share the same *OS kernel*, so they start instantly and make more efficient use of RAM.



Docker containers wrap up a piece of software in a complete *file system* that contains everything it needs to run, anything you can install on a server. When we have to use *Docker* for an *application*, it is better to divide the *app functionality* in *individual containers* like *microservices*. The advantages of using *Docker* are:

- **Lightweight** resource usage, as we have a *shared Kernel*;
- **Portability** of *Docker containers*;
- **Predictability** of interactions between *Docker containers* and *underlying system*, as the interfaces are standardized and interaction predictable.



The **CLI** or *command line interface* is used to handle every kind of usage that *Docker* can handle. From it, you can wrote and run several command to interact with the *server Docker daemon*.

Compose is the the tool which allows to organize several *containers* in a way that they work together in the same *app* (*microservices*). In *VMs* we need several *VMs* to do that.

10 Function Points

A **function point** is a unit of measurement used to express the amount of business functionality that an info system provides to a user. This unit of measurement is used to compute a functional size measurement, or FSM, of a software. The *functional user requirements* are divided into five categories:

- **Interal Logical File (IFL)**:
  - Is a user-identifiable group of logically related data or control information maintained within the boundary of the application;
- **External Interface File (EIF)**:
  - Is a user-identifiable group of logically related data maintained within boundary of another application. This means that an EIF for an application must be in an ILF of another application;
- **External Input (EI)**:
  - Is an elementary process that processes data that comes from outside the application boundary;
- **External Output (EO)**:
  - Is an elementary process that sends data outside the application boundary;
- **External Inequity (EQ)**
  - Is an elementary process that sends data outside the application boundary, but the intent of EQ is to present information to a user through the retrieval of data from an IFL of EIF. Differently from EO, the processing logic contains no math formulas and does not create derived data.

Each of the previous functionalities has three **weights**, *low, medium* and *high*, that are used to *correct* and to give a value to each of them, which ranges from 0 to 5. *IFL* and *EIF* are called **data functionalities** and their complexity is associated with the number of RET/DET where RET is the *record element type* and DET is *data element type*. EI, EO, EQ are also called **transactions** and FTR and DET are their *components*.

11 CoCoMo

The **Constructive Cost Model** or **CoCoMo** is a procedural software *cost estimation* based on LOCs (lines of codes). It is often used for predicting the various parameters of a project, like *size, effort, cost, time* and *quality*. The key parameters which define the quality of any *software products* are:

- **Effort**: amount of labor that will be required to complete a task, measured in $persons/month$;
- **Schedule**: amount of time required for the completion of the job, measured in weeks or months;

Effort is measured and estimated through the formula: $E = a \times (KLOC)^b$ where the two parameters depends on the particular type of project:

- **Organic**: a software project with team size small, the problem is well understand and has been solved in the past, and also the team members have a nominal experience regarding the problem;
- **Semi-detached**: a software project where the vital characteristics like team size, experience, knowledge is between organic and embedded;
- **Embedded**: a software project that requires the highest level of complexity, creativity and experience from the team member, so a large team;

As the *complexity* grows, *parameters* become higher. The model explained is the **basic model**, in fact it does not take factors like *reliability* and *experience* into account. No *system effort* can be evaluated only through two parameters and *LOCs*, indeed there is an **intermediate model** used for better *accuracy* and *correctness*, where these messing parameters are taken into account. These parameters/factors are called **cost drivers** and they are classified in:

- **Product attributes**: like *complexity*, *reliability* and *team size*;
- **Hardware attributes**: like *memory constrains* and so on;
- **Personal attributes**: like *experience* of team members on several tasks;
- **Project attributes**: like use of *software tools* and *software engineering techniques*;

The **effort formula** becomes: $E = (a \times (KLOC)^b) \times EAF$ where EAF is **Effort Adjustment Factor** which ranges from very low to extra high through a number. In the **detailed CoCoMo model** we have different *effort multipliers* for each cost driven attribute, in fact here we have a project division into several *modules* so that we can apply *CoCoMo* various times. These are:

- Planning and requirements;
- System design;
- Detailed design;
- Module code and test;
- Integration and test;

- Cost constructive model;

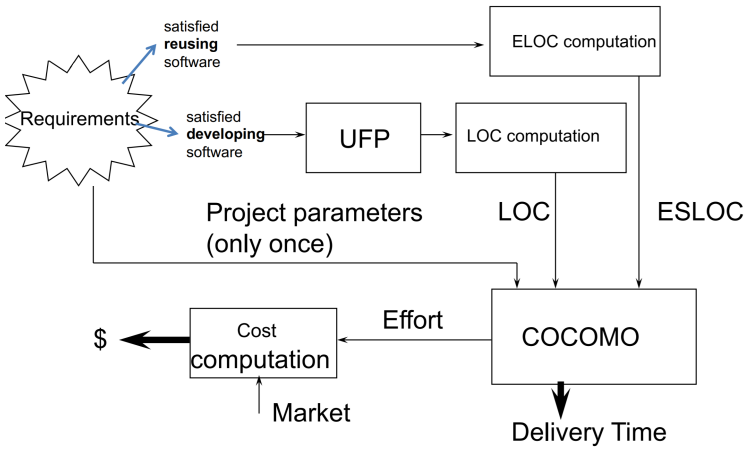
CoCoMo 2 is the revised version of the CoCoMo and consists of three sub-modules:

- **End User Programming**: where application generators are used to write code;
- **Intermediate Sector**: where we have app generators and composition aids, application composition sector and system integration;
- **Infrastructure Sector**: where infrastructure for the software development cycle is provided, like OS, database, ...;

The whole process of **CoCoMo 2** is divided into 3 stages:

- Prototyping estimation;
- Early design project state estimation;
- Post architecture stage estimation;

CoCoMo 2 model also provides a way to reuse some *SLOC*, *source code*, to adapt it to another *project* in a way such that we have this schema:



12 Exam Questions

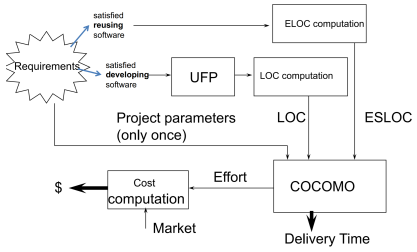
12.1 Function Point and CoCoMo question

Function Point is a technique used in order to valuate size of *software products* and *productivity* of development team. FP is an objective technique where the goal is to weight functionalities of the system in terms of *data* and *relevant processes* for guests. There are 5 main *functionalities*:

- **ILF**, internal file;
- **EIF**, external file;
- **EI**, input activities;
- **EO**, output activities;
- **EQ**, query activities;

For each of them will be computed a **weight of complexity** based on *DET, RET* and *FTR*. Knowing the complexity of each functionalities we sum every FP of them computing UFP. In order to compute AFP we just need to multiply UFP with *adjustment factor* that is based on 14 characteristics of the entire system each of them related with a integer number from 0 to 5. **CoCoMo** or **Constructive Cost Model** is a *procedural software cost estimation* based on LOCs, lines of codes, used for predicting parameters of a project, like size, effort, cost, time and quality. Function point are used in order to measure the functional size of a software, instead CoCoMo II use FP size count as primary input in order to estimate effort and schedule for a software project. CoCoMo can be used in three different version:

- **Basic model**, that in order to estimate the effort doesn't take factors like reliability and experience, the formula is: $E = a \times (KLOC)^b$ where the two parameters depends on the particular type of project;
- **Intermediate model**, that in order to estimate the effort use parameters called cost drivers, and the formula becomes: $E = (a \times (KLOC)^b) \times EAF$ where EAF is **Effort Adjustment Factor**;
- **Detailed model**, in which we have different effort multipliers for each cost driven attribute, where the formula is the same of the intermediate but it's repeated various time;



CoCoMo 2 is the revised version of the *CoCoMo* and consists of three sub-modules: end user programming, intermediate sector and infrastructure sector. The whole process of **CoCoMo 2** is divided into 3 stages: prototyping estimation, early design project state estimation and post architecture stage estimation. **CoCoMo 2 model** also provides a way to reuse some *SLOC*, *source code*, to adapt it to another *project*

12.2 SCRUM question

Scrum is an *agile process* that allows us to focus on delivering the highest *business values* in the shortest time. The procedure is divided in **sprints** of *software development* (2-4 weeks). *Requirements* of the *project* are captured in a **product backlog**. This sequence is repeated in each **sprint**, and no *changes* are made during a *sprint*, so there are no problems in *development process*. The **SCRUM framework** is composed by:

- **Roles**:
  - **Product Owner**: defines *product features*, decide the *release date* and content and rejects/accept the work result;
  - **Scrum Master**: represent *management* to the *project*, ensures that the *team* is fully functional and productive;
  - **Team**: typically composed by 5-9 people and it is *cross-functional*;
- **Ceremonies**:
  - **Sprint planning**: the phase where *sprint backlog* is created, giving an estimated time for each *task*;
  - **Daily Scrum**: where the *team* talks about things to do and so on;
  - **Sprint review**: the team presents what it accomplished during the *sprint*;
  - **Sprint retrospective**: the *team* take a look at what is and is not working;
- **Artifacts**:
  - **Product Backlog**: contains *requirement* and a list of all desired work on the *project*;
  - **Sprint Backlog**: is isolated to a single *sprint* and contains *goals* for it;

If we want to evolve a system of e-commerce in 6 month and every sprint lasts 4 weeks, so we could have at least 6 sprints:

- 1° sprint: as a guest, i want to buy items from the system;
- 2° sprint: as a guest, i want to sell stuff;
- 3° sprint: as a guest, i want to search item through categories;
- 4° sprint: as a logged user, i want to buy items with credit cards;
- 5° sprint: as a selling user, i want to customize shipping methods;
- 6° sprint: as a buying user, i want to review and give feedback's to every item purchased;

Poi devi creare una tabella di esempi dove sulle righe sprint, sulle colonne le feature, e ogni casella ore rimanenti per completare la feature

12.3 Message Orientated Middleware

A **message oriented middleware** is a model in which we have the following scheme: v With a **publish/subscribe** model we have two types of interaction:

- **Topic-based**: which means that the subscriber looks only for a certain type of info (topic object);
- **Content-based**: which means that the subscriber looks only for some messages, depending on their content (not only the topic);

Subscriber can subscribe to a topic and eventually add filters (*content based*) in an *asynchronous way*, which means that they will be notified of a new message in the queue either by *callbacks* (**push**) or by *explicit request* of subscribers (**pull**) when needed.

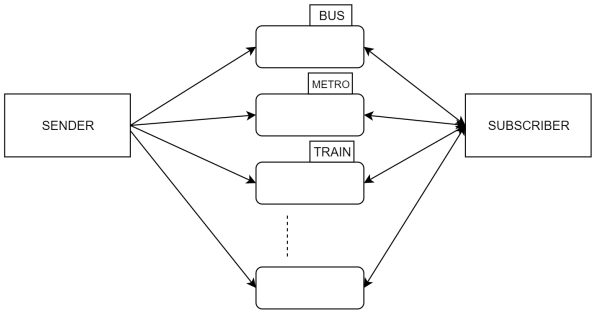


Figure 1: Architecture example

Message

InfoTrasporti {
Topic: *105*
Length: *30*
Body: "*out of order*"}

Sender

Sender {
Send message to queue with '*topic*' == *#autobus*
}

Subscriber

Subscriber {
takeMessage(*int #autobus*) {
if queue.isNotEmpty(*# autobus*)
take message from queue
else print "*nessuna info disponibile*"
}
OnResumeButton() {
#autobus = getLabel()
return takeMessage (*#autobus*)
}
}

Figure 2: Pseudocode example

12.4 REST question

REST refers to simple application interface transmitting data over HTTP without *addictional layer* as *SOAP*. **REST** means **REpresentational State Transfer**, and it consists of a *set of resources* accessible through **URI's** like "*/resources*" and through operations which are for example:

- **GET**: which simply manifests the intention of obtaining a resource;
- **POST**: when we want to add a resource to the server, if it doesn't exist;
- **DELETE**: trivial;
- **PUT**: update a resource that already exists, or it create it;

REST provides stateless interactions between the client and the server, the only thing that the client obtains is a Status response code, like 200 OK, 404 not find, and so on, when requesting a resource. REST addresses data object, contrarily to RPC which addresses software components. **SOAP** was designed before **REST**, and the idea was to ensure that programs build on *different platforms* and *different programming languages* could exchange data in easy way, instead **REST** was designed specifically for working with components as *files* or *objects*. *SOAP* is a **protocol**, instead *REST* is an **architectural style** in which a *web service* is a **RESTful service** if it follows the constrains of being: *client server, stateless, cache-able, layered system and uniform interface*. *REST* can also make us of *SOAP* as underlying protocol for *web services*.

Restful Server

```
ServerRest server = new ServerRest();
server.setAddress("http://localhost:8000");
server.setResourceClass(new ResourceRepository());
server.start();
```

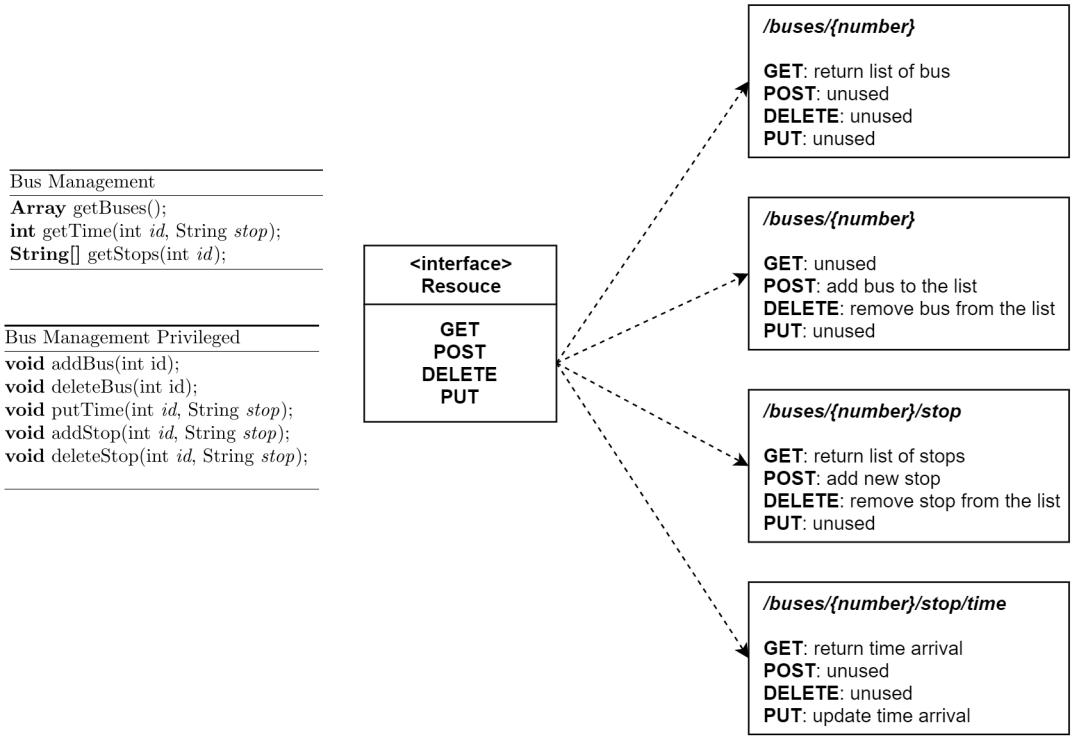
Client

```
Client client = this.getInstance();
cl.connect("http://localhost:8000");
cl.get("/resources");
   ...
```

Resource Repository

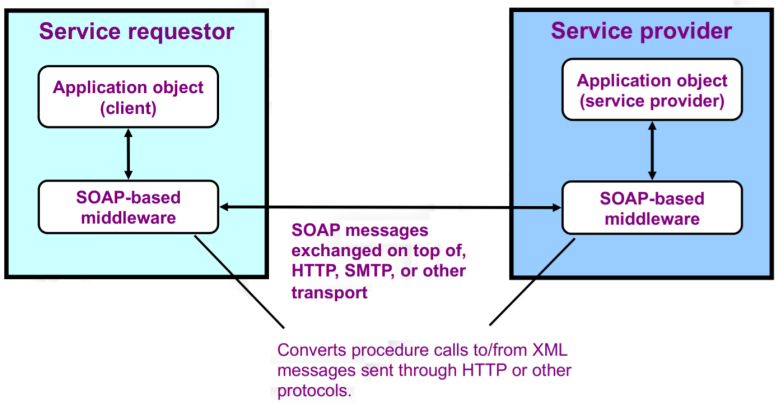
```
Array resArray = new Array();

@GET
@path("/resouces")
function getRes(){
   return resArray;
}
@PUT
@path("/resouces/{number}")
function putRes(number) {
   resArray.update(number);
}
@POST
@path("/resouces/{number}")
function putRes(number) {
   try resArray.add(number);
   catch return "CONFLICT";
}
@DELETE
@path("/resouces/{number}")
function deleteRes(number) {
   resArray.delete(number);
}
```



12.5 SOAP question

SOAP is an *XML-based communication protocol* for exchanging messages between computers regardless of their *operating systems*, *programming environment* or *object model framework*. **XML** is used as an *encoding scheme* for *request* and *response parameters* using *HTTP* as a *transport protocol* only, without exploiting the *HTTP methods* like *REST* does.



SOAP supports two possible *communication style*, **RPC** or **Document orientated**. In the first, *clients* express their *request* as a **method with parameters** and the *response* contains a **return value**. In the second, *SOAP body* is an **XML document fragment** and the *response* can also be *absent*. The receiver scans the response as it stands. **SOAP header**, instead, is used to define *handling methods* for *transport*, and other related *infos/parameters*. As we saw, request are made through specific interfaces: they are described in a **WSDL file**, *Web Service Description Language*, a *machine understandable standard* describing the operations of a *web service*, which specify also the *format* and the *transport protocol* to be used. We can see *WDSL* as a **contract** where we agree with the *provider* on how to interact with it. *WSDL file* describes also where the *web services interfaces* are located on the network. *WDSL interfaces* supports four types of *operations* that represent possible combination of I/O messages: **one-way, request/response, notification** and **solicit/response**.

Server

```
implementator = new FunctionImplementator();
serverAddress = "http://localhost:8000";
endpoint.set(implementator, serverAddress)
server.start();
```

Client

```
endpoint.connect(implementator,serverAddress);
function1().return
function2().return
    ...
```

Foo

```
@XMLTypeAdapter
    //constructors
```

Functions

```
function1();
function2();
function3();
```

Functions Implementator

```
endpointInterface(Functions);
function1 {
    return Foo
}
function2 {
    ...
}
function3 {
    ...
}
```