Faat – Freelance as a Team

Rodrigo Borrego Bernabé
University of Salamanca
+34605152418
rodrigobb@usal.es

Iván Álvarez Navia
University of Salamanca
+34 923294400 Ext. 1513
inavia@usal.es

Francisco José García-Peñalvo
Research Institute for Education Sciences
(IUCE) University of Salamanca
+34 923294400 Ext. 1302
fgarcia@usal.es

Abstract

Agile methodologies are reliable engineering and management practices, capable of helping in the development of quality and successful software in business environments. However, most of these methodologies are centered on a development team and its internal communication. Moreover, for simplicity, a single product development is taken into account with its successive releases. There is another scenario: that of a single programmer working alone and often in much smaller projects and in several at the same time. Also in this scenario the client proximity is not as described by the agile environment ideal. In that case, the priorities and needs change, communication takes on another meaning and working mechanisms are not always comparable to that of a team. This paper introduces Faat (Freelance as a Team), a methodology specifically designed for those professionals. Integrating existing practices to the needs and possibilities of an individual programmer. However, it has been frequently considered the possible application of this methodology to small teams and/or other more general scenarios. This methodology has been tested in the web-based learning applications.

Categories and Subject Descriptors

D.2 [Software Engineering]

General Terms

Management, Measurement, Documentation, Performance, Design, Human Factors, Standardization, Theory.

Keywords

Development Process, Agile Methodology, Personal Software Process

1 INTRODUCTION

Agile methodologies have been created and developed to address the main problems of software development. But they are centered on a team and its internal communication through a single product development with successive releases. They can't be applied "as is" by a single programmer working alone in smaller projects and without client proximity, an understudied scenario with special characteristics [1], [2] and [3].

Through the pages that follow a set of practices are presented, most of them drawn from the main agile methodologies, in particular, those considered most profitable and easier to implement for a single developer.

TEEM '15, October 07 - 09, 2015, Porto, Portugal
 ACM 978-1-4503-3442-6/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2808580.2808685

They are divided in three groups: *strategic practices* (guidelines to choose an optimal decision at each point in software development); *workflow practices* (describe the operational processes to be performed during product development and the elements involved in them) and *auxiliary practices* (other practices that each developer should apply in accordance with its capabilities, experience and personal judgment).

There is no acceptable way to apply this methodology leaving out some of the workflow practices because the remaining would not be able to provide a significant improvement. Without include all the strategic practices, improvement in the development process and resultant product quality will be minor.

Second part in this paper describes a proposed implementation process of this methodology in an everyday work. Examples of application in real projects are presented including those taken from the authors' experience in software development, eLearning [4] and interactive learning/training systems [5], some others created to test Faat and a case of study.

This methodology should not be considered as a set of immutable dogmas. Everything must be analysed critically and the reader is advised to apply their experience to the process.

2 STRATEGIC PRACTICES

2.1 Simplicity

Simplicity must be promoted ever and deliver the simplest and minimal solution satisfactory to the client.
Simplicity--the art of maximizing the amount of work not done--is essential. - Agile Manifesto

A simpler design means less development time and it is cheaper replacing code when little time has been employed on it. It is usual trying to build an overly ambitious first version of a product, when in fact acceptance by users is unknown, resulting in much time spent on underutilized features.

Simplicity is a subjective attribute, and measuring the complexity of a design is a hard job. But *eXtreme Programming* [6] centres on try to achieve a TUBE design:

- *Testable*, unit/acceptance tests to check code can be written.
- *Understandable*, by someone outside the development team.
- *Browsable*, whatever wanted whenever wanted can be found.
- *Explainable*, easy to show new people how everything works.

This practice is related with *Minimum Viable Product* (MVP) and *Minimum Marketable Features* (MMF) concepts, which both refer to the minimum set of features that are able to provide value by themselves [7].

This practice has been also enunciated as the KISS principle (*"Keep it simple, stupid"*) and also related to YAGNI (*"You aren't gonna need it"*). Developers must control the impulse to add new, unneeded, features without knowing if they would be needed in the future. In fact, only 10% of extra unrequired features will ever get used, resulting in great lost of time [6].

2.2 Embrace Change

Keep flexibility, removing obstacles to change.

Change is your best friend. - [8]

Although the cost of change is never trivial, you should plan for change and understand its costs [9]. A tolerant to change architecture should be provided or agile will became fragile.

To embrace change, common elements can be abstracted and encapsulate the most changing in stable interfaces. Other practices should be explored and implemented as a complement to this methodology.

A mandatory requirement for the acceptance of change is that it is controlled at all times. The change-control tools (git, subversion, etc.) are essential parts in any development process. They allow establishing points of stable code to come back if any changes introduced any major fault. They also provide other advantages like to establish alternative solutions to a problem, change the user story in development, etc.

[10] describes a workflow with *git*, a version control system, that includes releases, versions, bug fixing and work coordination between different requirements. It can be very helpful to individual developers and to teams.

2.3 Make Decisions

Move fast, break things. – Facebook's headquarters, 2010.

Some authors argue software should be agnostic, unlimited and as flexible as possible, but it is not true. Not only the optimal product does not follow these guidelines, but also trying to reach them goes against the objectives of this methodology and it is counterproductive in the software development process.

Agnosticism increases uncertainty; flexibility is against simplicity and, in practice, invalidates the estimate and therefore planning. Choices must be removed whenever possible. All possible users can't be pleased at a reasonable cost, so development must be limited to current specific requirements

Developers must make choices at the right moment, when they have the needed information, not before. Meanwhile they should focus on things that require immediate attention.

And one of the most difficult decisions, as mentioned above, is the rejection of unlikely scenarios; implementation of not required features "in anticipation of". [8] sums it up in one sentence:

Don't waste time on problems you don't have yet.

Because lots of them won't appear and even those they do, developers will be forced to implement a much simpler solution without unnecessary ornaments.

3 WORKFLOW PRACTICES

3.1 User Stories

A User Story describes functionality that, by itself, delivers value to the user [11].

User stories are the building blocks of agile development. - [12].

They are a simple way to describe a concise task that adds value to the user or product. Initially, it is superficially specified (the first three points) and not detailed until its development, but at the end of its life cycle, consists of the following elements: ID (unique), Title, Description, Acceptance criteria/tests, Time estimation, Priority, Classification and External references.

Description is the main point, built (at least) with the following format:

"As [role], I want [funcionality] so that [benefit]"

Example: As [reader], I want [to see related articles] so that [I can find another interesting readings].

After the description is complete, there must be a conversation with the client about the story to clarify the details and set a priority. Then, acceptance criteria/tests should be established, so a developer can decide when a user story is completed. Acceptance criteria are much more efficient when they are automatic [13].

There are only to completeness states for a User Story: pending or done (acceptance tests are passed). There isn't something like a 99% completed user story.

Subsequently is required to estimate the time it will take to complete the user story and once defined, estimated and prioritized it will become part of the product backlog (Product Backlog will be described later).

If a User Story needs additional information or references, look for a place to include them, as it may be relevant later.

To create good stories a developer should focus on six attributes. A good story is INVEST [14]:

1.*Independent* – without dependencies on another user story.
2.*Negotiable* – can be changed until they are part of an iteration.
3.*Valuable* – must deliver value to the end user.
4.*Estimable* – in the time needed to complete.
5.*Small* – so can be planned with a certain level of certainty.
6.*Testable* – completeness can be assessed through tests.

If a User Story if too big to be estimated with precision or completed in an iteration, it is called *Epic*. They usually refer to big sections with several features and must be split in several INVEST stories.

3.1.1 Bad Practices

User Stories should be checked to avoid the following mistakes:

- Omit benefit ("…so that…") in the description, being that part which underscores the value it delivers to the user.
- Write down *how* to do instead of *what* to do.
- User stories are not requirements specification.
- A "user story" is not a "use case" because it does not focus on the how nor is it an exhaustive definition of the requirements.
- Absence or uncertainty in the acceptance criteria.
- Lack of estimations, because they may create false expectations and difficult self-discipline.

3.2 Estimation

It is necessary to evaluate how long it will take to complete a task.

Why go to all the trouble of working on a schedule if it's not going to be right? – [15]

The starting point in solving any task: a user story, an operation of infrastructure or even a planning is to determine the estimated time it will take to complete.

Delay the estimation spreads developers focus, increases uncertainty and because estimate compels to begin the process of resolution: forces to materialize the 'how' it will solve and identify risks.

It should be noted that only after a great experience and in very stable projects and circumstances it is possible to make big estimations. So it is best to divide tasks to estimate into smaller segments: if it is a user story, it may be too large to be implemented directly [16]. So it is too big to be estimated. Estimations should be made in hours, although many agile methodologies allows estimating in abstract concepts (*point estimates*) or in relative amounts [17].

It should be so, because the time actually spent on the resolution will be in hours, which are deducted from the time available to delivery. And it must be done because the estimate should be accompanied by a subsequent revision of the time spent and to compare two measures, these should be in the same units. Not to estimate in hours conceals that actually no one knows the estimate, perhaps because the task to estimate is too large, as we said before.

Estimation values can be any positive number, but is suggested to use the Fibonacci sequence (up to 21/34).

Whereas the estimate should be during working hours, an alternative is to establish a scale based on working days with the following values: ½, 1, 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40[1]. A task that takes more than a week's work should probably be fragmented (and, being more strict, any higher estimate of 20 hours must be studied with suspicion).

3.2.1 Alone Estimation

One of the challenges of product development by a single individual, which is the current scenario, is the estimation of the *user stories*, and not only because of the no proximity of the client in most development situations.

In situations of a single developer, there may be difficulties and risks unnoticed; there may be technical deficiencies that adversely affect the estimate. In such cases the only thing that helps is to measure the actual effort spent on each task and after each iteration compare estimations against actual effort and draw conclusions for future estimates.

3.3 Planning

Development will never end without a (short term) planning.

Fix Time and Budget, Flex Scope - [8]

Much of the success of a product or service is based on proper prioritization of work so the client receives greater value. For this, it is required that the different user stories of the product backlog are properly estimated.

First thing to do in planning is group user stories in *themes*. Then, prioritize the user stories in each theme, following client specifications until complete a *roadmap*.

Based on prioritization and estimates, the road map is divided into a set of milestones and deliverables, with roughly the same workload and oriented to regularly deliver stable releases. Releases are divided into internal and external, depending if it is communicated to the client or remain in development environment. Development cycles that are aimed at creating such deliveries are called iterations.

The developer should work in the closest milestone and respect the deadline. When the deadline ends, employed time will be reviewed, measuring the development speed and completed tasks will be assessed. Iterations must end on time, not when the tasks are completed and not yield to the temptation of delaying the deadlines. It is preferable to move the stories not completed to the next milestone. Agile projects must be flexible in scope and rigid in timing.

Each milestone should involve adding value to the product. This way the risk of not delivering anything is removed, the client can see the progress of the project and deviations can be corrected as they occur when the cost is still acceptable.

[1] Considering an 8-hour day. The practical checks reveal that 6 hours per day is a much more reasonable estimate for what the scale would be: ½, 1, 2, 3, 6, 9, 12, 18, 24 and 30

Milestones and releases should be maintained small enough to keep things in perspective and not to take the risk of employing a lot of time on features that may not be delivered on time.

Remaining work should be continuously prioritized according to changes or opportunities that arise in the business, along with the entry of new requirements requested by the client.

3.3.1 Bad Practices

Product builders should not react immediately to any request for new features and attend to work on it without evaluation, even interrupting the current job. In addition to disrupt development flow and cause delays, it may miss the objective of the project and customers or users not get satisfied.

3.4 Product Backlog

Where classified and prioritized list of tasks of development is placed: a control panel of the product status.

More than a practice, a Product Backlog is a tool, a mechanism to give the value and visibility it deserves to the current state of development of a project. It is done to get a perspective on what needs to be done and have clear customer priorities.

A typical product backlog contains a list of the following different types of elements [18]: *features* (as user stories), detected *bugs*, *technical work* and *needs of knowledge acquisitions*.

The features in the form of user stories make up most of the elements of a product backlog, with stories of higher priority being more detailed than those addressed below.

Product backlog must also include the other types of elements, although other authors have indicated otherwise [19], because they require estimation, planning and priorization, so that they must be included in the development process, and consequently in the backlog.

Within the product stack is necessary to classify items by priority and value for the customer, usually in: essential, important, interesting or optional. And should be developed in that order. They can also be grouped by subject [16].

3.4.1 Bad Practices

A product backlog is not a specification of requirements, either in scope or in its immobility. It is lighter than a requirements document and allows the customer to make changes during the life of the project. Developers can also make it evolve during development.

3.5 Automatic Tests

Test everything, over and over. Better automatically.

Unit tests are one of the corner stones of eXtreme Programming (XP). […] You will also create your tests first, before the code. - eXtreme Programming

Untested *software* has errors. It is a fact. Each implemented feature, each completed user story shall be tested (and it is in the client acceptance contract). But as the tests are tedious, repetitive and prone to errors if executed by an individual, you should always lie to automate as much as possible. Implement an automatic test is hard, but it will be executed a hundred times, so worth the time invested.

Tests should be performed as soon as possible. In fact, it is very common in Agile integrating TDD (test-driven development), in which the tests are implemented before the code itself. When the tests are created before the code, the developer is concealed to consider what needs to be developed, reflecting the requirements

of the relevant tests [6]. Then create the code that implements the functionality is much faster and easier, no unnecessary code is created or not required features included (supporting simplicity, and with a little more time).

Moreover, passed tests reflect the current completeness status of the feature in development.

However, TDD is not always easy, so it is not integrated as a practice in this methodology. The procedure can be testing after coding in some sections and then make tests before in subsequently similar scenarios. Before or after coding, to make automatic tests never is a bad practice and they must be included with delivered source code.

3.6 Version Control System

Version or revision control is a fundamental practice in any serious software development process, not only in agile methodology. And not only for a team, also to a lonely developer:

- Automatic backup.
- Allows maintenance over several versions.
- Enables code comparisons.
- Allows returning to a stable point after a serious bug.
- Allows reviewing the history of code fragments.
- Developers can safely experiment (reverting changes if desired result is not achieved).
- Keeps code clean, without commented old code.
- Helps bug finding.
- Allows controlling advances in implemented code.

3.7 Re-evaluation

Watch, measure and evaluate your work.

The work done must be continuously assessed at all levels. Once a user story is completed, acceptance testing assesses their correctness but should also be checked manually to see if it is performed in the simplest way (remembering that something more understandable it is also simpler), following good practices and in the most effective way. If not, maybe it's time to refactor.

It is also necessary to assess whether the estimate was correct and whether the prioritization and integration into the product backlog has been adequate.

Agile methodologies include continuous improvement after every release or at the end of an iteration. The advantage for a single developer is reassessment can be made every time and over every aspect of the development, taking notes and making improvements.

3.7.1 Measure

For a right re-evaluation adequate measuring is a must. As much as possible should be measured, and done through specialized tools (and better automated). It is the method for strength and consistency in measurements.

The following should be measured:

- Time invested in each task in the product backlog.
- Percentage of code with automated tests (*code coverage*).
- Percentage of the completed product.
- Code duplicity and other error tags.

4 AUXILIARY PRACTICES

The following practices are highly recommended, but may not apply to all workplaces, at least at the beginning of the implantation of this methodology.

4.1 Refactoring

Continuous improvement of the design of existing code without changing the fundamental behavior.

Add a test, make it work, make it clean - Martin Fowler *Premature optimization is the root of all evil.- [20]*.

When combined with rigorous self-testing and automation, refactoring becomes a powerful mechanism to ensure the best possible product. Moreover, it is the fastest way to reduce technical debt [21].

Uncompleted user stories should be implemented in a solvent manner, but initially making the minimum code needed to pass the acceptance tests. Once passed resulting code must be refactorized.

Refactoring techniques are the application of good programming practices and further, removing anti-patterns and *code smells*. Detail them is beyond the scope of this text so the reader is referred to reference works: [22] or [23] and high value articles like [24].

Finally, do not refactor unless time investment will allow faster development in future work.

4.1.1 Bad Practices

Refactoring is improving the quality of the code without changing behavior; so it should not be used to provide scalability, generalize interfaces, etc.

There is a thin line between properly understood refactoring and premature optimization, considered in many cases as an anti-pattern, or at least, as two practices that should not be mixed or confused [22].

Code should not be reviewed over and over looking for refactorizations to make. Finished code should be reviewed only by his relationship with another fragment freshly made (to apply improved class inheritance, avoid duplication of code, etc.).

In general, there are not metrics to decide when to refactor or not. It is something assigned to the experience and intuition of the programmer [25].

4.2 Limited Documentation

Document, but only what is necessary.
Working software over comprehensive documentation. – Agile Manifesto *[26]*.

In no way should be inferred documentation interferes: user stories, well developed, are part of the documentation (a non-renounceable part). Is good to have an entity-relationship diagram or an equivalent representation of the data model; an API usable by a third party must have inexcusably a complete and thorough documentation. Mind maps and other graphical representations of ideas and procedures are favoured.

Documentation should be as close as possible to the code: at the method or classes header; in the database management tools; in a URL nested to the domain of the product (for APIs, for example); a wiki accessible directly from the control version system; in a subdirectory of the root of the source code of the project, etc.

Documentation utilization must justify the associated effort to develop it. The read/write proportion should be 1 to 100. IDE and tools frameworks often include automatic documentation generation that streamline creation and maintenance, worth taking the time to learn to use them.

Not adopting this practice means the developer is documenting in any other way, never that there is no documentation.

4.3 Partial Prototypes

A partial prototype is a very simple program to explore potential solutions.

When a technical difficulty threatens to hold up the system's development, put a pair of developers on the problem for a week or two and reduce the potential risk. - eXtreme Programming.

It is convenient creating partial prototypes to clarify technical or design problems or to increase the reliability of estimates of user stories. They should focus only on the problem under study and ignore anything else. Most partial prototypes will not have the quality to be integrated into the final product, even partially, so that should be developed with the idea of later disposal [27].

Partial prototypes must be included in the product backlog as an additional task to be performed, with their estimation and prioritization. Conditions of acceptance can be omitted.

4.4 Rubber Duck

Highly effective and very low cost debugging technique. It involves following four steps:

1. Obtain by any mean (preferably legal) a rubber duck.
2. Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.
3. Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line
4. At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

It works in any case [28].

4.5 Automation

Whatever can be automated should be automated as soon as possible.

When it is expected to have tasks to be repeated any number of times, they should be automated and this automation should be done as soon as possible or all the time spent on doing them manually will be lost time. The more the project advances, the more difficult it will be automate from scratch or less time will be available for it.

Furthermore, it is possible that many of the automated tasks are usable in other taks, incorporated into other projects, etc. so the time spent will still be useful even after the project for which they were created ended.

5 FAAT IN PRACTICE

There is no Royal Road to Geometry - Euclides to King Ptolomeo about an easy way to learn mathematics.

The best way to deal with the adoption of a methodology is with the help of a mentor, a person outside the team with a deep understanding of agile methodologies to plan, direct and control the implementation. The problem in the case of a developer alone is not usually having the resources to hire an outsider to lead him. For these situations, this chapter will attempt to expose a process of "self-implementation" of FaaT.

The implementation consists of three phases: knowledge, implementation and evaluation acting on three factors, which in turn influence the development of each phase:

- Current *processes*, practices and procedures that define the way the developer works.
- *Tools* the developer works with.
- *People*, two in the FaaT scenario: developer and client.

The client of a freelance developer uses to actually be several clients over whom it has little or no influence, so their role in the methodology has been consciously minimized, voluntarily assuming the risks it implies. On the other side, the developer assumes all other roles involved.

5.1 Workflow

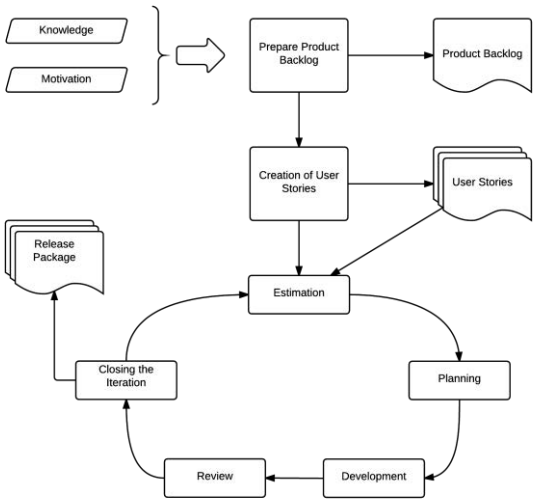

Figure 1 – FaaT's workflow

1. Prepare Product Backlog.
2. Creation of User Stories.
3. Estimation.
4. Planning.
5. Development.
6. Review.
7. Closing the iteration.

In the workflow described in Figure 1 a large percentage of the practices of this methodology are represented, but it is important that the developer has in mind others, vital to the quality of the result, especially strategic practices: *Simplicity*, *Embrace change*, *Make decisions*, *Refactoring* and *Version Control System*.

5.2 Knowledge and Motivation

The stage of knowledge means understanding the methodology to apply: getting an overview of the components and their relationships and their objectives and their implications, returning back to this chapter during implementation and reading some of the related articles and other valuable sources of knowledge in agile practices.

In addition, this stage involves a strong motivation to take the effort by a single person. The process implies a more complex management in which errors and shortcomings will arise.

The last part is the election of the first project in which to apply the methodology, which ideally will be: as small as possible, uncritical because there is a performance decrement [29], known or similar to others done before and from a friendly client.

5.3 Implantation

Faat does not pretend to be prefect, so following there are not impositions but "initiation guidelines" trying to turn a set of abstract practices in an efficient way to develop software. A certain numbers of tools and applications are mentioned through this chapter. They are just tools that the author used and that have been helpful, but the reader is encouraged to look for on its own the best suited.

All of the methodology processes involve the dedication of time by the user. Furthermore, the time spent on tasks is the input into the estimation, influences planning and is essential in the review practice. It is important to measure the time spent.

The author has been using Toggl (https://www.toggl.com) for more than 18 months at the moment of writing this document and he keeps using it because of its easiness, being available in as a web and mobile app, its powerful reports and its integration with other tools. Moreover, it is free for an individual developer.

5.3.1 Prepare Product Backlog

The central tool of the methodology is the product backlog, which is the "dashboard" of our development. Digital or physical, it should be as close as possible to a whiteboard, where you can write and erase, drop or paste notes or cards, which must also be groupable and distributable in at least two dimensions.

Physical boards have some advantages: are more communicative, operations performed on it are more immediate, are always visible (if the workspace is fixed) and are much more versatile for the grouping and placement of elements. But digital boards provides:

- Portability, essential if working in different locations, very common for freelancers.
- Having more than one at the same time is possible, essential for simultaneous projects.
- Dynamic elements such as graphics or links can be incorporated.
- Log of changes and notifications.
- Automatic calculations (e.g. sum of hours).
- Searches, filters, duplication and other standard features in digital tools.

Again, the reader has a lot of choices [30] but give a try to Taiga (https://taiga.io/) or Trello (https://trello.com/), more generic and flexible, so more suitable for a single developer.

Product backlog, as seen in the practice must include all the tasks to be done until the complete development of the product. Each one of those tasks will have a card, classified by:

- *State*, with at least the values shown in Figure 2: *To do*, *Doing*, *To be reviewed* and *DONE*. Maybe you will need an additional state for unready tasks (*inbox*/pending/etc.).

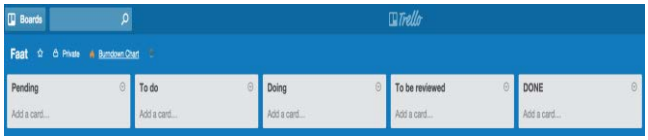

Figure 2 – States using columns in Trello

- *Type*: user story, technical work, knowledge acquisition need and bugs.
- *Priority*, for the client (essential, important, interesting or optional).
- *Theme* (optional).

In Trello you should create columns for state (and maybe theme) classification and labels for the rest.

Also, an additional column must be created for each release (as it will be described in next chapters).

5.3.2 Creation of User Stories

This step is one of the most important in the whole procedure; it's time to clarify everything the product will offer, to list all the operations that users can perform.

User stories are written in what we will call from now on cards, in a simple and concise way.

The first step is to identify the users who will use the product, identify roles or *personas* that will solve a need through the application. Then, in the name of those *personas*, we will create *epic stories*.

The *epics* are large and ill defined: with no more than an identifier, a title, a description and priority for the client. They will allow sketching product functionality without getting lost in the details and establish roughly the scope of the work to be performed.

But *epics* are not INVEST (see the proper practice) and they must be divided in smaller, simpler, achievable and estimable user stories. Do not divide all epic stories right now, only enough to fill an iteration (see later), because at this point there is still much uncertainty about the product. Thus the product backlog is kept clean (with the righteous detailed elements) and so it is easier to integrate new knowledge arising from exposure of previous releases to clients and users [31].

The different splitting techniques are beyond the scope of this document; however there are several patterns of great interest to apply presented in [32] the reader can read. Just an example:

Epic: As an administrator *I want* to manage the videos uploaded by the users *so that* I can control the publishing flow.

User story 1: As an administrator *I want* to see the list of videos uploaded *so that I can* check author, duration and state.

User story 2: As an administrator *I want* to see an uploaded video *so that I can* assess their propriety and quality.

User story 3: As an administrator *I want* to be able to publish a video *so that* every visitor can see it in the website.

Then the resulting user stories should be detailed, so that all relevant information is visible and accessible. Clarity, organization and immediacy are the premises to follow. At this point you're likely to raise doubts, so the client's presence is welcome; if not possible, the developer, in its role of *Product Owner* must extrapolate the relevant information or record the questions for asks the client later.

First fill id, type, title and description. Then priority for the client, acceptance tests (between 3 and 5 on average) and theme. A completed card also can include a checklist, an activity log or a proposed deadline. After a card is fully written move it from "inbox/pending" column to "to do" column in the backlog.

Remember that not all the tasks must be fulfilled right now, only a set big enough to complete an iteration. Once again should be said, although the user stories are the most important task to be included as cards in the product backlog, also it is needed at this point to create and detail the tasks of technical work and the needs of knowledge acquisition.

5.3.3 Estimation

The end result of the above process should be a product backlog with a good set of detailed tasks in their respective cards, properly classified and placed in the column "to do" or in their theme column if they have been created. In each column, cards must be sorted by priority with more important cards on top.

From now on the cards must be estimated using the guides and detailed recommendations in the related practice. Again as in the previous section it is not necessary to estimate all tasks, but enough of them to have material for planning a release.

In the case of study, the author used Scrum for Trello as a support tool for estimation. Scrum for Trello is a browser

extension that allows Trello register time estimates of user stories and also the time spent on its realization. In addition, the assignable values are configurable.

5.3.4 Planning

Once the more priority cards are in the product backlog with a sufficient level of detail and a preliminary estimate of the work required (modified with feedback from previous iterations if it is not the first), it's time to plan the next iteration or *sprint*.

Three points must be set: duration, term and list of tasks to do.

For an individual developer, sprints should last one or at most two weeks, but the exact duration can be conditioned by the objective that provides more value to client.

Establishing, as recommended above, 30 hours per week and considering that in most situations freelancers develop several projects simultaneously, a sprint of two weeks should not contain tasks that sum more than 32-35 estimated hours.

With customer priorities in mind an objective for the iteration is assigned: a phrase that summarizes the new value the product and ultimately the client will get [33]. No need to write, just keep it present when selecting tasks.

- *"Activities and anniversaries calendar support"*
- *"News administration and publishing flow"*

On rare occasions the objective can add value to the product without it being detectable by the customer.

- *"Refactoring and doc improvement week."*
- *"Testing and bug fixing 3-day-row"*

In the product backlog a new column is created for the iteration, with the deadline, the objective (if possible) and a name. Then move cards related to the objective to that column, starting for the most important (those placed at the top of their columns) until the sum of the estimated time of the moved cards reach the hours of work assigned to the iteration (roughly). Scrum for Trello will show the iteration estimated time (with the sum of cards estimations).

If an iteration is not populated enough with the assigned tasks, one or more tasks of knowledge acquisitions needs must be included. The risk of deviations in the planning is reduced when decreasing the uncertainty [29] so that incorporating these tasks in the very first iterations is a good practice.

5.3.5 Development

This phase involves completing each of the tasks assigned to the iteration, in the order in which they were prioritized in the previous phases. How to perform each task is beyond the scope of this document, but it is advisable to follow a common set of guidelines.

The first is that all the developed code must be under a version control system as stated in the practice with the same name. There are great alternatives available but Git in particular manages specially well with the creation and merging of branches, a useful feature in our workflow.

During the twenty months prior to the writing of this text, the author has been working, with very satisfactory results, following a version control flow described in [10][2].

Once a user story's development has been started create the tests that the code must pass to be considered complete. During this creation, the solution will start to grow in the background. Next, develop the code needed to provide the required functionality and to pass the tests. At last, refactor the resulting code and build up the related documentation. In summary [22]:

Add a test, make it work, make it clean.[3].

Before finishing the task, review it again, check its quality, write down the actual time spent in its resolution in the card and move it to the "to be reviewed" column.

Several practices have been mentioned in this step, but do not forget to follow the principle of simplicity and make decisions.

5.3.6 Review

When all the tasks of the iteration are done (or the deadline arrived), the material delivered should be reviewed, both quantitatively and qualitatively and it compared with the product developed to date in the previous iterations.

Planned in the iteration but uncompleted user stories must be checked, asking for the reason why they have not been completed on time and moving them back to the "to do" column (at the top, with high priority).

Another section to check is the speed achieved in the iteration, comparing the estimated hours with the actual time spent. This value is used to refine estimates of other pending user stories and to estimate the amount of work that is reasonable to assume for the next sprint.

It is also necessary to pay attention to the quality of the code developed, but as it is not possible peer review, the freelancer must rely on markers that some tools can provide. These tools do not ensure quality, but can reveal problems and errors. They provide some metrics calculated from the source code executing (sometimes automatically) some scripts.

The case study is developed in PHP (the server side) and AngularJS (the web client) so that mentioned tools are specific to these technologies but learn what they do and how. There are equivalent tools for the major programming languages.

1. PHPUnit https://phpunit.de/
2. PHPMetrics http://www.phpmetrics.org/
3. Plato https://github.com/es-analysis/plato
4. SensiolabsInsight https://insight.sensiolabs.com

This tools provides quantifications on *code coverage* of the tests, *technical debt*, too big classes, code *complexity* and *maintainability*, *dependencies* between classes, *accessibility* for new developers, *simplicity*, *error-prone* code, etc.

5.3.7 Closing the Iteration

After reviewing the developed functionality is the time of completion of the current iteration; the time of making and deploying a release. The source code is tagged in the version control system and prepared for its delivery, its installation in the server or other appropriate operations.

It is time to start over from the stage 3: Estimation.

5.4 Evaluation

Simultaneously to the workflow processes of the methodology, the other practices and the development itself, the freelance must find time to decide whether on the basis of the known, learned and implemented, should improve the use of the methodology, adopting additional practices, limit current, etc.

[2] It is simple and it works. But it is not the only one who has been successfully applied; in [34] you can find other alternatives that deserve evaluation.

[3] Some authors will add "make it fast", but in that case be careful not to fall into premature optimization.

This requires analysing the information collected and questioning whether there were improvements in software quality or performance, development process, developer productivity and the image the freelance projects to the market.

In addition, during the evaluation stage is time to consider if there are repeating operations in each or multiple projects and that can be automated in one of them and exploited in the next.

6 CONCLUSIONS

It has been developed an agile methodology, Faat for software development alone, a scenario with special circumstances and on which there are few studies and a large room for improvement.

Faat provides a controlled framework in which a single developer can improve the quality of products delivered to the customer, lowering risks and increasing performance. It has been tested successfully on real projects for e-learning via web.

It is also an *intended for use* system, which renounces to purisms to gain flexibility and applicability and provide immediate results. It can be applied without outside mentoring, with little interference in the freelance daily routine and extensible to more general scenarios.

7 REFERENCES

[1] Dzhurov, Yani, Krasteva, Iva, and Ilieva, Sylvia. Personal Extreme Programming – An Agile Process for Autonomous Developers. *International Conference SOFTWARE, SERVICES & SEMANTIC TECHNOLOGIES (S3T)* (Oct. 28, 2009), 252-259.

[2] Agarwal, Ravikant and Umphress, David. Extreme programming for a single person team. *Proceedings of the 46th Annual Southeast Regional Conference on XX (ACM-SE 46)* (2008), 82-87.

[3] Hollar, Ashby Brooks. Cowboy: An Agile Programming Methodology for a Solo Programmer. *VCU Theses and Dissertations* (2006), 741. http://scholarscompass.vcu.edu/etd/741.

[4] Dodero, J.M., García-Peñalvo, F.J., González, C., Moreno-Ger, P., Redondo, M.Á., Sarasa, A., and Sierra, J.L. Development of E-Learning Solutions: Different Approaches, a Common Mission. *IEEE Revista Iberoamericana de Tecnologías del Aprendizaje (IEEE RITA)*, 9, 2 (2014), 72-80. http://dx.doi.org/10.1109/RITA.2014.2317532.

[5] García-Peñalvo, F.J. and Seoane-Pardo, A.M. Una revisión actualizada del concepto de eLearning. Décimo Aniversario. *Education in the Knowledge Society (EKS)*, 16, 1 (Mar. 2015), 119-144. http://dx.doi.org/10.14201/eks2015161119144.

[6] Wells, Don. (1999) *The Rules of Extreme Programming*. [Accessed: 2015-5-23] http://extremeprogramming.org/

[7] Letelier, Patricio. (2014) *Agile Roadmap*. [Accessed: 2015-2-10] http://agile-roadmap.tuneupprocess.com/

[8] 37Signals. (2006) *Getting Real*.

[9] Beck, Ken. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, 1999.

[10] Driessen, Vincent. (2010, Jan.) *A successful Git branching model*. [Accessed: 2014-10-14] http://nvie.com/posts/a-successful-git-branching-model/

[11] Beas, José Manuel. (2011) *Historias de usuario*. [Accessed: 2014-11-08] http://jmbeas.es/guias/historias-de-usuario/

[12] Emerson, Maria. (2012, Jan.) *Writing Good User Stories*. [Accessed: 2015-3-15] http://mariaemerson.com/user-stories/

[13] Paredes, Adrián. (2008, July) *User Stories*. [Accessed: 2014-11-2] http://elblogdelfrasco.blogspot.com.es/2008/07/user-stories.html

[14] Cohn, Mike. (2004) *User Stories Applied*. https://www.mountaingoatsoftware.com/system/asset/file/259/User-Stories-Applied-Mike-Cohn.pdf

[15] Spolsky, Joel. (2007, Oct.) *Evidence Based Scheduling*. [Accessed: 2015-1-31] http://joelonsoftware.com/items/2007/10/26.html

[16] Cohn, Mike. (2014, Mar.) *Agile User Stories, Epics and Themes*. [Accessed: 2015-3-29] https://www.scrumalliance.org/community/spotlight/mike-cohn/march-2014/agile-user-stories-epics-and-themes

[17] James, Michael. (2008, Nov.) *Scrum effort estimation and story points*. [Accessed: 2015-5-02] http://scrummethodology.com/scrum-effort-estimation-and-story-points/

[18] Cohn, Mike. (2000) *Scrum Product Backlog*. [Accessed: 2014-11-15] https://mountaingoatsoftware.com/agile/scrum/product-backlog

[19] Beas, José Manuel. (2011) *Product Backlog*. [Accessed: 2015-4-20] http://jmbeas.es/guias/product-backlog/

[20] Knuth, Donald E. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6, 4 (Dec. 1974), 261-301.

[21] Leffingwell, Dean et al. (2014, July) *Refactors*. [Accessed: 2015-3-20] http://scaledagileframework.com/refactors/

[22] Fowler, Martin and Beck, Ken. *Refactoring: Improving the Design of Existing Code Hardcover – July 8, 1999*. Addison-Wesley, Chicago, 1999.

[23] Wake, William C. *Refactoring Workbook*. Addison-Wesley, 2003.

[24] Shore, James. (2010) *Refactoring*. [Accessed: 2015-5-12] http://www.jamesshore.com/Agile-Book/refactoring.html

[25] Beck, Ken and Fowler, Martin. (2006) *Code Smells*. [Accessed: 2015-5-13] https://sourcemaking.com/refactoring/bad-smells-in-code

[26] Beck, Ken, Beedle, Mike, van Bennekum, Arie et al. (2001) *Manifesto for Agile Software Development*. [Accessed: 2014-10-1] http://agilemanifesto.org/

[27] Cook, Charles. (2009, Feb.) *Spike*. [Accessed: 2015-4-24] http://www.cookcomputing.com/blog/archives/000588.html

[28] Errington, Andrew. (2002) *Rubber duck debugging*. [Accessed: 2014-10-25] http://rubberduckdebugging.com/

[29] McConnell, Steve. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmon, 1996.

[30] Garzas, Javier, Enríquez de S., Juan A., Irrazábal, Emanuel. *Gestión Ágil de Proyectos Software*. Madrid, 2013.

[31] Pichler, Roman. (2014, Aug.) *From Personas to User Stories*. [Accessed: 2014-9-12] http://romanpichler.com/blog/personas-epics-user-stories/

[32] Lawrence, Richard. (2009, Oct.) *Patterns for Splitting User Stories*. [Accessed: 2014-10-26] http://agileforall.com/2009/10/patterns-for-splitting-user-stories/

[33] ScrumManager. (2014, Apr.) *Planificación del Sprint*. [Accessed: 2014-12-20] http://scrummanager.net/bok/index.php?oldid=971

[34] Christensen, Spencer. (2014) *Git Workflows That Work*. [Accessed: 2015-5-4] http://blog.endpoint.com/2014/05/git-workflows-that-work.html