Advanced Software Engineering - Appunti

Francesco Lorenzoni

September 2023

Contents

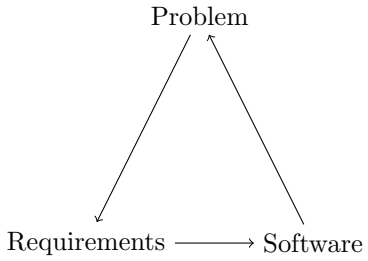
Chapter 1

Introduction

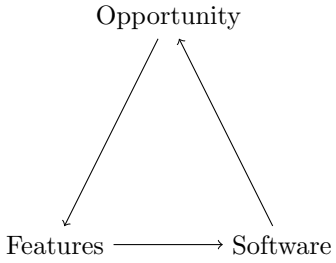
27 - Settembre

1.1 Product based

In *Project-based SE* there is loop which nowdays cripples software since its early stages of development. This is due to mutable nature of requirements, which often change throughout time along the features implemented by the software.



Product-based SE is opposed to *Project-based SE* and the above pictures changes as follows.



1.2 Agile

Agile is a collection of principles and methods applied in the software development field.

Opposed to project-based SE, in Agile the client is requested to express the requirements not in technical terms but in features.

Agile suggests an incremental development model

Principles

1. Satisfy customer through early and continuous delivery of valuable software
2. Welcome changing requirement, even late in development. Agile processes harness change for the customer's comptetitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale

4. Business people and devs must work together daily throughout the prokect
5. Build porjects around motivated individuals and give them the environment and support they need
6. The most efficient and effective method of conveying information to and within a dev team is face-to-face conversation
7. Working software is the primary measure of progress
8. Agile processes promote sustainable dev
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity i.e. art of maximizing the amount of work not done is essential
11. The best architectures, requirements, and designs emerge from self-organizing teams.

Extreme Programming was proposed as part of the agile methodology

1.3 Scrum

Since requirements changes are rather frequent, long-term plans are unreliable, hence SE aims to formulate short-term plans.

Scrum is found on **empiricism** and **lean thinking**; it asserts that knowledge comes from experience, and that decisions should be made on observations.

Other key terms are code **Transparency** among the team and with the customer, **Inspection** of produced code and software (artifacts), **Adaptation** to changes in features and requirements.

The **Scrum Team** is composed by:

1. **Product Owner**: must ensure that the dev team is always focused on the goal
2. **Scrum Master**: Scrum expert which drives the team to apply properly the Scrum framework.
3. **Developers**: actual monkeys people which write code

In scrum SW is developed in **sprints**, i.e. fixed-length periods with a specific goal to be achieved.

- Product backlog: to-do list of items to be implemented
- Timeboxed sprints
- Self-organizing teams

... **Prod Backlog Revised**
PBI Estimation Metrics

1.3.1 Timeboxed Sprints

Even if at the end of a srpint the goal hasn't been reached, "no worries", the work stops anyway; there will be a new sprint which will include the work which has not been implemented in the previous one.

1.3.2 Scrum Meetings

1.3.3 Agile activities

Test automation Continuous integration

1.3.4 Sprint reviews

At the end of each sprint there is a review meeting which involves the *whole* team. The *product owner* has the ultimate authority to decide wether the sprint goal has been reached or not. The sprint review should include a process review, in which the whole team shares ideas on how to improve their way of working.

Team size

1.4 5 - Ottobre

Chapter 2

Features, Scenarios, Stories

2.1 3 - Ottobre

Which factor drive the design of SW products?

- Inspiration
- Business/consumer needs not met by existing products
- Dissatisfaction with existing products
- Technical changes making new product types possible

Product-based software engineering needs less *requirements documentation* than project-based SE, since the requirements are not set by customers and it is allowed for them to change. The focus is instead on **features** (fragments of functionality); to understand which features are needed, we must first understand which may be **potential users**, through interviews, surveys, informal user analysis and consultation.

Flow-chart

User representations — **personas** — and natural language descriptions — **scenarios** and **stories** — help driving the identification of product **features**:

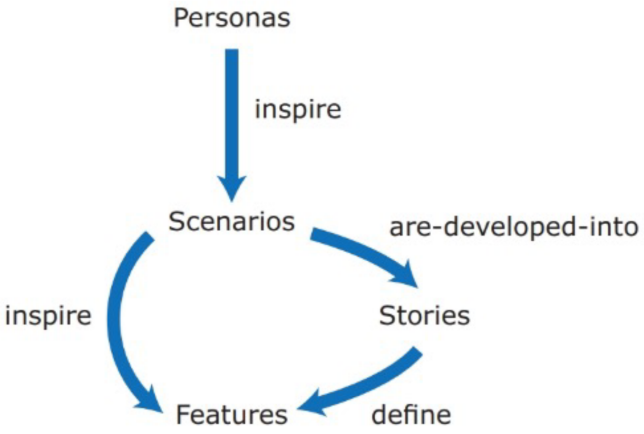
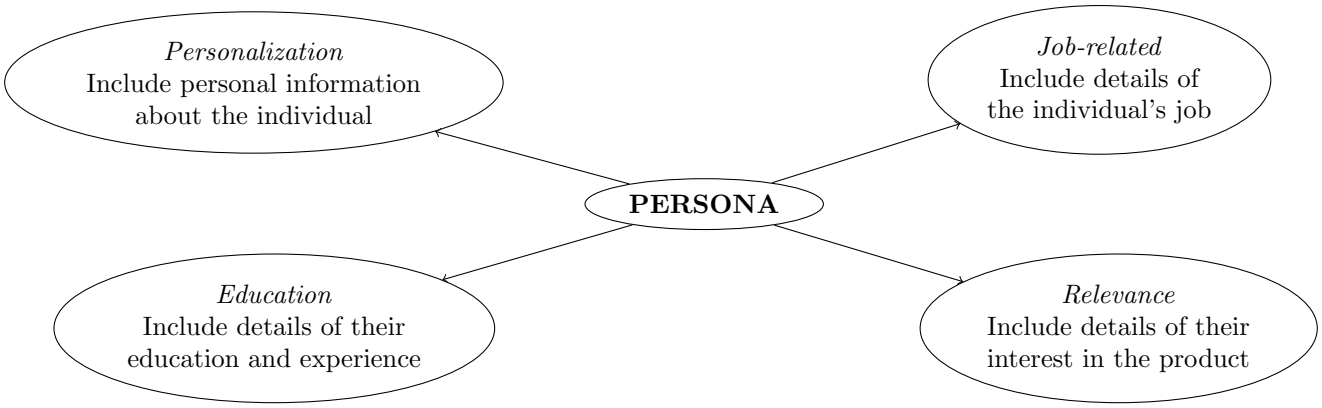


Figure 2.1: Features flowchart

2.2 Personas

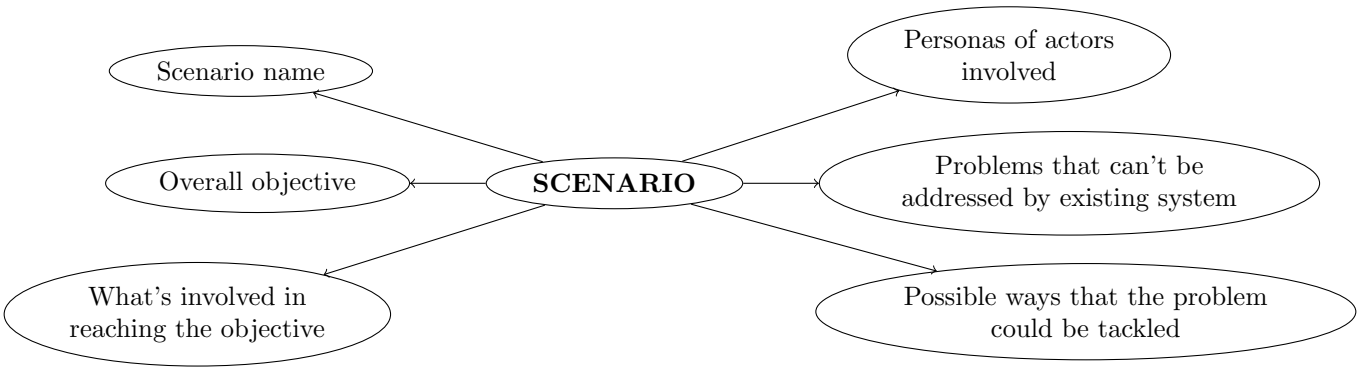
Personas represent the types of target users for our product. Each personas should highlight which are *background*, *skills* and *experience* of potential users. Usually only a couple of personas (max 5) are needed to identify ***key*** *product features.*



There are conflicting opinions about whether personas should include *photos* or not.
Photos may be misleading, since *"personas are not about how users look, but what they do"* (Steve Cable). *"Detailed personas encouraged the team to assume that demographic information drove motivations"* (Sara Wachter-Boettcher).

2.3 Scenarios

Having defined personas, to discover product features, it would aid to define *user interactions* with the product: a **scenario** is a narrative written from *user's perspective* describing a situation in which a user is using our product's features to do something she wants to do.
Scenarios are **not** *specifications*! They lack details and may be incomplete.



A proper amount of scenarios usually is 3-4 for each persona, aiming to cover the persona's main responsabilities. Each team member should create scenarios and discuss them with the rest of team and (possibly) users.

2.4 User Stories

As a	<role>
I want to	<do something>
So that	<reason/values>

Table 2.1: User Stories

Knowledge sources for feature design

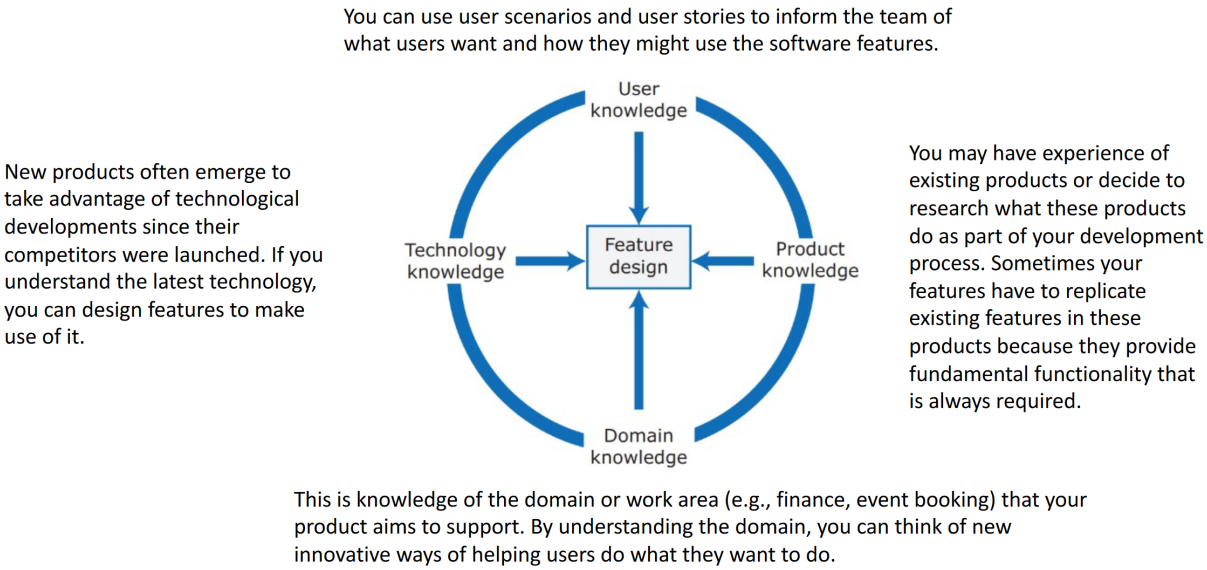


Figure 2.2: feature knowledge

While **scenarios** are high-level stories of product use, **User stories** are more fine-grained narratives. They allow to organize and chunk work into units which represent actual value to the customer, ultimately building software incrementally from the users perspective. Longer stories can be split into shorter stories, and to eventually prioritize them.
These words should recall *1. 3. 7. 10.* agile principles described in Section 1.2.
In fact, usually *Scrum product backlog* is a set of *user stories* sorted according to priority.

Even though it is possible to express all functionalities describe in a *scenario* using *user-stories*, scenarios can read more naturally, make stories understanding easier, and provide more context.

2.5 Feature identification

Our goal is to get a list of features that define our product, keeping in mind some **properties**:

- **Independence** → a feat should not dfepend on how other system feats are implemented and should not be affected by the order of activation of other feats
- **Coherence** → feats should be linked to a single item of functionality. They should not do more than one thing and should never have side effects
- **Relevance** → systesm feats should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

To derive features from scenarios and user stories, the dev team should discuss these and start prototyping to demonstrate first novel and critical features.

2.5.1 Feature Creep

Number of product features grows as new potential users are envisaged. To avoid this, 4 questions should be considered:

1. Does this feature add something new or is it simply an alternative way of doing something already supported?
2. Can this feature be implemented by extending an existing feat rather than adding a new one?

3. Is this feat likely to be important and used by most software users?
4. Does this feat provide general functionality or is it a very specific feat?

Chapter 3

User Stories

Consider **TicTacToe** — aka *Tris* or *Tiro Filetto* — and its basic rules. Suppose you have to develop a software that allows playing TicTacToe, i.e. our *Product*.

Proceeding in steps, let's define the **Personas** who would use our product.

Chapter 4

Seminario - Imola informatica

4.1 Takeaway Messages

Performance per se is not an accurate measure, there are many factors when developing SW systems which affect perfomance, like usability and efficiency.

4.2 Project Path

$$\textbf{Demand} \longrightarrow \textbf{Plan} \longrightarrow \textbf{Design} \longrightarrow \textbf{Develop} \longrightarrow \textbf{Release} \tag{4.1}$$

This (sadly not) deprecated path 4.1 leads to *situation rooms* and subsequent performance degradation, unsatisfaction and possible skyrocketing costs.

Performance should drive the whole production process, it shouldn't be treated as a post-go live concern, otherwise it may lead to the so called *situation rooms*[1].

4.3 Fitness function

A **fitness function** provides a summarised measure of how close a given design solution is to achieving the set aims.

4.4 Performance Best practices

"Starbucks does not use two-phase commit": they aim to maximize throughput, by using an employee chain to serve customers, from ordering to delivering coffee.

Enforce business process performance with adequate fitness functions:

- involve key stakeholders
- automatically assess and evaluate
- continuously review and tune

It is important to design IT architectures and solutions with real-world requirements in mind. For example "a customer shouldn't have to wait for more than 2s to *order* a coffee".
In distributed architectures, network's technical aspects and metrics must be taken into account: latency, available

[1] Often named also *war rooms*

bandwidth, dedicated or shared, network billing models...
Aside from requirements, also costs, performance and observability should be kept in mind.
To measure progress fitness function must be fed periodically with real-time data. Most of the times testing only in production is the only way to go, since mirroring the production environment and using/managing it during development would be hugely costful. However, precisely for this reason, production testing shouldn't be the only testing method.

4.5 Bad habits

- Worrying about performance only late in development
- Last minute testing
- Focusing only on performance as a technical POV, not user/business pov

Enable a culteure for perf across your entire value stream

embed per in business processes as well as IT systems
How can this integrate with the fact that user features shouldn't be tied to implementation details? How can be performance be measured aside from implementation/technical metrics?

evolutionary architectures need fitness functions

continuosuly refine fitness functions