Software Engineering Notes - Integration

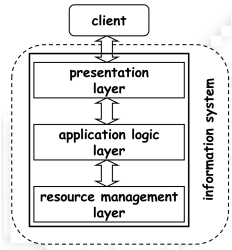
Nicola Di Santo

January 16, 2020

1 Distributed Programming

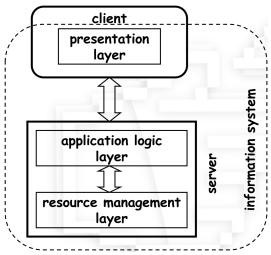
A distributed programming acrchitecture can be *logically* divided into three main layers: Presentation Layer (in charge of showing informations to end user and allowing user to interact with the system), Application Layer (responsible to implement all the business logic functionalities) and Resource Management Layer (to store and handle persistent data). Tiers are generally different from layers since tier represent the phisichal division on different machines of the logical layers. For example, in a client server architecture all the three layers are present but the architecture is 2-tier since only two machines are involved: the client (presentatio layer) the server (application and resource layer). We can have different types of **tier architecture**:

1-Tier Architecture:



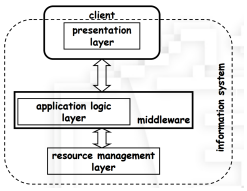
Client is any *user* or *program* that wants to perform an *operation* over the *system*, interacting with the **presentation layer**. The **application layer** represents whats the *system* actually does and it can have several forms: *programs, constraints, business processes*. The **resource manager 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. **Pros**: very performing and cheaper since there is no presentation layer, hence no cost to its developement. **Cons**: monolitic, a very rigid system (mainframe).

2-Tier Architecture:



A simple client-server 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. **Pros**: Portability, Application logic became faster since presentation layer is no more on the same machine of Servers. **Cons**: integration is still difficult as well as adaptation to changes. The number of user supported at same time is fixed and might not meet the requirements.

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 and generally requires comunication over internet. **Pros**: scalability and portability became easy. **Cons**: comunication cost increase and so does latency expecially if over internet.(no more a real problem actually)

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*

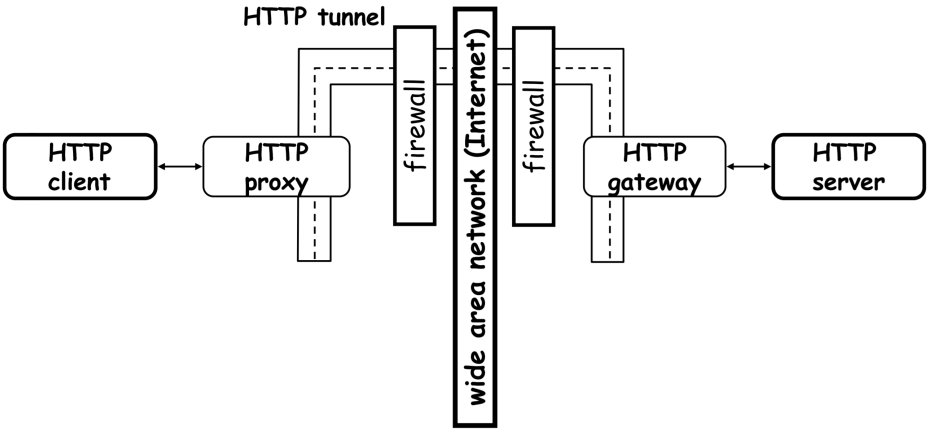
are often handled through *timeouts* and similar technique. To solve this there are two technique:

- **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. From this point people start talking about SaaS.

Remote Procedure Calls or **RPC** is a *point-to-point protocol* that supports the interaction between two *entities*. With the help from an IDL we are able to specify the interface that peers must use to invoche RPCs. Once compiled, the IDL will produce the Stubs. Stubs will be responsible of hiding the distributed environment and then create the RPC abstraction. As infrastructure grows a Binder will be added to provide name and directory service. **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 **Middleware** over internet, to pass through *proxy* and *firewalls* tunneling is used as well as a common internet protocol (HTTP or SMTP) as in this schema, to bypass firewall controls:



MOM (Message Oriented Middleware) Message Oriented Middleware is a concept that involves the passing of data between applications using a communication channel that carries self-contained units of information (messages). In a MOM-based communication environment, messages are usually sent and received asynchronously. Using message-based communications, applications are abstractly decoupled; senders and receivers are never aware of each other. Instead, they send and receive messages to and from the messaging system. It is the responsibility of the messaging system (MOM) to get the messages to their intended destinations.
In a messaging system, an application uses an API to communicate through a messaging client that is provided by the MOM vendor. The messaging client sends and receives messages through a messaging system.
Transactional version of the MOM exist.
A key concept are Message Brokers that allows to decouple message logic from sender and receiver and to but it into middleware. They are are a building block of MOM and not a replacement for it. The primary purpose of a broker is to take incoming messages from applications and perform some action on them. Message brokers can decouple end-points, meet specific non-functional requirements, and facilitate reuse of intermediary functions. For example, a message broker may be used to manage a workload queue or message queue for multiple receivers, providing reliable storage, guaranteed message delivery and perhaps transaction management.

2 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*. The greatest achievement of WS is standardization, that makes very eaasy EAI (cross Enterprises Application Integration). **Web services** perform *encapsulated business functions* such as:

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

Web services are:

- loosely coupled with asynchronous interactions;
- XML based (nowadays also JSON);
- Have the intent of making application available over internet in a easy fashion;

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 have **functional** and **non-functional** properies: the functional service description details the operational characteristics that define the overall behavior of the service; the non-functional description is about service quality attributes like service metering and cost, performance metrics (response time or accuracy), security, authorization, authentication, scalability, availability, etc. . *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 (Simple Object Acces Protocol) 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*.

3 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 (mutually exclusive), 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 - or a quality, with operators: $=, != , >, <$; It is not possible yet to define how much I need to achieve something. It is not stated what are differences among ranks. This might result into non accurate average: suppose i have a degree to measure people health from healty to death if I have 10 death and 10 healty the avg result into 3-medium health people.
- **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*. If we assume possible only E_{random}, and we assume that it is zero on avg, then we can say that repeating the experiment n times with n that goes to infiity and taking the avg of the measures will resulti into $M = avg(M_i) = T, iffn- > inf.$

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.

ANOVA (ANalisys of VAriance): ANOVA allows to analyze two or more samples comparing the internal variability within the groups (Var_W) with the variability between the groups (Var_B). The null hypothesis assumes that all groups have the same distribution, and that any observed difference in the samples is casual The idea is that if $Var_W >> Var_B$ then the observed difference is caused by the internal variability.
ANOVA is a general technique that can be used to test the hypothesis that the means among two or more groups are equal, under the assumption that the sampled populations are normally distributed. As example, the hypotheses are the followings:

- $H_0 : \mu_1 = \mu_2 ... = \mu_I$.
- H_1: at least two among the means are different.

Once the degrees of freedom are known (for both numerator and denominator) it is possible to evaluate the probability (p-value) associated with the values of F (Fisher Test). This test tells us whether to:

- accept $H_0 : p > a$.
- reject $H_0 : p <= a(F >= F - crit)$.

Most often the attempt of demonstrating an hypothesis substantiates in searching a relation between two variables: if we change A then B changes (following a certain rule).

4 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 and it is also used to estimate the productivity of dev teams. This unit of measurement is used to compute a functional size measurement, or FSM, of a software. The *functionalities* 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 (i.e. all the files internal to 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; (i.e. all the data coming from another application)
- **External Input (EI)**:
  - Is an elementary process that processes data that comes from outside the application boundary; (elementary input operation)
- **External Output (EO)**:
  - Is an elementary process that sends data outside the application boundary; (elementary output operation)
- **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. (elementary query/interrogation operation)

Each of the previous functionalities has three **weights**, *low, medium* and *high*, that are used to *score* the effort of a functionality. The resulting score can be adjustet using 14 indicators, resulting into the *AFP* (Adjusted FP).
Pros

- Widely used and accepted (standards, active organizations)
- Certified personnel available
- Objective calculation
- UFP independent of technology
- Can be used early in development process
- Equally accurate as SLOC

Cons

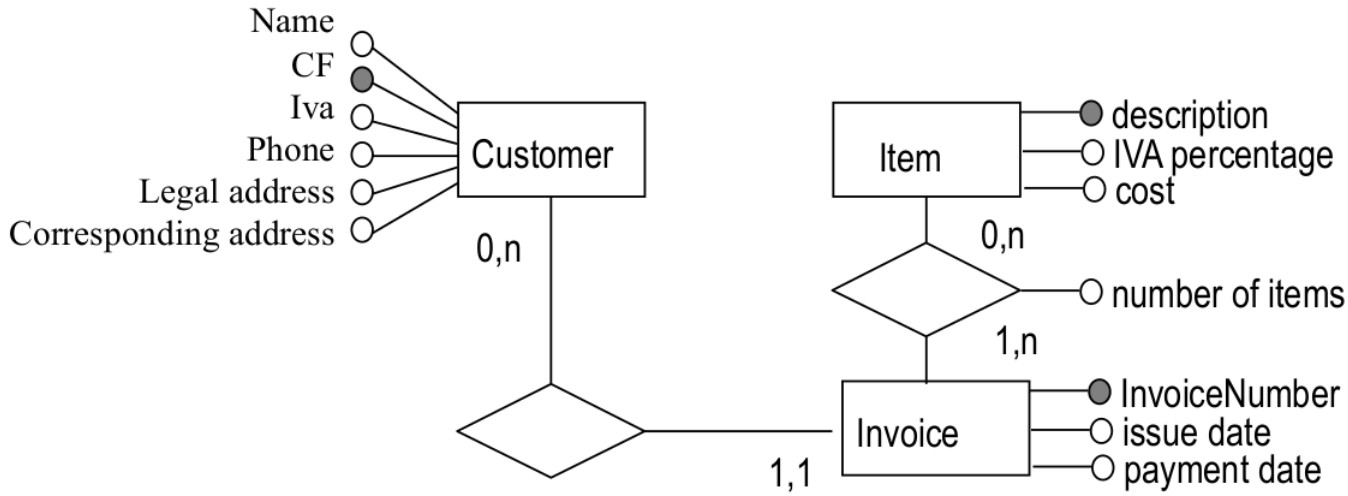
- Semantic difficulty - "legacy" terminology difficult for teaching, and FP are in themselves hard to grasp and compare
- Incompleteness – internal functionality? Stored data size vs. complex processing?
- Lack of automatic count
- Different versions

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*.

Each transactional operation can be classified as simple, medium or complex according to some given complexity tables. Those tables consider

- DET (Data Element Type) a non repeated field, recognizable by the user.
- FTR (File type referrenced): an ILF read or mantained by the app or an EIF read with a transactional op.
- RET (Record Element Type), a subgroup of element from an ILF/EIF

After giving a score to each element we can compute the UFP (Unadjusted FP) that gives an indication on the size og the system in terms of functionalities. After that we can (or can not) compute the AFP with the given formulas and parameters.
NB: functional (requirement) analisys is required to determine FP.
Example FP:



Assume we need to develop a system as in figure and we need to estimate the effort with FP. The "item" field will be taken from an externa WH (EIF) while "Customer" and "Invoice" will be present on our system (ILF).
From ER we can see easily how each EIL and ILF have only one RET, identifiable with a single entity, while each entity has as many DET as the number of attributes. As summary:

ILF/EIF	DET	RET
Customer (ILF)	6	1
Invoices (ILF)	4	1
Item (EIF)	3	1

From the transactional pow, instead, we need insert, update and delete from each ILF thus we have 6 EI (the user insert the informations). We have 1 EQ, that is print to the user the info of a client (query the system to get informations). Once the total number of FP is computed, it is possible to convert FP into LOC to perform further analisys, with CoCoMo for example.

5 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 model parameters are derived from fitting a regression formula using data from historical projects.
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 (M) is measured and estimated through the formula: $M = a \times (KLOC)^b$ while time is estimated from effort: $T = a \times (M)^b$, where the four 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;

Generally M is measured in Person-per-Month (short PM or MM - man per month). The total measure will then be fixed with some adjusting parameters.
COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. The first level, **Basic** COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is limited due to its lack of factors to account for difference in project attributes (Cost Drivers). **Intermediate COCOMO** takes these Cost Drivers into account and **Detailed** COCOMO additionally accounts for the influence of individual project phases. Last one is **Complete** COCOMO model which is short coming of both basic and intermediate. Cost Drivers are grouped into:

- **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;

Simple cocomo, also called cocomo-81 suffers from the fact that it was designed when only waterfall model was used for software processes. Second, the granularity of the software cost estimation model used needs to be consistent with the granularity of the information available to support software cost estimation. In the early stages of a software project, very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project, or the detailed specifics of the process to be used.

5.1 CoCoMo 2

CoCoMo 2 is the revised version of the CoCoMo. CoCoMo II enables projects to provide coarse-grained cost driver information in the early project stages, and increasingly fine-grained information in later stages. COCOMO II will then produce range estimates tied to the degree of definition of the project instead of specific points. The estimation will be more accurate as the project became more defined.
It consists of three sub-modules, depending on the complexity of the project:

- **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:

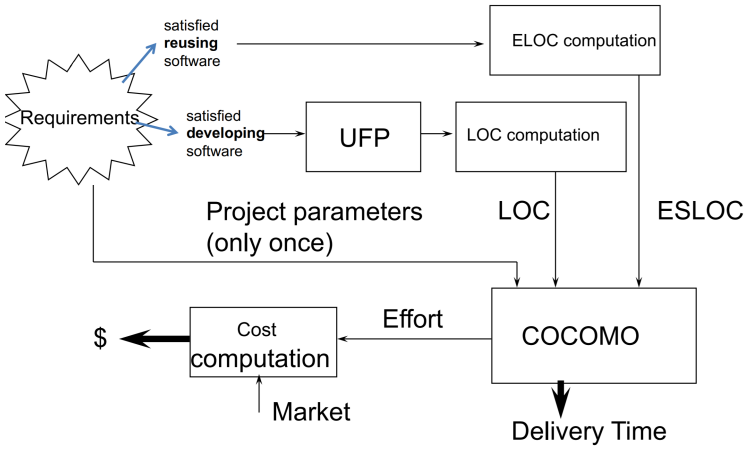
- Prototyping estimation: is the earliest project phases, or also callled spiral cycles, will generally involve prototyping, using the Application Composition model capabilities. The COCOMO II Application Composition model supports these phases, and any other prototyping activities occurring later in the life cycle.
- Early design model involves exploration of alternative software/system architectures and concepts of operation. At this stage, not enough is generally known to support fine-grain cost estimation. The corresponding COCOMO II capability involves the use of function points and a course-grained set of 7 cost drivers;
- Post architecture model involves the actual development and maintenance of a software product. This stage proceeds most cost-effectively if a software life-cycle architecture has been developed; validated with respect to the system's mission, concept of operation, and risk; and established as the framework for the product.

It uses source instructions and/or function points for sizing, with modifiers for reuse and software breakage; a set of 17 multiplicative cost drivers; and a set of 5 factors determining the project's scaling exponent. These factors replace the development modes (Organic, Semidetached, or Embedded) in the original COCOMO model.

In COCOMO II effort is expressed as Person Months (PM). person month is the amount of time one person spends working on the software development project for one month. This number is exclusive of holidays and vacations but accounts for weekend time off.
The logical source statement has been chosen as the standard line of code (SLOC). Defining a line of code is difficult due to conceptual differences involved in accounting for executable statements and data declarations in different languages. Since there are some trouble into using this kind of measure, due to instrinsecal different nature of various languages, SEI has defined a checklist of thinks to take into account to leveradge the differences among lenguages.
Cost drivers are used to capture characteristics of the software development that affect the effort to complete the project.
CoCoMo 2 model also provides a way to reuse some *SLOC* to adapt it to another *project*. Analysis of reuse costs across nearly 3000 reused modules in the NASA Software Engineering Laboratory indicates that the reuse cost function is nonlinear in two significant ways:

- It does not go through the origin. There is generally a cost of about 5% for assessing, selecting, and assimilating the reusable component.
- It does not go through the origin. There is generally a cost of about 5% for assessing, selecting, and assimilating the reusable component.
- Small modifications generate disproportionately large costs. This is primarily due to two factors: the cost of understanding the software to be modified, and the relative cost of interface checking.

Thus, as soon as one goes from unmodified (**black-box**) reuse to modified-software (**white-box**)reuse, one encounters this **software understanding penalty**. The COCOMO II treatment of software reuse uses a nonlinear estimation model. This involves estimating the amount of software to be adapted, ASLOC, and three degree- of-modification parameters: the percentage of design modified (**DM**), the percentage of code modified (**CM**), and the percentage of modification to the original integration effort required for integrating the reused software (**IM**). The Software Understanding increment (SU) is obtained from a table considering all those parameters.
CoCoMo II metodology schema:



6 Exam Questions

6.1 Function Point and CoCoMo question

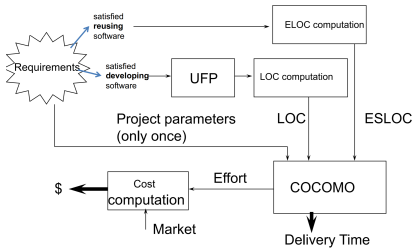
Function Point is a technique used in order to valuate size of *software products* and *productivity* of development team. FP is an objective technique where the goal is to weight functionalities of the system in terms of *data* and *relevant processes* for guests. There are 5 main *functionalities*:

- **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*

6.2 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

6.3 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.

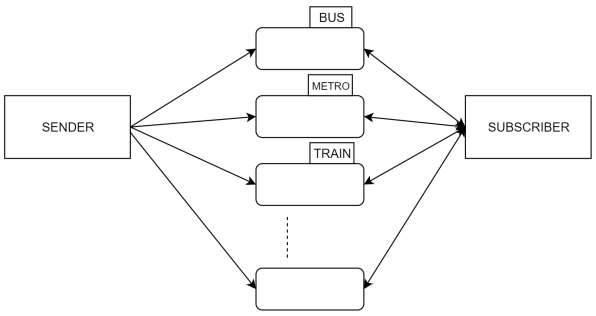


Figure 1: Architecture example

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

6.4 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();
```

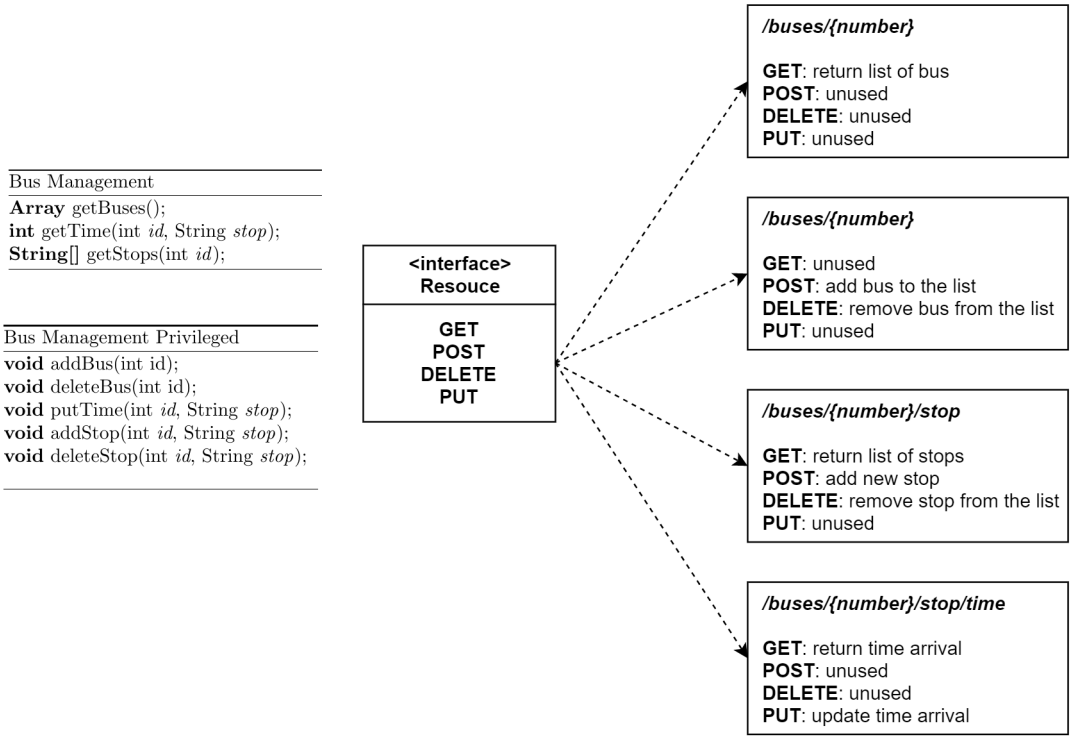
Client

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

Resource Repository

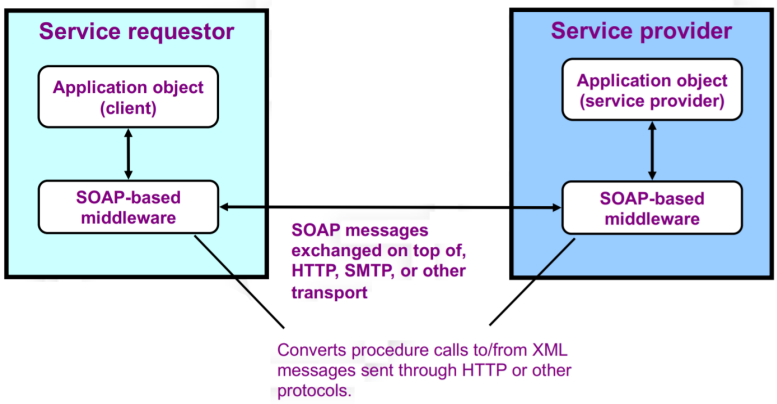
```
Array resArray = new Array();

@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);
}
```



6.5 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 {
    ...
}
```