POLITECNICO
MILANO 1863

Politecnico di Milano
A.A. 2016–2017
Software Engineering 2: "PowerEnJoy"

Project Plan

Pietro Ferretti, Nicole Gervasoni, Danilo Labanca

January 21, 2017

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide a detailed analysis of the PowerEnjoy software development project in terms of required cost and time. It highlights the estimation of

- project size, calculated using the *Function Points approach* by IBM;
- project cost and effort, calculated using the *COCOMO II* by Boehm.

Given the previous information we elaborate a feasible schedule considering all the necessary activities in detail, thus the best resources' allocation on each one. The last section of the document focuses on handling all the possible risks that could be met during the whole process, from the requirements analysis to the final testing and deployment.

1.2 Scope

The aim of this project is to specify and design a new digital management software for PowerEnJoy, a car-sharing service that employs electric cars only.

PowerEnJoy will offer a very valuable service to its users, letting them borrow cars to drive around the city freely, as an alternative to their own vehicles and public transport. Among the advantages of using PowerEnJoy we can note being able to find available cars in any place that is served by our system and having dedicated spots to park in (namely, PowerEnJoy's power grid stations). Furthermore, thanks to the fact that all the cars that we provide are electrically powered, PowerEnJoy is also very environmentally friendly.

1.3 List of Definitions and Abbreviations

1.3.1 Definitions

1.3.2 Acronyms

- **ITPD**: Integration Test Plan Document
- **DD**: Design Document
- **RASD**: Requirements Analysis and Specification Document
- **DB**: Database
- **PGS**: Power Grid Station
- **GPS**: Global Positioning System
- **API**: Application Programming Interface
- **ISDTN**: International Standard Date and Time Notation
- **EM**: Effort Multiplier
- **FP**: Function Points
- **ILF**: Internal Logic File
- **ELF**: External Logic File
- **EI**: External Input
- **EO**: External Output
- **EQ**: External Inquiries
- **UI**: User Interface

1.3.3 Abbreviations

- **Tx**: Task

1.4 List of Reference Documents

- Requirements analysis and specification document: "RASD.pdf"
- Design document: "DD.pdf"
- Integration testing document: "ITPD.pdf"
- Project description document: "Assignments AA 2016-2017.pdf"
- Example document: "Project planning example document.pdf"
- "COCOMO II – Model Definition Manual", version 2.1, 1995-2000, Center for Software Engineering, USC

2 Project size, cost and effort estimation

2.1 Size estimation: function points

Function points are useful in expressing the number of business functionalities our software has to provide to a user, and are used to compute an estimation of its size. They can be identified and categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. Each functional requirement is then assessed for complexity and assigned a number of function points. We based our computation on tables and values in the *COCOMO II Model Definition Manual v. 2.1.*

2.1.1 Internal Logic Files (ILFs)

Internal Logic Files are all the kinds of data used and managed by the application in order to offer the expected functions.
Data will be organized in the following tables in the DB:

- **User** : name, surname, username, password, dob, email, licenseID, cvv, cardNumber, accountStatus
- **Bill** : associatedLicense, total, date, rideID, carID, paymentStatus
- **Car** : model, plate, ID, available, issues
- **Report** :carID, description, associatedLicense, date
- **Safe area** : latitude, longitude, ID
- **PGS** : latitude, longitude, ID
- **Plug** : availability, ID
- **Reservation** : ID, associatedLicense, carID, date, status
- **Ride** : ID, associatedLicense, associatedBill, date, status, ridingTime, carID

The software will operate directly on the single rows and tables, joining them appropriately if needed.
All this data is modeled in simple structures so its complexity can be considered low (referring to tables).

ILF	Complexity	FPs
User	Low	7
Bill	Low	7
Car	Low	7
Report	Low	7
Safe area	Low	7
PGS	Low	7
Plug	Low	7
Reservation	Low	7
Ride	Low	7
Total		63

2.1.2 External Logic Files (ELFs)

The situations where our system demands external data are when it needs information regarding geolocation, or when it must guarantee the legal validity of driving licenses.
In particular:

- **GraphHopper API**:
  - Given the string containing the address, the API returns a pair of float numbers representing the coordinates of that location.
  - Given two pair of coordinates, the API return a float representing the time necessary to travel between two positions.
- **Eucaris API**:
  - Given name, surname, driving license ID and expiration date as string,s the API returns a boolean value representing the correspondence with an existing driving license in Eucaris DB.

In the final analysis, as the data involved are only strings and numbers of small size, we can assess these logic files as low complexity.

ELF	Complexity	FPs
Reverse geocoding	Low	5
Isochrone distance	Low	5
Driving Licenes legal validity	Low	5
Total		15

2.1.3 External Inputs (EIs)

PowerEnjoy offers a diverse set of functionalities that requires the user's input. In particular:

- **Login**: this functionality demands only two strings as parameters, the username and the password, that will be compared with the ones stored in the DB. We can consider this a low complexity operation.
- **User update**: this functionality includes a collection of operations that allow to modify each aspect of the user's profile. The input data are simple strings. Since the possibilities are conspicuous and the different elaborations aren't basic, and futhermore they interest several components, we can regard this functionality as an average complexity operation.
- **Pay bill** (automatically/manually): this is one of the most complex operations. It involves internal components and external APIs and it demands two numbers as input. Given the relevance of the operation and the parts interested, we can consider this as a high complexity operation.
- **Create reservation**: this functionality requires as input the user ID and car ID that we can consider simple, but its complexity is due to the components involved. At the moment of the creation, the application verifies if the user is suspended, and after that the availability field of the tuple representing the car is modified and the car itself is put in order to be unlocked. Moreover the reservation made by the user is inserted in the list of reservations. We classify the operation as having average complexity.
- **Cancel reservation**: the analysis made for the previous functionality is also valid for this one. The operations are comparable as elaboration, and thanks to this the complexity of this functionality is average.
- **Start ride**: this action is very simple as it needs only the user ID and the car ID and starts the tally for the duration of the ride. Made this consideration we can see this functionality as low complexity.

- **End ride**: this functionality is one of the most complex ones because it involves many different components, and starts the payment process (that has high complexity on its own). When a ride ends the car is set as available in the DB, it creates a bill for the user and initiates the payment. As usual the inputs are simple strings.
- **Park**: this functionality expects as inputs the position of the car that are a pair of numbers, the user ID and the car ID. In the elaboration of this data the components interested are the internal car system and the DB to verify if the position belongs to a safe area. Due to the urgency of the action this functionality can be classified as average complexity.
- **Unlock car**: this functionality receives the position of the user and his ID, and the car ID. After that the system proceeds to notify the car to open the doors and calls the *start ride* function. Since it includes another functionality and communicates with the car we classify this functionality as average complexity.
- **Update car**: this functionality requires only two parameters: the car ID and the status to be updated. This operation mainly involves the DB. For these reasons this functionality has low complexity.
- **Update plug**: as the previous functionality, this one too only requires two parameters and affects the DB. So it's a low complexity functionality.
- **Set car available**: as the previous functionality, this one too only requires two parameters and affects the DB. So it's a low complexity functionality.
- **Set car available**: as the previous functionality, this one too only requires two parameters and affects the DB. So it's a low complexity functionality.
- **Report issue**: this functionality receives the parameters of a form as input, so a set of strings, and modifies the status of the car in the DB and the car record itself. We can consider this functionality as an operation with average complexity.

EI	Complexity	FPs
Login	Low	3
User update	Average	4
Pay bill	High	6x2
Create reservation	Average	4
Cancel reservation	Average	4
Start ride	Low	3
End ride	High	6
Park	Average	4
Unlock car	Average	4
Update car	Low	3
Update plugs	Low	3
Set car unavailable	Low	3
Set car available	Low	3
Report issue	Average	4
Total		60

2.1.4 External Inquiries (EQs)

In this section we will discuss about External Inquiries that could be defined as elementary processes that send data or control information outside the application boundary. An external inquiry is a functionality that provides information to a user after a query or a request, by retrieving data from an ILF or ELF.
In our case we have:

- **Get user info**: it returns a set of strings representing the user's info, from the DB.
- **Get bills**: it returns a list of bills (mainly strings) owned by the user, retrieved from the DB.
- **Car/PGS/Safe area search with position**: in this case the function needs a parameter that is a pair of numbers and returns a list extracted from the DB. For these operations there are some computations to do. Because of that we classify these actions having average complexity.
- **Car/PGS/Safe area search with address**: as for the previous cases, these functionalities also have an average complexity, also considering that there are interactions with the *GraphHopper API* in order to convert the address in coordinates.

- **Money Saving Option**: this is one of the most complex operations, because it has extract information from the DB and do complex elaborations to find the best solution.
- **Cars in need of maintenance**: this is a simple functionality that is employed by maintenance operators and returns a list of cars that are unavailable.

EQ	Complexity	FPs
Get user info	Low	3
Get bills	Low	3
Car/PGS/Safe area search with position	Average	4x3
Car/PGS/Safe area search with address	Average	4x3
Money Saving Option	High	6
Cars in need of maintenance	Low	3
Total		39

2.1.5 External Outputs (EOs)

The situation our system needs to notify external agents are the following:

- **Un/Lock car**
- **Car update**

The first case has a low complexity, while the second is a little bit more complex because it needs to extract the information from the car.

EO	Complexity	FPs
Un/Lock car	Low	4x2
Car update	Average	5
Total		13

2.1.6 Overall estimation

Function Type	**FPs**
Internal Logic Files	63
External Logic Files	15
External Inputs	60
External Inquiries	39
External Outputs	13
Total	190

This table contains a recap of the evaluations of each type of function point. By enumerating the function points we can find an estimation for the number of lines of code the software will have. The formula we are going to use is:

$$SLOC = AVC \times FPs$$

where

AVC is a language dependent factor

FPs is equal to the number of function points (190 in our case)

Since our system will be developed using the Java Enterprise Edition stack, we will take 46 as the AVC factor (following standard empirical measurements[1] for an average project).

In the end, we have:

$$SLOC = 8740 \text{ lines of code}$$

[1] http://www.qsm.com/resources/function-point-languages-table

2.2 Cost and effort estimation: COCOMO II

To estimate the cost and effort required to develop the software we use the COCOMO II model[2].

Since the architecture and structure of our software has been already designed and thought out in details in the Design Document, we will use the COCOMO II Post-Architecture Model.

2.2.1 Scale Factors

Five scale factors are used to compute the scale exponent E, that accounts for the economies or diseconomies of scale that characterize our project.

If $E < 1.0$, the project exhibits economies of scale, otherwise diseconomies of scale are present (usually because of communications overhead or large-system integration overheads).

There are five scale factors: *Precedentedness*, *Development Flexibility*, *Architecture/Risk Resolution*, *Team Cohesion*, *Process Maturity*. Each of them can be classified on a scale from "Very Low" to "Very High".

Scale factors:

- **Precedentedness (PRED):** Precedentedness is high if the product is similar to previously developed projects.

  We never developed anything similar and on the same scale, so we rate this factor as *Very Low*.

- **Development Flexibility (FLEX):** This factor measures how rigidly must the software conform with the requirements and the interface specifications.

  We do follow the requirements to reach all of the customer's goals, but this is not a project were extreme rigorousness is necessary due to hardware constraints or danger to things or people. We rate this as *Nominal*.

- **Architecture / Risk Resolution (RESL):** High if the architecture has been carefully designed, the risks have all been identified and accounted for, and there is low uncertainty and risk in general.

  We consider the architecture we developed as robust and detailed, and the risk management plan covers most of the problem we could encounter during the development, with detailed strategies for all relevant risks. We rate this factor as *High*.

- **Team Cohesion (TEAM):** This factor quantifies the personnel's experience as a team, willingness to work together and the presence of a shared vision and commitment to the project.

[2] http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

We, as the developers for this project, have already worked together multiple times and have a lot of experience working as a team; we therefore rate this factor as *Very High*.

- **Process Maturity (PMAT):** this factor follows the Capability Maturity Model (CMM), associating a rating to each CMM level.

  We are not at a process level where everything is quantified and optimized yet, but we do follow clear, defined methods. It is reasonable to judge it as a level 3 in the CMM ("defined"), equivalent to a *High* rating.

Scale Factor	Level	Value
Precedentedness (PREC)	Very Low	6.20
Development Flexibility (FLEX)	Nominal	3.04
Architecture / Risk Resolution (RESL)	High	2.83
Team Cohesion (TEAM)	Very High	1.10
Process Maturity (PMAT)	High	3.12
Total		16.29

Table 1: COCOMO II values for the scale factors.

2.2.2 Cost Drivers

To accurately determine the amount of effort the development will require, the nominal, baseline effort is adjusted with several Effort Multipliers. In the Post-Architecture Model these are also known as *cost drivers*.

There are 17 different cost drivers, and each one can have a rating ranging from "Very Low" to "Very High". The ratings map directly to a multiplicative factor, that can increase or decrease the estimated amount of effort needed to develop the software.

Product Factors:

- **Required Software Reliability (RELY):**

  This factor conveys the necessity for our software to work correctly over a long period of time. If the effect of a software failure is only slight inconvenience then RELY is very low. If a failure would risk human life then RELY is very high.

  Our software doesn't control the cars' hardware (with the exception of the locking/unlocking mechanism), so there's no worry that failures could cause accidents or injuries to our customer, and the unlikely cases of people getting stuck in cars can be avoided with simple failsafes (being able to unlock the car from the inside). In case of a system failure the only

negative effect would be that our service would be unavailable for our customers, but we judge this to lead to at most moderate losses. We consider this driver *Nominal*.

- **Database Size (DATA):** This cost driver attempts to capture the effect large test data requirements have on product development. The rating is determined by calculating D/P, the ratio of bytes in the testing database to SLOC in the program.

  We believe our testing database will be at least 100MB of size. Given that we estimated the source code to be around 9000 lines long, this brings the D/P ratio to a order of magnitude of 10^4.

  Following the indications of the COCOMO II model, this is equivalent to a rating of *Very High*.

- **Product Complexity (CPLX):** To evaluate complexity we need to weigh five areas: control operations, computational operations, device-dependent operations, data management operations, and user interface management operations.

  Our software doesn't need complex computations or special operations, except for some data handling. Checking on the COCOMO tables for product complexity and averaging the results we judge the complexity rating to be *Nominal*.

- **Developed for Reusability (RUSE):** This cost driver accounts for the additional effort needed to construct components intended for reuse on current or future projects.

  We have no requirements asking us to develop code ready to be reused in other projects or products. No reusability constraints. We have no requirements for developing code to be used in other products, and know of no plans for new or future projects. Therefore we have very little reusability constraints and we can rate this driver as *Low*.

- **Documentation Match to Life-Cycle Needs (DOCU):** The rating scale for the DOCU cost driver is evaluated in terms of the suitability of the project's documentation to its life-cycle needs.

  We have no specific instructions about the software's documentation, so we plan to include the right amount of documentation without wasting too much effort to produce more than what will be actually needed for the extent of the product life-cycle. This factor is rated as *Nominal*.

Platform Factors:

- **Execution Time Constraint (TIME):** This is a measure of the execution time constraint imposed upon a software system. The rating is expressed in terms of the percentage of available execution time expected

to be used by the system or subsystem consuming the execution time resource.

We want our software to be as responsive and efficient as possible, without requiring extremely performant hardware. We expect to occupy 3/4 of all the available processing power on an any given moment. Following the COCOMO II guidelines, this is equivalent to a *High* rating.

- **Main Storage Constraint (STOR):** This rating represents the degree of main storage constraint imposed on a software system or subsystem.

  In today's terms storage is easily, readily available and very cheap. We will have no problem acquiring enough hardware for all of our storage needs, so this factor can be safely seen as *Nominal*.

- **Platform Volatility (PVOL):** "Platform" is used here to mean the hardware and software (OS, DBMS, etc.) which our software will have to rely on to work. This rating ranges from low, if there is a major change every 12 months, to very high, in the case of a major change every two weeks.

  We want our software to be kept updated to fix bugs and possible vulnerabilities. A major update every 2 months and minor ones every week seems reasonable. This corresponds to a rating of *High*.

Personnel Factors:

- **Analyst Capability (ACAP):** Analysts are the people who work on requirements, high-level design and detailed design. The major attributes that should be considered in this rating are analysis and design ability, efficiency and thoroughness, and the ability to communicate and cooperate.

  Even if we believe our analysis to be sound, we consider ourselves to be no more than average analysts. This driver is *Nominal*.

- **Programmer Capability (PCAP):** This driver accounts for the developers ability, efficiency and thoroughness, and the ability to communicate and cooperate.

  We are very confident in our abilities in programming, and we know that we can work really well as a team. We see nothing wrong in choosing a *Very High* for this driver.

- **Personnel Continuity (PCON):** The rating scale for PCON is in terms of the project's annual personnel turnover: from 3%, a very high continuity, to 48% in particular settings of high turnover.

  We are going to be the ones developing this piece of software, and we are pretty dedicated to following this through until the end. We rate the probability of any of us leaving as negligible; therefore we predict the continuity of the project development to be *Very High*.

- **Applications Experience (APEX):** The rating for this cost driver is dependent on the level of applications experience possessed by the project team developing the software system or subsystem. The ratings are defined in terms of the project team's equivalent level of experience with this type of application.

  We sincerely don't have much experience with this type of application, except for basic client-server architectures. This driver can be rated as *Low*.

- **Platform Experience (PLEX):** This factor expresses the experience with using powerful platforms, including graphic user interface, database, networking, and distributed middleware frameworks.

  We have very little experience with database, networking, and distributed middleware software. We have to rate this as *Very Low*.

- **Language and Tool Experience (LTEX):** This is a measure of the level of programming language and software tool experience of the project team developing the software system or subsystem.

  We have a good amount of experience with Java, but not as much with the J2EE framework. We can consider this as *Nominal*, not too high and not too low.

Project Factors:

- **Use of Software Tools (TOOL):** This driver measures the capabilities of the tools and editors the developers are going to use.

  Modern Java IDEs like Eclipse and Netbeans are very advanced and perfectly fit into the COCOMO classification for "strong, mature, proactive life-cycle tools, well integrated with processes, methods, reuse". What these IDEs lack can be easily integrated with additional software and plugins, like Sonar. We believe a rating of *Very High* to be appropriate.

- **Multisite Development (SITE):** This driver is a value averaging of two factors: site collocation (from fully collocated to international distribution) and communication support (from surface mail and some phone access to full interactive multimedia).

  As developers we are approximately located in the same metro area, and are able to meet whenever it is needed to work together. Furthermore we have access to many different communication tools, like emails, messaging applications, source control software and web-based project management applications. We rate this factor as *High*.

- **Required Development Schedule (SCED):** This rating measures the schedule constraint imposed on the project team developing the software. The ratings are defined in terms of the percentage of schedule stretch-out

or acceleration with respect to a nominal schedule for a project requiring a given amount of effort.

We have no specific requirements or constraints on development time and schedules. We can keep the typical time needed for development, with no schedule acceleration or stretching. This factor is therefore *Nominal*.

Cost Driver	**Level**	**Value**
Required Software Reliability (RELY)	Nominal	1.00
Database Size (DATA)	Very High	1.28
Product Complexity (CPLX)	Nominal	1.00
Required Reusability (RUSE)	Low	0.95
Documentation match to lyfe-cycle needs (DOCU)	Nominal	1.00
Execution Time Constraint (TIME)	High	1.11
Main Storage Constraint (STOR)	Nominal	1.00
Platform Volatility (PVOL)	High	1.15
Analyst Capability (ACAP)	Nominal	1.00
Programmer Capability (PCAP)	Very High	0.76
Personnel Continuity (PCON)	Very Low	0.81
Application Experience(APEX)	Low	1.10
Platform Experience (PLEX)	Very Low	1.19
Language and Tool Experience (LTEX)	Nominal	1.00
Usage of Software Tools (TOOL)	Very High	0.78
Multisite Development (SITE)	High	0.93
Required Development Schedule (SCED)	Nominal	1.00
Total		0.907

Table 2: COCOMO II values for the cost drivers.

2.2.3 Effort equation

We use the formula from the COCOMO II Post-Architecture Model to estimate the effort needed for the development.
We will refer to the estimated Person-Month effort needed to develop the code as PM.

$$PM = A \times Size^E \times \prod_{i=1}^{17} EM_i$$

$$E = B + 0.01 \times \sum_{j=1}^{5} SF_j$$

where
$A = 2.94$ is a calibrated constant
$B = 0.91$ is a calibrated constant
EM_i are the effort multipliers (cost drivers)
SF_j are the scale factors
$Size$ is the estimated thousands of lines of source code ($KSLOC$)

By substituting the numbers obtained in the previous analysis and the COCOMO II model we have

$$E = 0.91 + 0.01 \times 16.29 = 1.07$$

$$PM = 2.94 \times 8.7^{1.06} \times 0.907 = 26.41 \text{ person-months}$$

We can see that $E > 1$, displaying diseconomies of scale. This is mainly because this project is mostly unprecedented.

2.2.4 Schedule estimation

According to the COCOMO II model, we can estimate the approximate is the time necessary to develop the code in calendar months. We will refer to the estimate as $TDEV$.
Assuming no schedule compression or stretching-out,

$$TDEV = C \times PM^F$$

$$F = D + 0.2 \times (E - B)$$

where
$B = 0.91$ is a calibrated constant
$C = 3.67$ is a calibrated constant
$D = 0.28$ is a calibrated constant
E is the scaling exponent for the effort equation
PM is the number of person-months obtained in the effort estimation

Our results are

$$F = 0.28 + 0.2 \times (1.07 - 0.91) = 0.31$$

$$TDEV = 3.67 \times 26.41^{0.31} = 10.13 \text{ months}$$

3 Schedule

In this section we're going to present an undetailed and high-level schedule of the project. Since this agenda is drawn up in the early days of the software development, we could need to reschedule some activity later.
These are the main tasks and a simple description of each of them:

- **RASD writing**: we analyse the requirements submitted by the customers and draft the document that contains the most important features of the project and it's indispensable for the completation of the other documents.
- **DD writing**: we make decision about architectural design and composition of the components of PowerEnjoy system, then we go on with the draft of the document. It's really important to terminate it before starting to code.
- **ITPD writing**: we make a detailed analisys of the components and then write the document that describes the strategy we are going to use for unit testing, integration testing and system testing.
- **PP writing**: we define planning assumptions and decisions, the risks that we could incur during the development and how we could tackle them.
- **Development**: we proceed implementing the real software.
- **Code inspection**: before we move forward with the testing, we check if the code if bug free and prove to enhance the quality of the code.
- **Testing**: during and after the implementation of the software we perform the testing of the code and the system.
- **Deployment**:
- **Start up**:

Since we choose to follow bottom-up approach for testing, the latter, development and code inspection could be performed almost simultaneously.

3.1 T1 - RASD writing

3.2 T2 - DD writing

3.3 T3 - ITPD writing

3.4 T4 - PP writing

3.5 T5 - Development - Code Inspection - Testing

3.6 T6 - Deployment - Start Up

4 Resource allocation

Task	Pietro Ferretti
T1	All
T2	All
T3	All
T4	All
T5	External APIs analysis
	DBMS preparation
	System development
	Unit test
	Bug fixing
	Client-side development
	Meeting with customers
	Final revision

Task	Nicole Gervasoni
T1	All
T2	All
T3	All
T4	All
T5	External APIs analysis
	DBMS preparation
	System development
	Integration testing
	Bug fixing
	UI development
	Meeting with customers
	Final revision

Task	Danilo Labanca
T1	All
T2	All
T3	All
T4	All
T5	External APIs analysis
	DBMS preparation
	System development
	System testing
	Bug fixing
	Client-side development
	Meeting with customers
	Final revision

5 Risk Management

Risk management is the identification, assessment, and prioritization of risks followed by coordinated and economical application of resources to minimize, monitor, and control the probability and/or impact of unfortunate events.
In this section we list PowerEnjoy project's main risks and outline possible strategies to handle the most critical ones.

Risk	Probability	Impact
Personnel shortfall (recruitment issues, employee illness or accidents, ...)	moderate	catastrophic
Inaccurate Requirements	moderate	critical
Unrealistic Schedule	moderate	critical
Unrealistic Budget	moderate	critical
Stakeholder commitment loss	low	critical
New car rental laws	low	catastrophic
Inability to obtain proper permits from authorities	low	catastrophic
Inability to obtain a deal with a mobile data provider	low	critical
Issues with hardware supplier (wrong or defected items, late deliveries, ...)	moderate	critical
Wrong user interface	moderate	critical

Strategies:

- *Personnel shortfall:* reorganize team so that there is more overlap of work and no single person is fundamental to the project; in case of issues in finding high qualified worker consider buying already developed products and simply adapting them.
- *Issue with hardware supplies:* use and test hardware components as soon as they are available; replace potentially defective components with bought-in components of known reliability.
- *Inaccurate requirements:* derive traceability information to assess requirements change impact; divide software functions in several different module so that modify a requirements will not have effect on all the produced code.

- *Stakeholder commitment loss:* prepare a briefing document showing how the project is making a very important contribution to the goals of the business.
- *Unrealistic budget:* have it reconsidered by showing the stakeholder how the project is making a very important contribution to the goals of the business; buy already developed products and assembly them.
- *Unrealistic schedule:* early in the development, divide bigger activities in smaller ones and hired new people to work on few specific functions then combine them when ready, while later consider buying already developed products and adapting them.

A Hours of work

- Pietro Ferretti: hours of work
- Nicole Gervasoni: hours of work
- Danilo Labanca: hours of work

B Changelog

- PP v1.0, published on January 22, 2017