# What every Product Owner should know

ScrumSense

agile coaching and training

# Contents

# Introduction

Agile software development and Scrum has seen explosive growth since the signing of the Agile Manifesto in 2001. Yet the understanding of what the core values, principles and practises continue to be debated and discussed.

In my own growth as a coach I have had the opportunity to work with many teams and their product owners. Reflecting on my experience and analysing the Scrum framework and the challenges associated with its implementation has led me to the belief that the Product Owner role is one of the core innovations within the framework. Some have described the Product Owner role as a single point of failure. This has been reflected in my coaching engagements. Frequently I find that identifying the correct person for the Product Owner role and empowering them is a significant impediment to the implementation of good Scrum.

If we contrast the traditional project manager role with that of the Product Owner what stands out is the emphasis on empowering a single individual with the authority to decide both scope and schedule. The project manager is usually forced to emphasise the cost of change and limit it to de-risk the project. This is in stark contrast to the Product Owner who must be empowered to decide and influence scope through an understanding of the domain, value and priority of requirements.

The implementation of basic Scrum practises will likely result in a benefit to the business through the mere mechanics of the framework. Focus, commitment and openness through practises like stand ups, planning, sprint goals and sprint burndowns are easy to understand and usually not contentious. But it is unlikely that this rote implementation of the framework will result in the “hyper-performance” that Jeff Sutherland speaks of in his presentations and papers.

For this level of performance, we need something else. The Product Owner role is the key to unlocking this level of performance. Their collaboration with their customers and team allow for better decisions to be made.

A good ScrumMaster can take the team to the point of being self-organised and motivated; they can help the team achieve good Scrum. But to achieve truly great Scrum, an empowered, dedicated Product Owner with a strong vision is crucial.

# The Product Owner Role

## The Vision

*The vision should communicate the essence of the future product in a concise manner and describe a shared goal that provides direction but is broad enough to facilitate creativity.*
*- Roman Pichler*

At the core of what powers agile software development, is the principle of self-organisation. Self-organisation occurs whether or not we intend it to. What is critical to success however, is whether the self-organisation occurs with the desired direction or intention. If the Scrum Team is to be successful, they must be aligned and the common alignment is achieved through the Product Vision.

If you worked in corporates through the late 90's you probably have "vision fatigue". At one time, it seemed like everyone was punting the importance of vision through roadshows and weekends in the bush. This vision is not like that.

A significant distinguishing characteristic is that this vision should be arrived at through collaboration between the Scrum Team, customers and users. This inclusive process is vital in ensuring alignment and understanding of the vision. Peter Senge said, "A shared vision isn't an idea. It's a force in people's hearts.

An effective product vision should describe your target customer or user and the value proposition for that customer/user. It should be short and memorable and pass the "elevator test".

A common technique for creating this vision is "The Product Vision Box" which asks teams to come up with the copy for a box in which their product could ship. This includes the key features which drive purchase and what customers would find compelling. (See Luke Hohman's  Innovation Games for more info) The key constraint in the sizing of the box helps the team to select the 4 or 5 bullet points that constitute the selling points. This is a small team, consensus driven activity.

An alternative comes from Geoffrey Moore's book Crossing the Chasm. It follows the form:

- For (target customer)
- Who (statement of the need or opportunity)
- The (product name) is a (product category)
- That (key benefit, compelling reason to buy)
- Unlike (primary competitive alternative)
- Our product (statement of primary differentiation)

For example, the iPod's vision statement might read as:

- For music lovers
- Who want their music on the go
- The iPod is an MP3 Player
- That offers intuitive playback functionality
- Unlike Sandisk Sansa
- Our product is easy to use and synchs with our integrated music library iTunes

Note, the visioning process should not be a long running one. When starting new Scrum teams, we typically spend 4 hours in building the vision.

## Typical Mistakes in Product Visioning

### No vision

This leads to what is sometimes called "feature soup". Without a strong guiding principle of what they key customer and their needs from the product, the temptation is to build a product based on pleasing all potential customers and needs. This checklist driven approach results in product bloat. The key value unlocked through an iterative incremental approach to product development, is not building the things you don't need. Code that is not written does not produce bugs and does not need to be maintained.

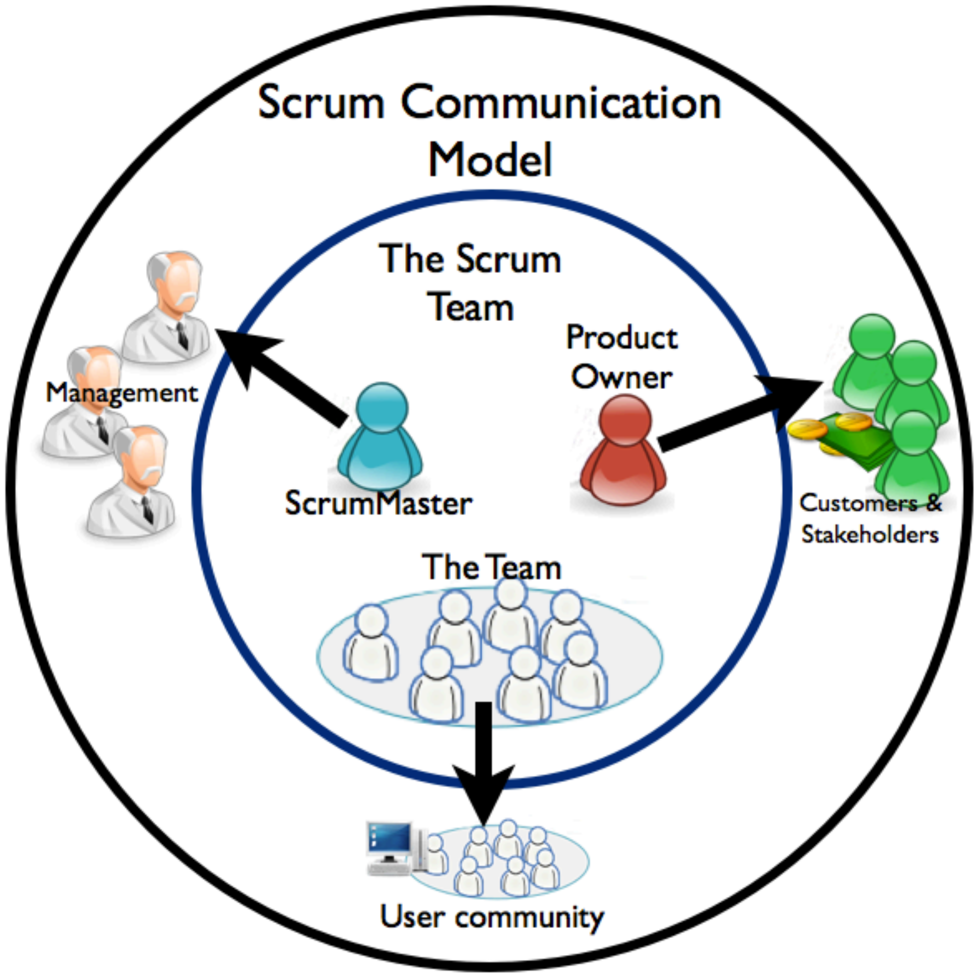
This modern approach to product development is exemplified by the 37Signals group and their online book “Getting Real”. In particular read: “Build Less”

From the essay:

> *“Conventional wisdom says that to beat your competitors you need to one-up them. If they have four features, you need five (or 15, or 25). If they're spending x, you need to spend xx. If they have 20, you need 30.*
> *This sort of one-upping Cold War mentality is a dead-end. It's an expensive, defensive, and paranoid way of building products. Defensive, paranoid companies can't think ahead, they can only think behind. They don't lead, they follow.”*

**The Shopping List**
This vision is nothing except a list of features that the Product Owner wants, with no unifying principle or clear goal. This does nothing to guide the team (or inspire the team). It also rarely passes the elevator test and usually is also difficult to remember. What it adds up to is that it does nothing to align the team or help them make decisions, and as a result the “secret sauce” in Scrum (self-organisation) never emerges or emerges misdirected.

## Scrum Communication Model



Scrum attempts to simplify the often confusing web of relationships through defining a clear set of roles and responsibilities.

Within the inner circle above, lies the Scrum Team. This includes the Product Owner, ScrumMaster and the Team. Within this boundary there should be no barriers to communication. This team is collectively held accountable for the success of the product, and there should be an open dialogue between all its members.

The outward focus of the Scrum team's communication is also illustrated above. The Product Owner's main focus of communication should be with customers (who pay the bill) and other stakeholders who may have an interest in the product. Ensuring that their voices are heeded in the development of the product is vital to ensure the successful outcome.

Lyssa Adkins in her recent book "Coaching Agile Teams" states "all the real benefits of agile are realised through collaboration". The Product Owner's collaboration with customers, stakeholders and the team are the key to unlocking this value.

Through the interactions within this group we can better understand what to build, but even more importantly, when to *stop* building functionality. The value of code not written is difficult to measure but undeniable; we have no need to build, maintain, debug, test or release this code.

The Team should as far as possible communicate directly with the user community. Sometimes this may be the same as the customer (if you are dealing with a direct customer facing product). However, where this is not the case (as in many enterprise scale applications) having the Product Owner as a proxy or conduit for information from users is not advised. Face-to-face interaction (or at least direct communication) between the user community and the Team reduces the likelihood for miscommunication.

It is often the case that the ScrumMaster might interpret their mandate to "protect the team" over-zealously, and attempt to limit or eliminate interaction between the Team and users. However, this information is essential to the team. The ScrumMaster should be protecting the team from interruptions, not information.

## Working with the team

If self-organisation is the secret sauce of agile, collaboration is how we create value. Collaboration between the Product Owner and team is the vital intersection between what Tobias Meyer calls the "voice of the WHAT" and the "tribe of the HOW". This critical division of responsibility allows a tension to create the right environment for the best solutions to emerge.

In the day-to-day sprint activities it is vital that the Product Owner is available to answer the team's questions and requests for clarification. In particular, as teams mature this demand on the Product Owner's time will accelerate. Mike Cohn in his book "Succeeding with Agile" draws a compelling diagram of this acceleration seen below:

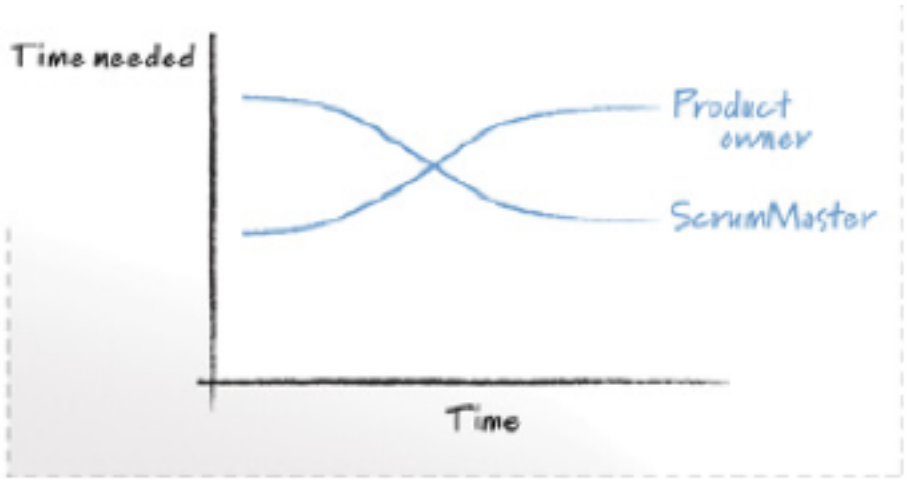


Image credit: Mike Cohn

As teams mature their need for intervention from the ScrumMaster diminishes (as impediments are removed and the team becomes more self-organising), while their demands on the Product Owner increase.

Serge Beaumont, an Agile Coach for Xebia states: "*Implementing Agile is like Pac-Man eating a power dot: the tables are turned, and they consume the backlog faster than a Product Owner can fill it up. I've seen on all my assignments: it is almost a law of nature for this to happen...*"

The ability for the team to know, rather than assume, removes the possibility of creating waste. And to ensure that this happens the team must feel that the Product Owner is there to support them by providing the needed guidance. If the Product Owner cannot answer their question, his immediate priority should be getting the answer. A good metric to aim for as Product Owner is that 85% of the team's queries should be answered in 15 minutes or less. You'll quickly realise that to accomplish this, requires co-location with the team.

**Definition of Done**
One of the main ways the team and Product Owner set the basis for their collaboration is through a negotiated definition of done. Setting this definition without input from the Product Owner can lead to a stifling of progress as the team strives for a high benchmark. This cannot be done in in isolation from the economic realities. On the other hand, many of the problems in our industry arise out of a lack of concern for quality, and the definition of done is the primary mechanism for ensuring quality within the Scrum framework.

**Sprint Retrospective**
In the past there has been some discussion as to whether the Product Owner should be in the retrospective. So to clarify, the Product Owner is invited to attend the session. This is part of the ScrumMaster's attempt to create a safe space for the discussion. It is expected that at times the discussion will hinge on the Product Owner's responsibilities and thus the Product Owner should be there to participate and collaborate.

**Backlog Grooming**
The Product Owner and team should be meeting at least once per sprint to prepare for future sprints. This Backlog Grooming session can also include members of the user

community. This group should collaborate in writing the stories for future sprints (including decomposing larger stories into smaller ones).

As mentioned earlier, this collaboration is how we unlock value in a Scrum environment. The ability to draw on multiple viewpoints and ideas will help to create better stories. This is in stark contrast to the model of Product Owner as visionary; writing stories in isolation. This moves the model away from the Product Owner arriving at this meeting with a sheaf of stories for estimation towards a joint story writing workshop.

Note: it is desirable for the team to estimate in this meeting as well, rather than estimating during sprint planning. By separating the events where estimation and commitment occur, we make clear that these two are are separated conceptually as well.

## The Sprint Review

A good sprint review is an essential part of the "inspect & adapt" cycle. While the ScrumMaster (as master facilitator) should be facilitating this meeting, this is a crucial meeting for the Product Owner. The focus of the meeting in the product under review.

The review should be an informal workshop which covers the following topics:

- Review of the Sprint Goal vs. the Sprint outcome
- Demo of functionality delivered
    - focus on its "fit for purpose"
    - how the product could be improved
- Discussion of challenges
    - Technical debt encountered
    - Impact of external factors (interruptions etc.)
- What (if any) impact on the current priorities (or vision) from functionality delivered
- Choose the next Sprint Goal

In reviewing the product increment delivered it is critical that only "Done-Done" functionality is accepted and demonstrated. This is vital in ensuring the expectations are managed with stakeholders and also allows for better fidelity in planning. Teams only get velocity credit for stories completed in a sprint.

The vital contribution of the Sprint Review is in the collaboration between the team, the Product Owner and the stakeholders present. To ensure that this happens it is important that the Product Owner manages the stakeholder expectations in preparation for the Sprint Review.

It is not unusual, in fact it is expected, that new requirements will emerge during the review. This should not be dealt with harshly (as scope creep). This is a vital part of the process; only when we see working software in operation do the requirements emerge. Make sure to capture these new stories and manage stakeholder expectations by indicating when they could expect to see these items delivered.

Lastly, by choosing a sprint goal in the Sprint Review we avoid the "Sprint Shopping List" as a goal. This becomes a statement of intention for the sprint created with the help of the team and stakeholders.

As Product Owner, make sure to thank the team if you're happy with the sprint outcome. Also, express your disappointment if that is the case. Openness is another core Scrum value. Be wary of singling out individual effort (or lack thereof). It is critical that the accountability is levelled at the team; so praise the team for their team work.

### The Sprint Goal

A good sprint goal provides the same guiding effect that we observe from a good product vision. It provides context for the team, allowing them to make good decisions when executing on their commitment.

It also helps the Product Owner focus by providing a need for the Product Owner to have a strong theme or goal. Merely re-stating the stories committed to in Sprint Planning will not be sufficient to provide this focus. It requires a grasp of the business value and the ability to communicate this as a coherent goal on the tactical level of the sprint. An inability to articulate a sprint goal is usually symptomatic of poor product vision. This is "feature soup" at the sprint level: "story soup".

A good sprint goal also provides the team with some "wiggle room" in terms of meeting their commitment. If the team runs into trouble during the sprint, examine their ability to deliver in light of the sprint goal to see if there is room to de-scope items without affecting the goal.

Finally, the sprint goal is also used as an acid-test of whether a sprint should be terminated. Termination is a drastic and hopefully infrequent event in most scrum team's lives. If however the environment or assumptions for choosing the sprint goal no longer match the reality, this could be cause for the Product Owner to choose to terminate.

Note; the inability of the team to deliver on its full commitment is not grounds for termination. A burndown graph indicating incomplete delivery is a trigger for a conversation between the team and Product Owner to negotiate how the committed increment can still be delivered (see "wiggle room" above).

Sprint termination leads to a new sprint cycle (starting with the review and retrospective of the terminated sprint).

## Story Writing Principles

### Definition of Ready
Jeff Sutherland's 2009 paper "Are you ready-ready to be done-done?" gave the Scrum framework a new concept which is of significant value. It is based on the team's "Definition of Done" and applies to the readiness of the story to be taken into a sprint. Jeff reports significant benefits in terms of velocity as a result of putting this in place.

This forms part of the story grooming process. It establishes a set of criteria (or a checklist) of items which need to be in place before a story can be taken into Sprint Planning.

Good examples of what this might contain are:
- Story estimated at 13 story points or less
- Story written in user story format

- Story has at least the happy path test case written
- Story has a business value estimate
- Story has an initial UI sketch mockup
- Constraints on story indicated

Establishing a set of “Ready” stories (ideally 1.5 - 2 sprints worth) will help to prevent thrashing in the backlog. It creates a quality benchmark and optimises the flow of stories from the team.

The danger is that you might “over-invest” in a story, so the key is to not have too many stories ready to go. The principle of “just enough, just in time” should be respected. Also, this is not a license to start creating spec documents; the intention is still to allow for solutions and requirements to emerge through the process. As a result I advocate a fairly light “definition of ready”; just enough to encourage Product Owners to prepare adequately for Sprint Planning.

## User Experience (UX) and User Centered Design (UCD)

These emerging (and overlapping) approaches to product development can be useful additions to the Product Owner toolkit. Both processes place the user at the forefront of the development process.

The starting point is a conversation with real users to discover what they need and how using your product will help them realise their goals. Too often we believe that we know better than our users do, what it is that they need. This takes the form of a number of interviews where data on the demographics and ways in which these users interact with the product are gathered. In particular an emphasis is placed on what  the user’s goals for interaction are.

This information is then used to help you and your team to visualise and appreciate the user’s perspective by creating “personas”. These are fleshed out portraits of the main users (refer to your Product Vision to determine who your main users should be). We define in as much detail as possible what this user’s characteristics are: name, age, a picture etc. In effect you’re creating a biography of this fictional person.

Once this is established you want to use this persona to identify the user’s goals and relationship with the product. This will help you to construct a story of this person’s interaction with the product. The value lies in this collaborative process of discovering these stories with the team, allowing for a deeper understanding of the product. It will also help to identify stories which you might miss or raise the priority of stories based on the user goals.

This is a large and growing field of product development with more and more interaction between UX designers and the Agile world. I would encourage you to read more about it. See the list of recommended books and articles for links.

## What about bugs?

It is a sad reality of our industry that we must contend with the errors we produce as part of the product development process. The impact of this variability is that it introduces a level of unpredictability on the schedule and delivery. Common responses to this is to try and reduce the impact by placing bugs in a different workstream, away from the product development effort. This might seem pragmatic, but it removes the accountability for the error (and the potential learning) from the team responsible for the error. I am a strong advocate for the team fixing their own bugs.

As a product owner, your chief responsibility is determining the priority in which work should be done. My recommendation is that you treat bugs just like any other type of work. Identify the business value of fixing and impact of not fixing the bug and schedule it relative to other stories. I would recommend that you dispense with the usual story format for bugs; the fit is usually poor. What I think it vital however is that the test cases for the bug are clear.

The significant characteristic of bugs is that the more quickly they are fixed the better. Programming is complex work and the context in which the bug was created is a small window. Most developers will barely recognise their own code after two weeks have elapsed. So scheduling bugs to be fixed as soon as possible will mean that the team is likely to be better prepared to fix the bug.

It is only worth tracking and scheduling defects that have escaped the sprint. Bugs found during the sprint as part of testing a story are best tracked as tasks on the task board.

Lastly, it is worth analysing defect clusters with the team to identify the root cause for the escaped defects. This is usually a periodic activity that can be the topic of a retrospective or A3. It is best to discuss with your ScrumMaster when to schedule such an activity.

## Less is more

User Story format is usually expressed as:

As a *user role* (WHO)
I want *some functionality* (WHAT)
So that *get some business value* (WHY)

I am a supporter of using physical artifacts rather than electronic to track Scrum team's progress. I recommend that you therefore try to write the story in this format using a large nibbed pen on a small index card. The idea is that you want to express only the bare details. It is not a novel or a recipe. The story itself should be a **reminder** of the functionality and not a complete specification. Think of this as the headline of a newspaper story, rather than the actual story copy. Don't try to be grammatically correct; remove extraneous words like "should" or "be able to". They have no place in a headline.

To help make your headline as punchy as possible, move as much detail as possible to where it belongs: in the test criteria. For example, you should not be describing all the capture details for a form in the story.

A good rule of thumb: if you're having trouble fitting your story on a card, use a smaller card.

An example of a typical user story: “As a user I want to be able to delete items from my shopping cart when I am ready to checkout so that I can make sure that I only get the items I really want”.

An example of a better version of the same story: “As a regular shopper I want to delete items from my cart so that I get the items I really want”.

## The Full Story

All three elements of a user story are equally important.

The “WHY”
The reason we ask Product Owners to write the “Why” is to help you understand what the value of this piece of functionality is. To whom is it valuable? How does this value get created through the functionality. This information is also valuable for the team; it helps them contextualise the story. If anything however, it is more valuable to the PO. Evaluate the story in light of your Product Vision. Are you providing something at odds with this vision? If so it might be time to revisit the vision or question the value of the story...

The “WHO”
The “Who” in the story should be someone quite specific. This is hard, because it means you need to consider quite carefully who your application’s users and customers are. The more detail you can pack into the profile of your users, the less likely you’ll be to miss critical functionality.
Typical ways we divide users are things like: new user vs registered user, frequent user vs. infrequent user, limited rights vs. administrator, Incoming source (Google, Twitter, Facebook etc).
The Product Owner role is rarely (if ever) an acceptable user role for a user story. If the PO is undertaking activity within the application, look to their role within the application to describe them (e.g. if the PO also happens to be a content editor, website admin, website SEO & analytics etc).

For more on this see the section on UX & UCD above.

The “WHAT”
The description of the functionality should be expressed in terms which do not anchor the team to a specific solution. Try to avoid technical terms like database, cookie etc. Stick to terms which the user would use themselves (e.g. I want to remember my log in details vs. I want to use a cookie).

## Small is beautiful

Stories should be as small as possible, while still adding value to the application. I’ve rarely if ever run across stories which are too small. Small stories optimise throughput of functionality. They allow for shorter feedback cycles and as a result, better quality.

Splitting stories into smaller stories is a skill which is acquired by practise.

Start by selecting only the bare necessary aspects. The analogy to keep in mind is that all cars have the same basic functionality (an engine, a seat, steering wheel and brake). This is true whether you’re driving a go-cart or the latest SUV. There is an “essence” of what the functionality that a car provides. What we want to understand is what the essence of the feature is and defer as much of the non-essential functionality as possible.

Identify how the feature can be used in multiple scenarios or in alternative ways. This is something which can be moved into another story. For example, “move file” could be used to delete files, if the “bin” was implemented as a folder. So if we limited “delete” to a specific function only, then we can limit the scope of the “move” function.

Move safety features (validation, error prevention, data protection) as far as possible into a separate story. This allows us to incrementally improve the quality of the function (and the product). It also makes explicit to the development team what safety features are valuable. This is particularly the case for dealing with edge cases and exceptions.

Finally, stories which enhance the usability or performance of the application should ideally be placed in a separate story. This approach allows us to first get the functionality “right” before we get it “good”. This is again about adding quality incrementally. It allows us to test the application with real scenarios and data to test our understanding of the usability and performance.

Performance is also usually a fickle problem; it’s best to leave this till last, and improve it incrementally. It’s often the case that initial performance gains are easily realised (80/20 principle) and investing large amounts of time to squeeze out the last 20% may not be the best use of effort.

Usability functions (e.g. auto-completion, sexy visual design, shortcut keys) are also best left till the basic functionality is completed, before adding to the feature. This has two advantages; it allows some flexibility in the co-ordination with the design team and allows evaluation of the application first, before deciding the priority of the “polish”.

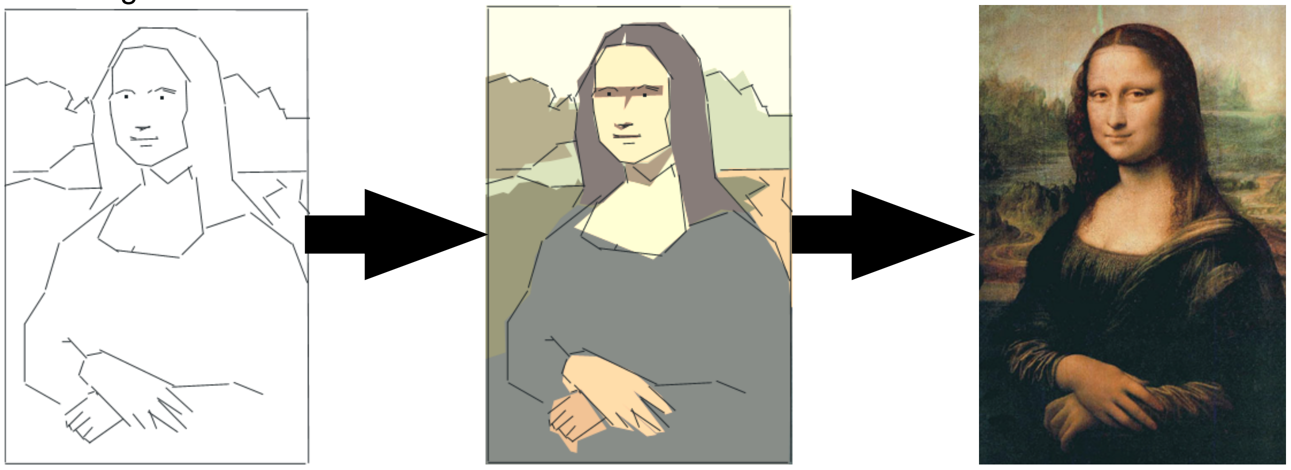
You should also use your test scenarios to help you identify opportunities to split stories (more on test scenarios below).

Other things you could use to split stories:

- Workflow (e.g. draft, review, publish)
- Operations (e.g. create, update, delete)
- Business rule variations (e.g. admin & limited rights)
- Major / Minor effort (adding the first credit card e.g. VISA, vs. adding additional credit cards)
- Simple/Complex (dealing with none, one, many as separate stories)
- Data entry method (xml feed, file upload, copy/paste vs. typing etc)
- Defer performance
- Split the investigation / PoC phase from the delivery

## Incremental is Inevitable

Something which both you (as PO) and the team will need to become comfortable with is working incrementally. This means that we will revisit features again and again as we flesh out and grow the feature.



Like the illustration above, we want to start by describing the bare working features (of the whole application) and then slowly add more detail as we go. This is quite different from working in phases.

The principle difference is that we want to get to a “walking skeleton” which demonstrates a full set of functionality while limiting the depth of the individual features. This allows us to get a feel of the whole system end-to-end as quickly as possible. Phases on the other hand try to select components or modules to defer.

This process of incrementally building up the feature needs to be subject to the process of building valuable, working software. This feels wasteful sometimes, and there are times when this does incur some overhead, but it also allows us to eliminate work “not done”. One of the aims of an incremental process is to stop adding additional elements to our feature when the incremental value added is low. At this point, we would usually be better off enhancing or adding functionality on another feature. This process is sometimes called “trimming the tail”.

## The details are in the tests

If the user story should be as simple as possible (but no simpler), then we can get almost limitlessly detailed in the test descriptions. A good pattern for describing tests is:

GIVEN: startingcriteria1 AND startingcriteria2 AND startingcriteria3 ...

WHEN: some event

THEN: outcome1 AND outcome2 AND outcome3 ...

For example

**User Story:**
**As a** bank customer,
**I want** to withdraw cash from an ATM,
**so that** I don’t have to wait in line at the bank.
**Scenario 1: Account is in credit**

**Given** the account is in credit

**AND** the card is valid
**AND** the dispenser contains cash

**WHEN** the customer requests cash

**THEN** ensure the account is debited
**AND** ensure cash is dispensed
**AND** ensure the card is returned

Notice how the "**AND"** criteria flesh out the scenario and offer additional opportunities for scenarios. If we identify an "OR", for example: "AND the dispenser does not contain enough cash" this should be a different scenario.

**Scenario 2: Insufficient cash in dispenser**

**Given** the account is in credit
**AND** the card is valid
**AND** the dispenser does not contain enough cash

**WHEN** the customer requests cash

**THEN** decline transaction
**AND** ensure the card is returned

The advantage of this approach is that it allows us to identify less valuable scenarios and place them in separate stories.

## Non-Functional Requirements

"a non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours. This should be contrasted with functional requirements that define specific behaviour or functions" - Wikipedia

Some examples are:

- Accessibility
- Audit and control
- Availability
- Compliance
- Interoperability
- Maintainability
- Performance / response time
- Portability
- Resource constraints (processor speed, memory, disk space, bandwidth etc)
- Response time
- Scalability (horizontal, vertical)
- Security
- Usability

What is key to understanding the impact of non-functional requirements is that they are usually cross cutting concerns which impact many aspects of the application. This is not a once-off story which can be completed. It recurs with every new story added.

One approach we can use to help us understand what our requirements are and when they impact stories is the following:

Start by formulating your non-functional requirement (in this case Responsiveness) as a story. The reason for this is that it forces us to consider who the user is that is impacted and what the benefit is that we derive from imposing a constraint on the application

“As a Video on Demand (VOD) catch up viewer, I want my video to play within 30 seconds, so that I don’t get frustrated and download a torrent instead”

Now we need to formulate the test criteria to express our expectations:

“GIVEN I’m viewing “My funny cat” video clip AND I am using Firefox 3 WHEN I click play THEN the video buffers AND playback starts within 30 seconds”

In future, when we add additional functionality to the VOD player (for example pop up bubbles in the video canvas) we would impose the performance constraint on this story to ensure that the addition of the new feature does not diminish the performance on the application.

Closing the loop is the final part. When writing the new story above “AS A social VOD user, I WANT to see other people’s comments in this video SO THAT I can comment on it as well” you would indicate that this is subject to the performance constraint

| Functional Requirement | Performance | Scalability | Security |
|---|---|---|---|
| AS A social VOD user, I WANT to see other people’s comments in this video SO THAT I can comment on it as well | X | | X |
| | | | |
| | | | |

## Technical Debt

**What is technical debt?**
Ward Cunningham coined the term in a 1992 report to draw an analogy between shipping the first release of a software product and going into debt. There are a few key considerations in order for the metaphor to hold. Firstly, that the decision to go into debt is a conscious one, and does not happen by accident. Secondly that the decision carries with it an escalating cost as the “interest” on the debt comes due.

This leads to the recognition that eventually there will come a time when the cost of carrying the debt (the interest) becomes significant enough to warrant repaying the debt.

Since the introduction of the term, there has been a dilution of its meaning. As a result I prefer to distinguish some different kinds of technical debt in order to highlight the different ways in which the Scrum team will need to deal with it. This taxonomy of technical debt owes a debt to “Uncle” Bob Martin and Joshua Kerievsky for their influence on its descriptions.

The most common form of technical debt is what “Uncle” Bob calls “mess”.  This is especially apparent when working with legacy applications. The sheer amount of messy code can be an impediment to the team’s progress. This would include things like: no tests, bad or inconsistent style, poor implementation, no use of design patterns, poor naming conventions, the frequent appearance of the word “Hack” or “TODO” in comments. The team should be cleaning up this code as they work (teams could include a “boy scout principle”[1] in their “definition of done” for example).

If you find yourself the Product Owner of such a legacy application it may not be sufficient to hope that the team can repair this code on a tactical level through their definition of done. In those circumstances a strategic approach may be necessary. Key to this is measuring the amount of mess and creating a set of stories to address the mess. These stories should be scheduled in the backlog and prioritised. A handy tip for prioritising is to make the technical debt stories visible in the team area and have the team dot vote whenever they encounter a piece of mess. This will quickly indicate which stories should be prioritised.

Another form of technical debt is defined as part of the Scrum framework: “Undone Work”. This is defined as stories which are created due to the team’s inability to deliver truly “done-done” work at the end of the sprint. In Scrum the concept of potentially releaseable software implies that the decision to release is a business one. As a result, if you find that there is still some barrier to releasing your product due to a technical limitation you are accumulating undone work. Some examples might include regression testing, deployment to production, integrating with 3rd party or another team’s application or library.

As a product owner, you should be accumulating this undone work as part of the backlog. The intention is to make visible the impact of adding new stories on your ability to release. We typically find that the rate at which undone work accumulates is non-linear with the addition of functionality.

At a tactical level, you would just do the undone work before you release. Strategically, you must question the value of of the undone work, preferably with your team. This conversation will hopefully lead to a set of technical debt stories to reduce the impact of undone work. In particular, items like manual regression testing must be replaced with an automated test suite. Automating the build and release process is another candidate for implementation.

In the Lean Software movement, we speak of the “concept to cash” period. The more quickly your new functionality is in the market place earning value, the better it is for the organisation. The most significant barrier to this is usually found in the undone work. The business value proposition is the reduction in time-to-market with an ancillary benefit of freeing up of available capacity for building more features.

[1] The Boy scout principle says “Leave the campsite cleaner than what you found it”. So in code terms, this applies to cleaning up and refactoring code to leave it better than you found it.

The final category of technical debt is the only one I would characterise as being “real” technical debt. This is what Joshua Kierevsky calls “design debt”. It is often the case that when implementing a feature a Product Owner finds themselves having to choose between a “hack” and the more elegant solution. This is a conscious decision to trade “time to market” (or cost) vs. a better performing, more scaleable or more maintainable solution. We advocate strongly that the Product Owner is uninvolved in the decision of how a requirement will be implemented, but this is something which is not always possible.

When faced with this choice I recommend that firstly the team does not compromise on the quality of the implementation. If this is to be a “hack” or solution chosen for expedience then it should still be written as cleanly as possible. This means applying the same definition of done. Secondly, the reason for the design decision should be documented (preferably in way which is easily found when encountering the code).

And finally, a technical debt story should be written, estimated and placed in the backlog to be prioritised. This allows the Product Owner to track the accumulation of technical debt and make informed decisions as to the longevity of the product’s code base. It also creates a transparency allowing all stakeholders to remain informed of the code’s level of technical debt.

The reason this is important is that “design dead” products are created through the process of accumulating technical debt without an awareness of the impact. Symptoms of technical debt in your product include a gradual reduction in the team’s velocity till the ability to add new features effectively stalls. I find many Product Owners frustrated with the rate at which the team is able to develop, particularly given that past performance seemed better. The way to avoid this happening in the first place is to raise awareness of technical debt.

# Release Planning

## What is Business Value?

Donald Reinertsen in his book “Managing the design factory” postulates that all of our industry’s problems can be traced back to the lack of a good economic model of what we produce. Our inability to define the true value in real monetary terms drives poor decisions about what to build and when to build it.

We have analogies in models like Kano and Weigers which attempt to classify the customer need and use this as a starting point for prioritisation. What is however emerging from the Lean Software Development movement is an emphasis on real economic value.

This value can be derived from a number of different sources. For example:

- actual money customers are willing to pay for the product
- reduction in costs of development or implementation of the product
- legislative penalty associated with non-delivery of functionality
- reduction in risk or gain in knowledge about a risk
- time to market or first mover advantage
- reputation

In determining the value of a story or feature this should be a good set of criteria to start evaluating the story’s value. It should also be analysed in terms of the fit to your vision. If your vision is market share growth then the emphasis would be on new features that would attract new clients. If your vision is to retain existing customers then perhaps you need to prioritise features in your product that exist in your competitor’s offering.

David Anderson (best known for his Kanban system for software development) introduced a good set of categories for understanding the contribution of a feature with respect to the market and the strategic direction for the product.

Table Stakes
- Undifferentiated
- Commodities
- “must have”
- Customer retention

Differentiators
- Drive customer choice/selection
- Drive profits

Spoilers
- Spoil a competitors differentiators
- Customer acquisition

Cost Reducers
- Reduce cost to produce, maintain, implement or service
- Increase margin

This aligns the prioritisation of features with your vision. The key insight however is that a release focussed on just one of these elements (where the bulk of features are drawn from a single category) is unlikely to be a good choice. The allocation of effort to these categories will depend on the phase of the product life cycle.

Mature products will drive more effort towards Spoilers and Cost Reducers. Their Table Stakes should already be in place and the drive towards innovating in their market space may not represent a cost-effective direction.

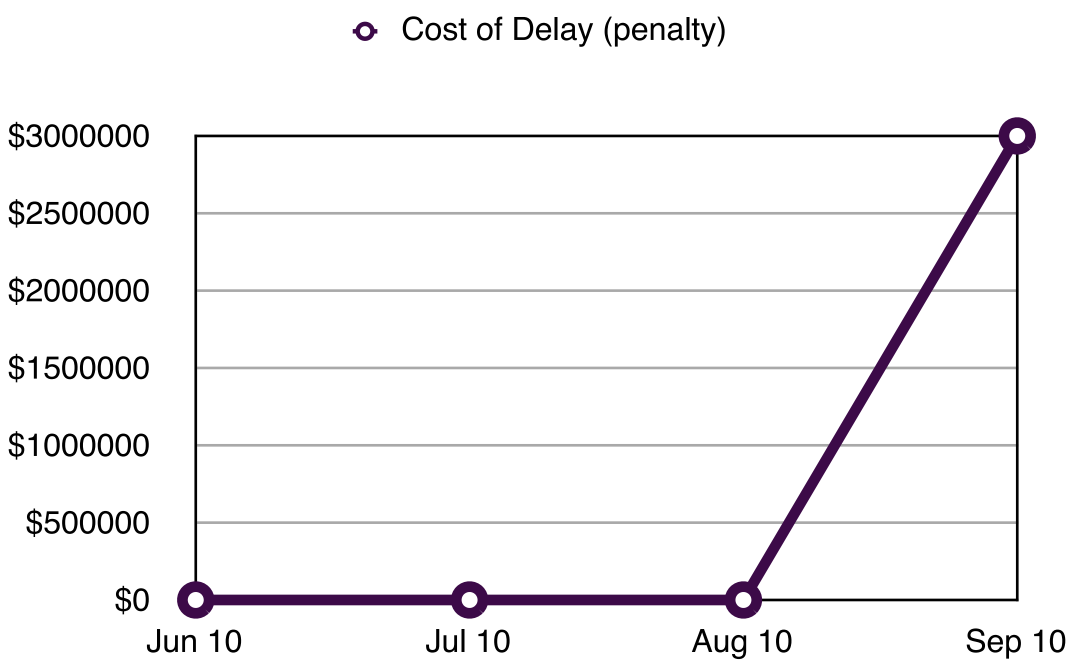
Conversely new entrants into the market will need to invest in Table Stakes and Differentiators with a helping of Spoilers dependent on the number of players within the market.

All of this leads inexorably to the need for a clear understanding of the Product Vision and the market in order to best identify the types of features and the business value they represent.

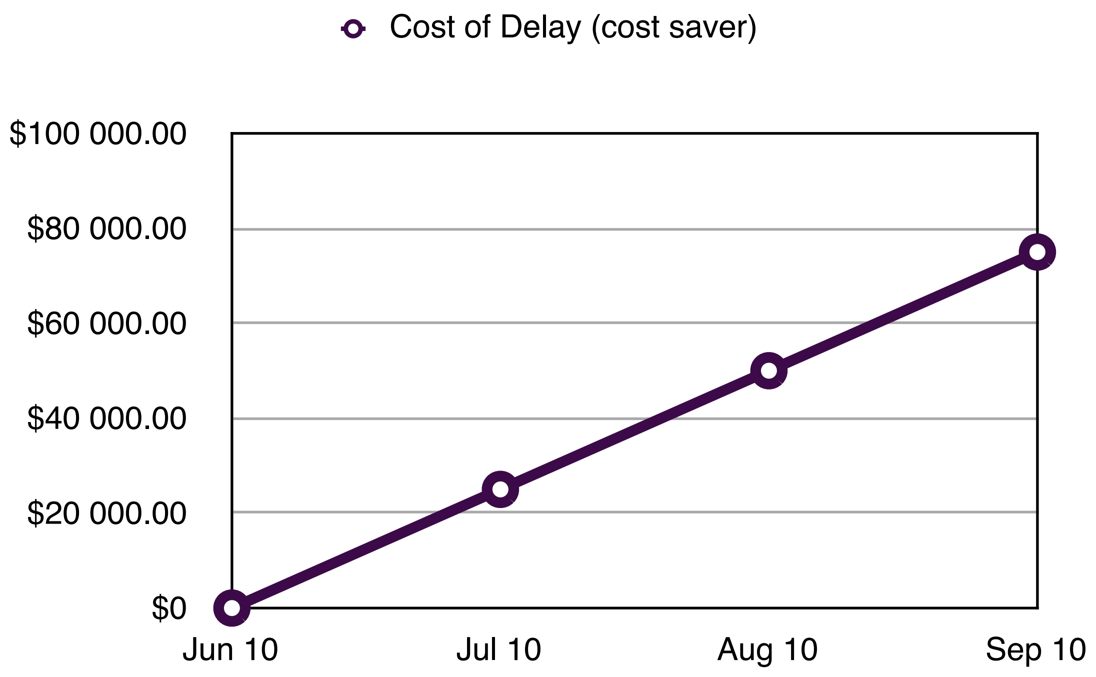
The last element which should be taken into consideration in determining priority is time. With the limited resources available, the implication is that when the team is working on a feature this automatically translates into not working on some other feature. This “time value of development” must inform the priority of feature choice.

We use velocity as a means to estimate the probable delivery date of items. This empirical measure of the rate of delivery should allow the Product Owner to calculate the “cost of delay”. This is sometimes an easier mechanism to discover which features should be prioritised than a raw monetary value. What is needed is an understanding of the rough order of magnitude of the cost associated with non-delivery of the feature and the shape of that curve.

For example: a feature associated with a legislative change would be catastrophic if not implemented by a defined date. Examples of such features are non-compliance with new regulations mean a a penalty (for example $3,000,000 at 1 September 2010)



In contrast a feature such as a cost saver (perhaps a usability tweak which will reduce calls to the call center) has a linear cost of delay and the slope is much less steep.



Notice the difference in scale and curve associated with it. If you are currently in June 2010, you may choose to prioritise the usability tweak because although the cost saving is not as great as that of the penalty, the timing indicates that if you can finish both of them, you should choose to do the cost reduction feature first.

## Impacts of batch size

One of the key differences in Agile software development as opposed to a more traditional approach is that Agile favours small batches. This concept has however been embedded into the practises rather than made an explicit one. The most obvious place we see this is in the short timescales imposed on iterations. When Scrum was first codified it advocated sprints no longer than one calendar month.

Our industry has shortened this even further as we seem to be standardising on two weeks. This combined with an emphasis on working software creates a small batch approach to developing and releasing products. This allows for shorter feedback cycles and many other benefits.

The one which is less well understood and derives from the manufacturing world, is that small batches reduce variability. Variability is usually felt in our low predictability but has knock on effects on quality and efficiency. The longer it takes to complete something, the more likely that we will experience quality issues.

From this manufacturing paradigm we now understand that there is a mathematical relationship between batch size and variability. Donald Reinertsen describes big batches as a snake swallowing an elephant. What is clear is that if we mean to improve our predictability, we must reduce batch size accordingly.

In Product Owner terms this means that the size of features and stories need to be trimmed to the minimum. See “Small is beautiful” above on how to go about splitting stories.

## Story Mapping

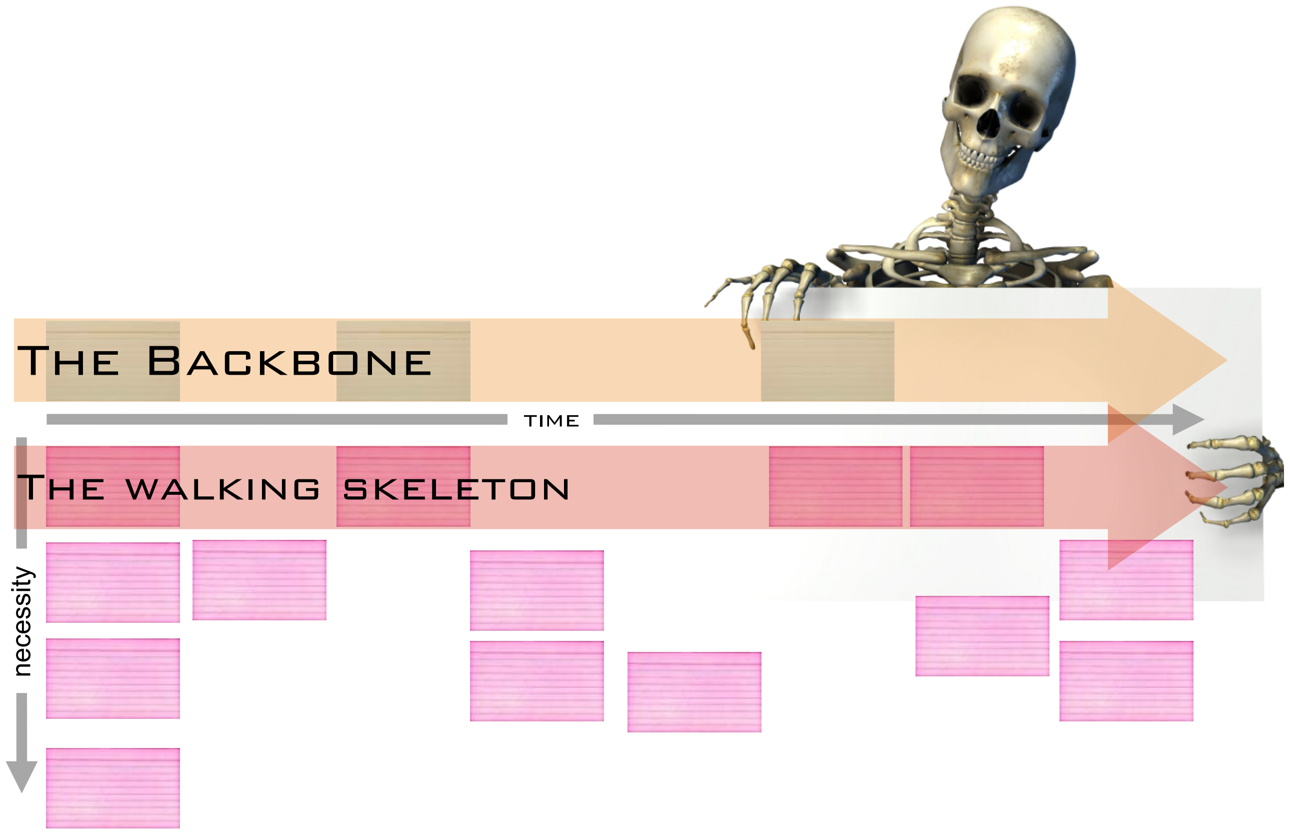
**a user centric approach to ordering the backlog**

One of the disadvantages of small stories is that they can make it difficult to see the big picture. This big picture context can be crucial for the development team to understand how a particular story should be developed. This approach uses the flow of the user’s experience through the application as a means to structure the backlog. It starts with establishing the users and their goals for interacting with the application.

There are many user personas which can be defined for the many types of people who will be interacting with the application. These personas can be differentiated on a number of criteria, the most important of which is the goal of the persona. For example we may want to make a distinction on a video-on-demand website between a ‘browser’ and a ‘soap opera fan’.

The ‘browser’ will be looking to find out about new shows or popular shows, or what their friends are currently watching. Their user experience is defined by the goal of finding novel content.

In contrast, the ‘soap opera fan’ wants to catch up on the episodes of their favourite show and perhaps see what the comments of other fans on the most recent episode are. Their user experience is defined by the goal of getting to their favourite content as quickly as possible.



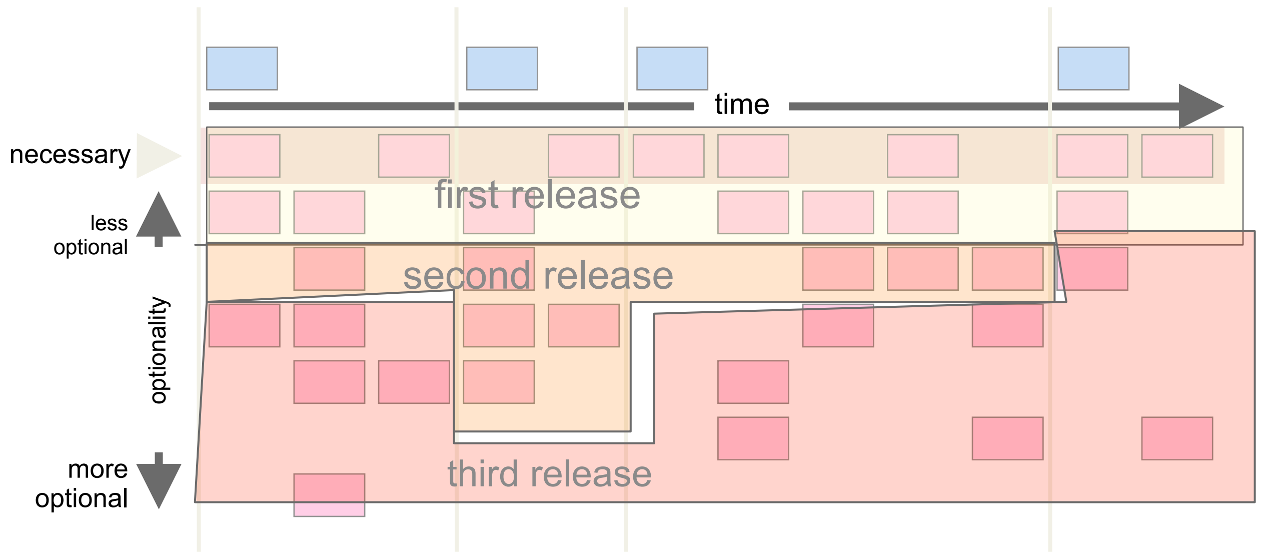
Once we've established some personas we can start to create a timeline of their interaction with the application. (We're going to stick with our VOD site for now.) This follows the process flow of, for example: registration, log in, manage show subscriptions, watch show, interact with show etc. (In the illustration above, this is the backbone we're trying to build).
We might want to order the flow so that it reflects the overall vision or goal of the project. This allows us to maximise the business value, since the vision will likely reflect the business value aims of the project. In our scenario above, that might mean that we would build the "manage show subscriptions" first and only later integrate the registration functionality. This does require some different ways of working for the team; the advantage would be maximising business value delivered (and potentially mitigating risk).

Once we've established this backbone for our principal user persona, we can use the additional personas to help us identify gaps or additional functionality and where they slot into our timeline.

The stories should be ranked in order of necessity below the functional group to which they belong. So in our example, "manage show subscriptions" might have stories like "*As a  browser I want to search for popular shows so that I can choose to watch what everyone else watches*" and "*As a soap opera fan I want my favourite shows to be on the landing page so that I can conveniently access my soaps*". Depending on the goal of the project or the relative size of the market segment we might choose to prioritise the "soap opera fan" over the "browser". This would be reflected in the necessity within the 'meat' of the skeleton.

## Release planning with a story map

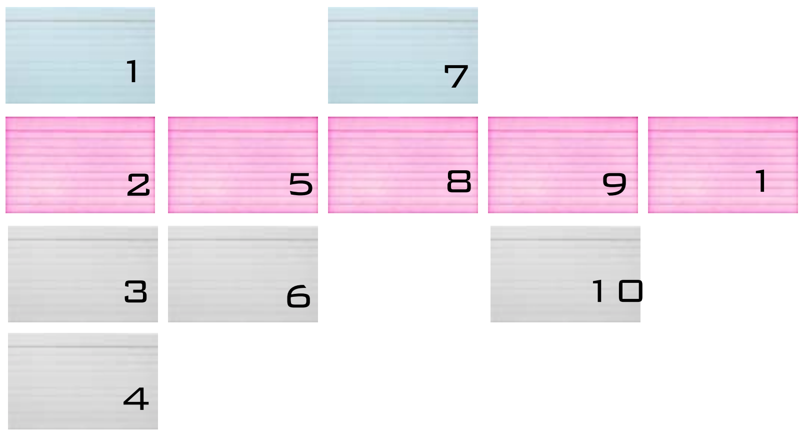
A story map contains two levels of prioritisation, and we need to get to the point where this is collapsed into a single level for the purposes of creating an ordered plan of stories.



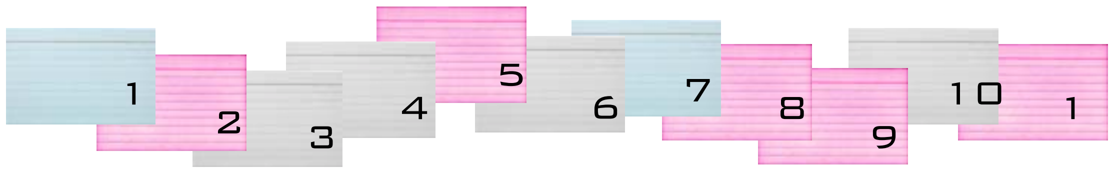
Our first task is to create a release plan. We choose from the top row(s) of our stories (the backbone) all the stories we need to achieve a release of software that takes us through the whole application end-to-end. If we've sliced our stories as small as possible this first release should have some stories from most of the features.

Now collect the stories from the release and order them with respect to the timeline (top-to-bottom and then left-to-right, see below).

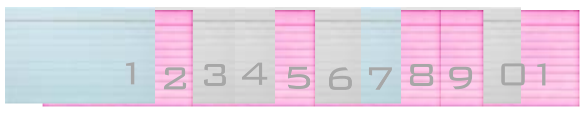
1.



2.



3.



This stack is our prioritised backlog and can now be estimated by the team.

Ideally this story map is a living, physical artefact in your team space. This should be kept separate from the sprint backlog. Use markup on the stories like stickers or colours to make it clear what has been completed or is currently in the sprint.

## Calculating the team's velocity

The team's velocity is fancy way of saying "How many story points the team delivers in a sprint". This will vary from sprint to sprint. This variability will be greater while the team is new or when some change is introduced (new team members, new technology etc). A good rule of thumb is to use the last three sprints and average them to obtain an estimated "true" velocity of the team. If the team's velocity varies by more than 50% from sprint to sprint, it is best to adjust this estimate to the lower end.

For example, a team which delivers 10,40,10 over three sprints has an average of 60/3=20 Story Points. But this is likely to be an optimistic estimate of their true velocity; in this case I would use 12-15SP to produce a release plan.

## Producing the release plan

| Sprint | Start | End | Story | SP | Sprint commitment |
|---|---|---|---|---|---|
| Sprint 1 | 24 Jun | 30 Jun | A1 | 2 | |
| | | | A2 | 5 | |
| | | | A3 | 3 | 10 SP |
| Sprint 2 | 1 Jul | 7 Jul | B1 | 5 | |
| | | | C | 8 | 13 SP |
| Sprint 3 | 8 Jul | 14 Jul | D1 | 5 | |
| | | | B2 | 5 | 10 SP |
| Sprint 4 | 15 Jul | 21 Jul | E | 5 | |
| | | | B3 | 2 | |
| | | | D2 | 3 | 10SP |
| Sprint 5 & 6 | 22 Jul | 4 Aug | G | 20 | 20SP |

A few things to notice about our release plan. We've stuck with a velocity of about 10SP. We can already give our stakeholders an indication of when a particular story (e.g. E) is likely to be delivered (21 Jul).

Story G, which is a large story still somewhat in our future, we've planned over two sprints. It's too large to fit into a single sprint, and as we get closer (probably during backlog grooming in Sprint 3) we can then break it into smaller stories. The principle of backlog grooming is "Just enough, Just in time".

## Recommended Reading List:

Jeff Patton - Story Maps [1], [2]

User Stories Applied - Mike Cohn

Agile Estimating and Planning - Mike Cohn

Succeeding with Agile - Mike Cohn

Roman Pichler: Agile Product Management with Scrum

37 Signals: Getting Real

Scrum Roles - Tobias Mayer

Practical Tools for the Product Owner: Focus, Value, Flow - Serge Beaumont

The Principles of Product Development Flow - Donald Reinertsen

Managing The Design Factory - Donald Reinertsen

Scrum and CMMI – Going from Good to Great: are you ready-ready to be done-done? - C. Jakobsen and J. Sutherland