Table of Contents

Incubaid Agile Development Process

Incubaid is a Belgian startup incubator with a proven track record of bringing innovative and exciting cloud infrastructure products to market.

The goal of this ebook is to describe a set of conventions on how a development process can work based on agile methodologies.

These ideas are opinionated but do result out of years trying to create good products and still maintain flexibility.

We invite everyone to contribute to this set of documents.

- https://github.com/Incubaid/dev_process

Please use the issue tracker to ask questions, post feedback, ask for improvements, report bugs, ...

- https://github.com/Incubaid/dev_process/issues

Agile

- We are strong advocates of Agile software development processes like Scrum and Kanban
- We believe the best process uses ideas from both

Book

- Online version (html)
- PDF version

To build & edit yourself

- checkout this repository (git)
- install nodejs: https://nodejs.org/en/download/
- install gitbook

```
sudo npm install gitbook-cli -g
```

- go to the downloaded repo and do

```
sudo gitbook install
gitbook serve
```

- you can also install the editor: https://www.gitbook.com/editor

to build the website

```
gitbook build .
```

to generate pdf's

install https://calibre-ebook.com/download

```
gitbook pdf .
```

Terminology

Agile

Roles

- See Roles

GitHub or other Git management system e.g. GOGS

Organizations

Organizations are a group of two or more users that typically mirror real world organizations. They are administered by users and can contain both repositories and teams.

E.g. Incubaid is an organization (on gogs they are also called owners).

IYO

ItsYou.Online

GIG Identity management solution, integrates with GOGS and other GIG App's.

Users

Users are personal GitHub/IYO accounts. Each user has a personal profile, and can own multiple repositories, public or private. They can create or be invited to join organizations or collaborate on another user's repository.

E.g. despiegk is a user who has access to an organization or repository in an organization.

Repository

A repository is the most basic element of GitHub/GOGS. They're easiest to imagine as a project's folder. A repository contains all of the project files (including documentation), and stores each file's revision history. Repositories can have multiple collaborators and can be either public or private.

https://github.com/Incubaid/dev_process is an example repository.

Issues

Issues are suggested improvements, tasks or questions related to the repository. Issues can be created by anyone (for public repositories), and are moderated by repository collaborators. Each issue contains its own discussion forum, can be labeled and assigned to a user.

Example: https://github.com/Incubaid/dev_process/issues

Product Owner

- a person who owns a product line from a technical perspective e.g. Jumpscale
- a product line is an organization/account on github level e.g. Jumpscale
- this person owns that account & will make sure all is organized properly
- see roles for more info
- This is not the product manager !!! Product manager defines product as how it fits for customers and works together with the product owner from engineering perspective.
- Owns an organization on github

Project Owner

- a person who manages a project
- a project can be internal or with customers
- each project has a project repo (see types of repo's)
- the project owner is responsible for the project repository and will make sure that repo is organized well
- see roles for more info
- Owns one or more repo's (proj_...) on gogs

Product Manager

- Owns product from a go2market/commercial/marketing perspective.
- Defines roadmap.
- Organizes stakeholder meetings.
- Owns one or more repo's (prod_...) on gogs

Kanban as a Nice Add-on to Scrum

our methods are mainly using scrum methodologies but we added some Kanban twists to it.

Kanban primarily follows four core principles (info from link)

- Visualize work
    - Create a visual model of work and work flow, so as to observe the flow of work moving through the Kanban system. Making the work visible, along with blockers, bottlenecks, and queues, instantly leads to increased communication and collaboration.
- Limit work in process
    - Limit how much unfinished work is in process and reduce the time it takes an item to travel through the Kanban system. Problems caused by task switching and the need to constantly reprioritize items can be reduced by limiting WIP.
- Focus on flow
    - By using work in process (WIP) limits and developing team-driven policies, the Kanban system can be optimized to improve the smooth flow of work, collect metrics to analyze flow, and even get leading indicators of future problems by analyzing the flow of work.
- Continuously improve
    - Once the Kanban system is in place, it becomes the cornerstone for a culture of continuous improvement. Teams measure their effectiveness by tracking flow, quality, throughput, lead times, and more. Experiments and analysis can change the system to improve the team's effectiveness.

We need features from both

Advantages of Scrum	Advantages of Kanban
Transparency	Flexibility
Improved credibility with clients	Focus on continuous delivery
High product quality	Increased productivity and quality

Product stability	Increased efficiency
Team members reach sustainable pace	Team members have ability to focus
Allows client to change priorities and requirements quickly	Reduction of wasted work/wasted time

Why can Kanban help as add-on

Kanban: A Good Fit for Teams that View Work as Firefighting.

Toyota production plants and Scrum teams exist to build product. The literature speaks of successful application of Kanban in the service industry, analogous to firefighting or hospital emergency rooms. It's tricky to schedule your next fire unless you live in the world of Fahrenheit 451.

Many development teams run in firefighting mode, often with "swooping" from the Product Owner during a Sprint. And getting into a discipline is hard: it's easy to want to be able to react immediately to customer change instead of integrating a request into the business plan, and it feels good to release whenever you see fit. (link)

The Ideas

GitHub Integration

Git/GitHub is a most amazing tool to track changes and can be used in many more ways than people deem possible.

We suggest using GitHub or similar tools like Gogs for:

- Specs
- Documentation
- Planning work (stories in home repo, see stories)
- Tracking of bugs
- Tracking of feature requests
- And of course all coding work using pull requests as a tool to track changes
- Tickets from customers
- Its important that all change is being tracked and people get visibility on everything which happened and is going to happen

Requirements

- If security is important to you make sure everyone uses 2-factor authentication
- If possible use clean login names on GitHub, many people use funny names which is nice but makes it hard for people to remember
    - Our suggested login names are $first7lettersLastName+$firsLetterFirstname e.g. Kristof De Spiegeleer becomes *despiegk*

Terminology Git related

Branch

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or master branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the master branch to publish your changes.

Clone

A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere, or the act of making that copy. With your clone you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online. It is, however, connected to the remote version so that changes can be synced between the two. You can push your local changes to the remote to keep them synced when you're online.

Collaborator

A collaborator is a person with read and write access to a repository who has been invited to contribute by the repository owner.

Commit

A commit, or "revision", is an individual change to a file (or set of files). It's like when you save a file, except with Git, every time you save it creates a unique ID (a.k.a. the "SHA" or "hash") that allows you to keep record of what changes were made when and by who. Commits usually contain a commit message which is a brief description of what changes were made.

Contributor

A contributor is someone who has contributed to a project by having a pull request merged but does not have collaborator access.

Diff

A diff is the difference in changes between two commits, or saved changes. The diff will visually describe what was added or removed from a file since its last commit.

Fetch

Fetching refers to getting the latest changes from an online repository (like GitHub.com) without merging them in. Once these changes are fetched you can compare them to your local branches (the code residing on your local machine).

Fork

A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.

Git

Git is an open source program for tracking changes in text files. It was written by the author of the Linux operating system, and is the core technology that GitHub, the social and user interface, is built on top of.

Markdown

Markdown is an incredibly simple semantic file format, not too dissimilar from .doc, .rtf and .txt. Markdown makes it easy for even those without a web-publishing background to write prose (including with links, lists, bullets, etc.) and have it displayed like a website. GitHub supports Markdown, and you can learn about the semantics here.

Merge

Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another. This often happens as a Pull Request (which can be thought of as a request to merge), or via the command line. A merge can be done automatically via a Pull Request via the GitHub.com web interface if there are no conflicting changes, or can always be done via the command line. See Merging a pull request.

An organization holds X repositories

Private Repository

Private repositories are repositories that can only be viewed or contributed to by their creator and collaborators the creator specified.

Pull

Pull refers to when you are fetching in changes and merging them. For instance, if someone has edited the remote file you're both working on, you'll want to pull in those changes to your local copy so that it's up to date.

Pull Request

Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests each have their own discussion forum. See Using Pull Requests.

Push

Pushing refers to sending your committed changes to a remote repository such as GitHub.com. For instance, if you change something locally, you'd want to then push those changes so that others may access them.

Remote

This is the version of something that is hosted on a server, most likely GitHub.com. It can be connected to local clones so that changes can be synced.

SSH Key

SSH keys are a way to identify yourself to an online server, using an encrypted message. It's as if your computer has its own unique password to another service. GitHub uses SSH keys to securely transfer information from GitHub.com to your computer.

Upstream

When talking about a branch or a fork, the primary branch on the original repository is often referred to as the "upstream", since that is the main place that other changes will come in from. The branch\/fork you are working on is then called the "downstream".

Blame

The "blame" feature in Git describes the last modification to each line of a file, which generally displays the revision, author and time. This is helpful, for example, in tracking down when a feature was added, or which commit led to a particular bug.

Github magic comments

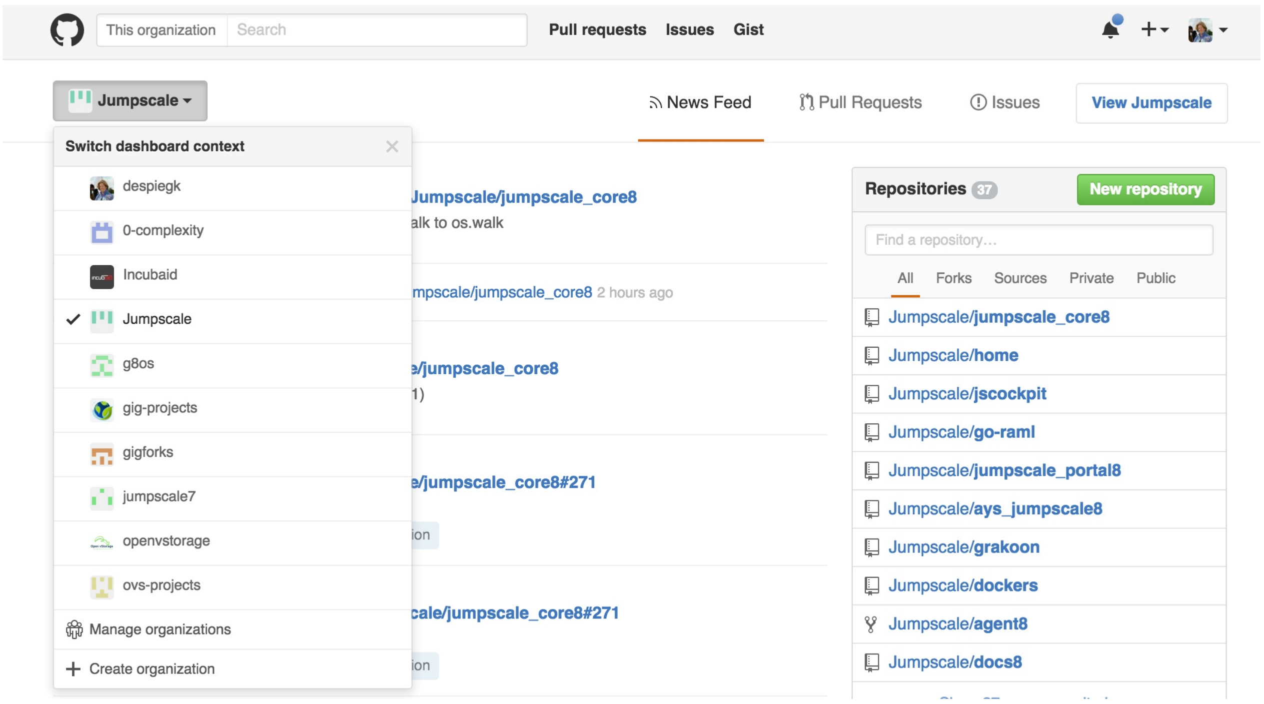
We support some 'magic' comment that you can use in issue and help to manage story and tasks.

`!! prio $prio` Set the prioriry of an issue. $prio is checked on first 4 letters, e.g. critical, or crit matches same

`!! p $prio` alias above

`!! move gig-projects/home` move issue to this project, try to keep milestones, labels. if it's a story, move all the tasks related.

GitHub Organizations (gitorg)



We define 2 types of organizations:

- Code Organizations ("codedorg")
- Project organizations ("projorg")

Whatever we describe can be used on github as well as on similar tools like Gogs.

Code Organizations

- Groups repositories related to the development of a product
- Typically code repositories or the home repository
- Examples:
  - https://github.com/jumpscale
  - https://github.com/openvstorage
  - https://github.com/0-complexity
  - https://github.com/incubaid

Details in the section about Code Organization repositories

Are always in Github.

Project/Company organizations

- Groups repositories related to the management of projects, customers or sales
- Gogs is an amazing tool to support all kinds of processes, next to just software developement processes
- There is no code in a project organization
- Examples:
    - https://github.com/gig-projects/

Details in the section about project organization repositories

Are always in GOGS

Gogs versus Github

- We suggest you use Github for all product related repositories, especially when you work with opensource software. Github is today by far the leading git management tool for these types of projects.

- We suggest you use Gogs for all project organizations, because these repo's need to be private and security is more important. We have forked a version of Gogs which integrates with itsyou.online, this allows you to have very good security & single sign on (IYO).

Code Organization Repositories

In a **product** organization ("codedorg") following repositories can exist:

- **home** repository
- Source **code** repositories
- **AYS** repositories (AYS templates, is not same as cockpit repo's)
- **Websites** (www) repositories (optional)

Below each of the codedorg repositories is discussed.

Home repository

- Only one per "codedorg"
- Always named `home`
- Is the main repo of a codedorg
    - Is owned by the product owner (is not the product manager).
    - Product owner needs to make sure its clear he owns this organization/account & is responsible of content
- It holds
    - Documentation common to the repos in the codedorg
    - Explains the purpose of all other repos in the codedorg
    - Starting point for people to find their way in all other repos in the codedorg
    - Can hold specifications and product requirements document (PRD) information common to the repos in the codedorg (in other words if not linked to 1 specific repo)
- Milestones
    - Are version numbers if they are relevant over all products in the account/organization (if relevant)
- Types of issues in the `home` repo:
    - Only issues of type Question (type_question) and Feature Request (type_feature)
        - Used for questions and feature requests that are relevant to the whole Code Organization
        - All other questions and feature requests should be asked in the repository the question relates too

- This is the ideal repo to define your roadmap covering multiple product and component repos in the codedorg

- Suggest to include following documents:
  - **README.md** (`$homerepo/Readme.md`)
    - Link to all documents mentioned below
    - Linkts to join related telegram groups
    - Is starting point for full account/organization
  - **terminology.md** document (`$homerepo/terminology.md`)
    - Defining all relevant terms related to your products
  - **components.md** document (`$homerepo/components.md`)
    - Defining the components which make up the products
    - Describing what these components do
    - Links to relevant code repositories
  - **roadmap.md**
    - See the Roadmap section

DO NOT USE THIS REPO FOR:

- Escalating bugs
- Story cards

Code repositories

- No specific naming convention
- Can have any name but NOT starting with the prefixes used by for other types of repositories, as described on this page
- This repo has following content (this content should not be in any other repo type)
  - all code
  - all specs
    - specs related to code is just next to code
    - generic specs are in /specs directory
  - all documentation (put underneath /doc in repo)
  - all feature requests (by means of FR issues)
  - all bug issues (by means of bug issues)
- Milestones
  - Are releases for a product and linked to a delivery time
- Types of issues in code repos:
  - Issues of type Bug, Feature and Question

  - DO NOT use stories or tests at this level!
  - Question: should we allow tasks at this level??? #TODO *1
- Bug/feature management
  - All reported issues or feature requests are grouped, it acts like a funnel
- How to easily make sure bugs and features are linked to a story.
  - Put unique story card name at beginning of title e.g. story1:...
  - The Incubaid development process tools will find the story and make sure that the information gets linked.

www = websites repositories

- Always named as `www_$name`
- Contains the code for a website
- Is simular to a code repo but this case for a website.
- When code changes website is updated and hosted automatically
- Types of issues in website repos:
  - Issues of type Bug, Feature and Question
- We use tools like caddy/hugo to automatically publish a website

AYS templates repositories

- Always named as `ays_$name` : examples
  - `ays_$customer_$envname`
  - `ays_gig_testenv_dubai1`
- Milestones
  - For grouping feature requests and bugs
- Types of issues in AYS template repos:
  - Issues of type Bug, Feature and Question
  - Should we allow tasks here???

Project Organization Repositories

In a **project/company** organization ("projorg") following repositories can exist:

- **home** repository (home)
- **Project** repositories (proj_*...)*
- **Organization** repositories (org_...)
- **Product** repositories (prod_...)
- **Cockpit** repositories (cockpit_*...*)
- **Websites** (www) repositories (optional) (www_...)

Below each of the "projorg" repositories is discussed.

Home repository

- Only one per projorg organization
- Always named `home`
- Starting point for people to find their way in all other repos in the projorg
- Types of issues in home repos:
  - Issues of type Question and Task
    - Used for questions and tasks that are relevant to the whole Code Organization
    - All other questions ans tasks should be created as issues in the repository the question or task relates too
  - Question for questions not related to one specific other repository
  - Overall
- Example:
  - https://docs.greenitglobe.com/gig/home

Project repositories

- Always named as `proj_$customer` or `proj_$customer_$projname`
  - Projects relate to customers or specific projects (so not linked to a team)
  - We use this to organize our work related
    - to 1 customer, only useful if customer project is large enough
    - to specific projects inside an organization if they are not linked to 1 group of people
  - Do not create specific `proj_$customer_$projname` unless if subproject is large

enough

- Milestones
    - defines a deadline for the project, there can be multiple but tasks or stories can only belong to one
    - freely chosen per project
- Types of issues in project repos:
    - Story, Lead, Ticket, Monitoring, Question or Task
- DO NOT USE STORIES FOR
    - feature requests or bugs
- Owned by project owner (sometimes called project manager)

Organization repositories

- Always named as `org_$name`
- These are repo's which are specific to a group of people
    - group of people can be
        - for development eg. a scrump team: naming convention org_devel_cairo1
- Suggested standard organization repos:
    - org_development (engineering)
    - org_support (all support requests (tickets) which are not in project repo yet)
    - org_internalit (internal IT & organization of company)
    - org_product (product management)
    - org_marketing
    - org_finance
    - org_legal
    - org_hr
    - org_quality (all QA related issues, automation code, performance testing, portal testing and automated tests)
- Milestones
    - Defines a deadline (date) for the projects, there can be multiple, but tasks or stories can only belong to one
    - Freely chosen per project
- Types of issues in organization repos:
    - Story, Lead, Ticket, Monitoring, Question or Task
- Examples:
    - org product mgmt (https://docs.greenitglobe.com/gig/org_products\)
- Owned by person who owns the organization e.g. marketing manager, ...

Product repositories

- Always named as prod `_$name`
- REMARK: is not part of Code Organization repo's, this is the commercial side of the business, belongs to a product manager
- Is product management / integration information.
- e.g.
    - how does the product relate to the component repo's (underneath the Code Organization repo's)
    - naming conventions
    - roadmaps (customer facing)
    - integration (AYS templates)

Environment repositories

- Always named as `env_$customername_$environmentname`
    - We use the following convention for `$environmentname` : `$countrycode - $G8type - $environmentnumber`
    - `$G8type` can be one of the following:
        - **scale** = for scaleout environments (separate CPU 1U nodes (1 motherboard per server) which are scale able, storage and CPU all in one unit)
        - **conv** = converged environment (multiple nodes in one system eg. 4+ nodes in one physical box with shared backplane. Storage + multiple CPU nodes in one box)
        - **stor** = for environments which are using +2 storage nodes with a minimum CPU config (5 nodes)
        - **G8** = for our G8 configs (CPU + separate storage on 70 disk nodes)
    - Examples:
        - `env_leal_mu_g8_1`
        - `env_gig_be_scale_1`
- Milestones
    - Defines the deadline (date) for the environment to be operational
- Freely chosen per project
- Types of issues in environment repos:
    - Monitoring, Question or Task (auto-created)
- If testing e.g. on an environment and issues are found create ticket_ issues on this repo.

- There can be an ays repo inside (info about the environment, even multiple version of ays repo's)
- Types of issues can be of type
    - Story, Task, Bug, Feature, Question, Monitor or Test, CustomerCase, Issue

- ## Cockpit repositories

- Always named as `cockpit_$customername_$purpose`
    - $customername = the reseller of the operator capacity, or "gig" for internal use
    - $purpose = a short useful name which describes the cockpit function, e.g.:
        - `git` for our internal gitrobot
        - `resell` if it is cockpit for a reseller
        - `demo` if it is a Cockpit for demo purposes
    - Examples:
        - `cockpit_combell_resell`
        - `cockpit_gig_moehaha-demo`
- Used to deploy an infrastructure from out of GIT
    - Documents a full IT environment
    - Has all required process information embedded in AYS service instances
    - All changes are strictly controlled by GIT with pull requests
- Is the cockpit environment which runs our management framework, e.g.
    - AYS robot
    - Telegram Chatbot
    - Rogerthat Chatbot
    - Portal
- Can be used to manage X nr of G8 environment
    - Can also checkout an env_... repo and the AYS info is in the env repo (see above)
- AYS repositories are inside
    - There can be more than one AYS repository hosted inside a Cockpit repo
        - Hosts the AYS service recipes and AYS service instances which make up the environment to be managed
        - Is a directory which has an empty file .ays inside
- Milestones
    - Defines a deadline for the environment, freely to be chosen
- Types of issues can be of type

  - Story, Task, Bug, Feature, Question, Monitor or Test, CustomerCase, Issue

www = websites repositories

- Always named as `www_$name`
- Contains the code for a website
- Is simular to a code repo but this case for a website.
- When code changes website is updated and hosted automatically
- Types of issues in website repos:
  - Issues of type Bug, Feature and Question
- We use tools like caddy/hugo to automatically publish a website

GitHub/Gogs Milestones

used to plan future.

In Code Organizations (codedorg)

- Only use version nr's e.g. 8.1.0
    - Used to organize work around a component release !!!
- If different components make up a product & version is per product then this milestone name needs to be the same over the relevant repo's !!!
- Milestone is used to group feature requests and bugs
- Stories can link back to a specific milestone & issues in a repo
    - Its handy to use a query to show e.g. all bugs or e.g. JumpScale8 repo and then link this 1 query to a story card in the projorg

version nrs

- major 1.0.0
- minor 1.1.0
- feature 1.1.2
- build 1.1.2.333

There are maximum 4 milestones per codedorg repo

- Version number 1 (nearest to today, major, minor or feature release)
- Version number 2 (next release, major, minor or feature release)
- Version number 3: Optional: A major or minor version number after nr 2 (only relevant when 1 or 2 are not major or minor)
- Roadmap (always use this name!, is for future work)

DO NOT

- Use timing milestones, e.g. may16
- Use names

In Project organizations (projorg)

- Milestone is a time on which a company (organization) wants to deliver something

- This can be e.g.
    - Date to deliver a project to a customer (for a proj_... repo)
    - Just a date to put a stick in the ground, e.g. gen1_release which is pinned to a certain date
    - a certain month e.g. June
- Each story can only belong to one milestone
- Tasks belong to a story and as such also to a milestone
- Only use following naming conventions (no need to specify month nr's or year nr's)
    - nov_mid
    - nov_end
    - q4
    - future
- PLEASE be consistent with the names used otherwise presentation on a higher level layer becomes dificult.

DO NOT

- Use version milestones, e.g. js8.1

Product Management

- Product owners need to be well defined, this gets documented in `org_development` repo
- Regular meetings are super important see meetings
- Communication
    - Call for meetings on telegram channel, when info worth sharing about product group also use this
    - Telegram: `$company_product_$shortnameOfProductGroup`
        - e.g. gig_product_ovc
        - e.g. gig_product_g8os
        - e.g. gig_product_js

Product Management versus Product Owner

- Product Manager
    - is the middleman in the company
    - its like a project manager for a project but in this case for a set of products
    - owns relation towards sales/market
    - uses prod_ ... repo's in gogs
- Product Owner
    - Is a technical person who is part of engineering/development organization.
    - Makes sure that
        - roadmaps get implemented
            - milestones are properly set
        - repo's are properly managed
        - quality of the code is ok
    - uses coderepo's in github

Bugs & Feature Requests Funnel

IN: BUGS - FEATURES ->

OUT: STORIES

Principles

- The idea is that in a code repo you log all the bugs and feature requests
- Anyone can report bugs or feature requests
- The product owner will sort through the issues and give them priorities (using labels)
- The bugs or feature request are grouped and planned to be executed (attached to milestones = version nr's)
- There can be max 4 milestones (versions+roadmap) in a code repo, so easy to attach bug or FR to a version

How to link FR/Bugs to stories

- If you put $storycardname: at a prefix to the title of the issue then this bug/FR will automatically be linked to a story card.
- This only works if $storycardname is unique (which has to be for active stories)

- This is done by the Incubaid development process tools

Tickets, Tasks, Bugs & Feature Requests

What is what

- Ticket: new suggested types: customercase/issue
    - An issue raised by a customer or internal employee
    - Escalates issues like downtime, performance degradation, question, ...
    - Tickets are put on project repo's or support repo
- Bug
    - Issue of product
    - Only raised by support person or developer (gig or external if opensource)
    - If customers see a bug they normally will create a ticket first and the support person will create a bug and follow up
    - Is done on the code repository
    - If someone tumbles upon a blocking bug, the link to the issue needs to posted to the Bug Assesment Meeting (BAM) Telegram group so that the whole organization does not have to wait to the next BAM meeting for the assessment of the blokker
- Feature request (FR)
    - Raised by developers or anyone having interest in the code base (is done on the code repository), these feature requests are funneled and prioritized before they become story cards
- Tasks
    - Only used on `org_repo's` or `proj_repo's`!
    - The tasks are linked to the subject for which the task is (not the projecy where people belong too)
        - E.g. if our legal council needs to help on a contract in `proj_mauritius` then the task is in `proj_mauritius` NOT in `org_legal`

Bug/FR format:

Title: `$storyname:$taskDescription [4h]` or `$taskDescription [4h]`

- Time format is optional
- Can be in h or days e.g. 4h or 4d
- E.g.: `storyx:cuisine start of postgresql does not work [1h]`

Body for Feature Request:

- requirements for FR or link to requirements in same repo
- define why this is important for you (because you filed this FR)
- define required deadlines if relevant
- optionally: link to acceptance criteria or specify them here
- link to specs if any in the same repo
- remarks
- optional: time estimate see specs

Body for Bugs:

- screenshots if relevant
- on which environment this was seen
- link to logs if any
- remarks
- optional: time estimate see specs

Task format:

- IMPORTANT:
    - do not overuse tasks, this leads to abuse very quickly and is not really compatible with scrum or kanban
    - personally I would never use them and only put tasklists on story card level if you really feel the need for it
    - scrum is all about the story card owner having a good view on what the story is about and with gut feel & knowledge put best estimate in the title of remaining time (gutfeel is much more reliable then trying to put tasks in)
    - if you try to describe work by tasks its never correct and at least 50% error because tasks are never complete, in my personal optinion this simply does not work.
- DO NOT describe them too formal !!!!, needs to be a quick placeholder
- DO NOT Describe (all this belongs to code repo)
    - features

    - bugs
    - requirements
    - docs
    - remarks to do with product
- only content should be purely process driven

Title: `$storyname:$taskDescription [4h]`

- Time format is optional
- Can be in h or days e.g. 4h or 4d
- E.g.: `storyx:cleanup the text for button something [1h]`
- E.g.: `storyx:complete the autotest nr ... [1d]`

Body:

- Each task is linked to 1 owner who executes the task
- Sometimes its useful that the owner of the task puts a time estimate e.g. [1h], this is always the remaining time going from now. This needs to be revisited often

Stories are the Start of All

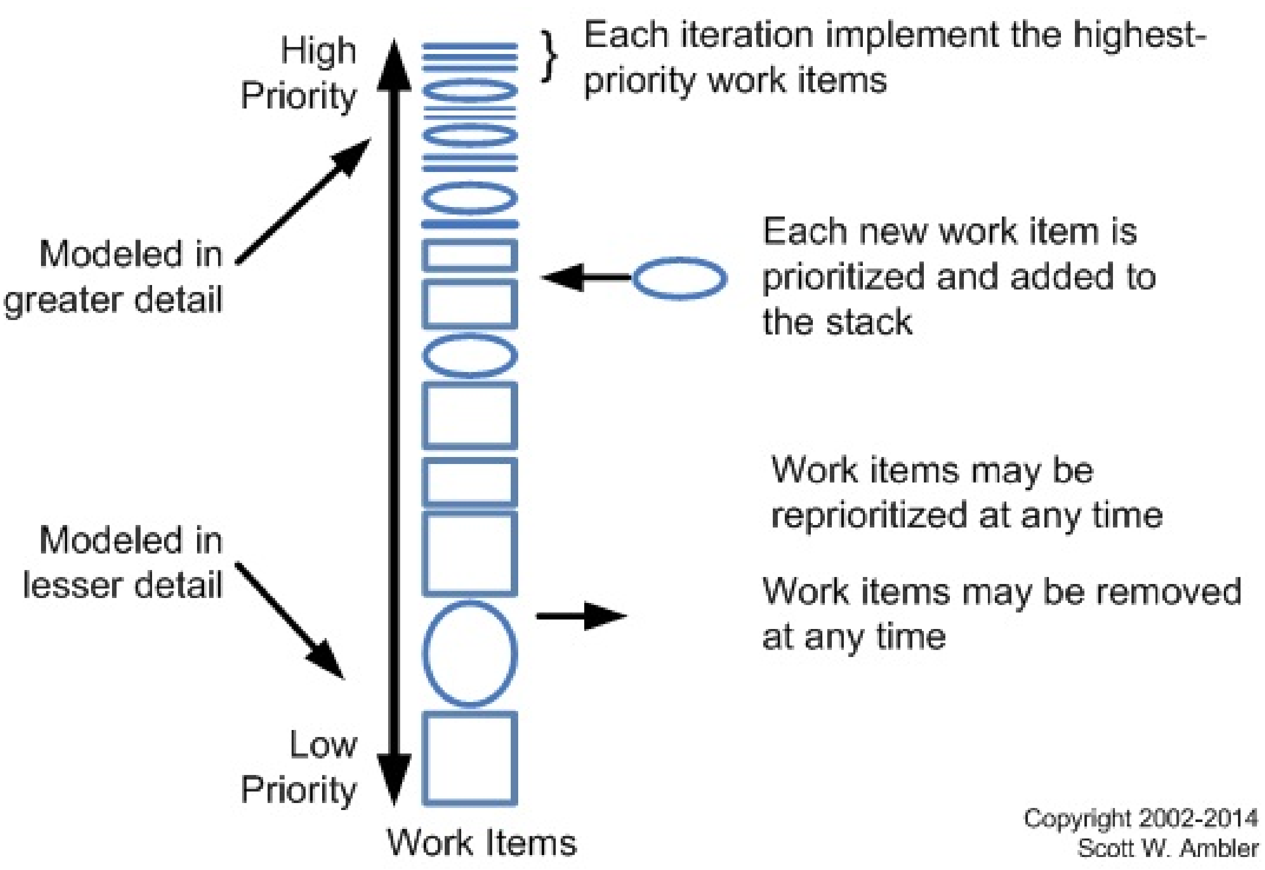
Stories help us to organize our work. They need to comply to certain requirements to be effective though.

A story normally represents the view of a person, e.g. "I as product owner for product X would like to see Y features to be implemented because... customer Z needs this... our strategy is..."

Properties of a good story

- Well chosen name
- Clear description of what we want to achieve
- Priority set a label:
    - Critical
    - Urgent
    - Normal
    - Minor
- Milestone: is a label, linking the story to a specific date e.g. nov_end (see milestones)
- Assignee: is the story card owner
- Estimation of the effort in days or hours (this needs to be redone on max 2 daily basis by story card owner)
    - only put this in title
- Links (URLs) to
    - Specifications (in code repo)
    - QA acceptance criteria (code repo)
    - Product Requirements Document (PRD) if relevant (core repo or home repo in product org/account)
    - Features or bugs which are related to the story (code repo)
    - Kanban about releated FR/Bugs (using version=milestone or prefix in BUG/FR)
- Format:
    - Title: `Story title` [`time left e.g.4d`] (`Story name`)
    - The story name is used as reference in Tasks related to the story

Stories can constantly be re-prioritized


High Priority
Low Priority
Modeled in greater detail
Modeled in lesser detail
Work Items
Each iteration implement the highest-priority work items
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Work items may be removed at any time
Copyright 2002-2014 Scott W. Ambler

Story format

Title: `$storyDescription [4d] ($storyname)`

- See specs for definition time estimates.
- The time estimate is in title, this is the remaining time for this story
  - this is only set by the story card owner !!!
  - this needs to be set on max 2 daily basis !!!
  - time is working time (sum over all people)
  - this is an estimate this is not the sum of the times used inside the story or specs underneith

Body

see specs for definition of task tables and time estimates.

```
## GOAL:

Clearly describe what needs to be achieved.

## DESCRIPTION
```

```
Describe what story is about.

## TASKS

- is an optional task table, see [specs](specs.md)
- i suggest not to use underlying tasks, you can but risk is high that people will
 use these tasks as feature requests or bugs, better just to use this table as a h
igh level overview and as a way how to calculate how much needs to be done.

## LINKS:

- Tasks: link to a Kanban board, using waffle.io, see [example here](https://waffl
e.io/gig-projects/org_development?search=js8beta:&label=type_task)
- Specs: link to the Markdown page holding the specifications for the story (code
repo)
- ...

## REQUIREMENTS:

- very high level overview and/or links to requirements (which are in code repo)
    - only the high level overview, the rest is in the code repo's

## REMARKS:

- List all remarks
```

When does a story card go into validation stage?

- For stories related to projects or customers:
    - When work or subproject is done and is ready for validation
- For storries related to code:
    - When code done
    - When autotests (if relevant) are created
        - When autotests are already executed and succesful.
    - When all documentation relevant to the story is done
        - Do not forget to document how to test the story card
        - Do not forget to document on how to install/build everything related with the story card
    - When all tests are done

When is a story card complete (closed)

- When validation was done by story card owner and other stakeholders

- Result should be that everyone is able to install and test the code/work relevant to the story, and this just by reading the story card

Use of Incubaid GitHub development tools

Our tools will automatically find the tasks and embed them in the story so it's very easy for people to follow what the linked tasks are and how far we are from completion.

Suggestions

- do not use documentation phase in kanban, documentation is a part executing on the story like any other task (specs, ...)
    - specs/documentation should 20% be done before you start with story and while doing story the specs/docs get completed

Labels #TODO: *1 the labels used are not in line with this doc, lets fix

Type

- To define type of issue
- 1 needs to be used (and only 1)!

Name	Description	Used in
story		home, org, project
task		home, org, project
ticket	type	org, project
bug	issue reported by anyone	code, ays, doc, www, quality
feature	improvement asked for by anyone	code, project, ays, doc, www, cockpit, org
question	anyone wants to ask something	home, code, project, ays, doc, www, cockpit
monitor	monitoring system found an issue	project, cockpit, www
lead	lead for sales, can convert in business	project, org
test	a test task	org, cockpit

Priority

- Define priority in which issue needs to resolved
- 1 needs to be used, and only 1!

Name	Description	Used in
critical		all
urgent		all
normal		all

minor		all

Process

- Define 2 optional process related labels

Name	Description	Used in
duplicate		all
wontfix		all

State

- Define state of issue in the Kanban flow
- 1 needs to be used, and only 1!
- Remark
  - These states need to be useful over all projects, e.g. over customer projects, code projects, stories, ...
  - Good filters should be used in waffle to represent a Kanban well

Name	Description	Used in
new	new or issue in backlog (is no label, is the default)	all
inprogress	people are working on it	all
question	needs input	all
verification	testing, verification with customer	all
closed	nothing to do, 100% done	all

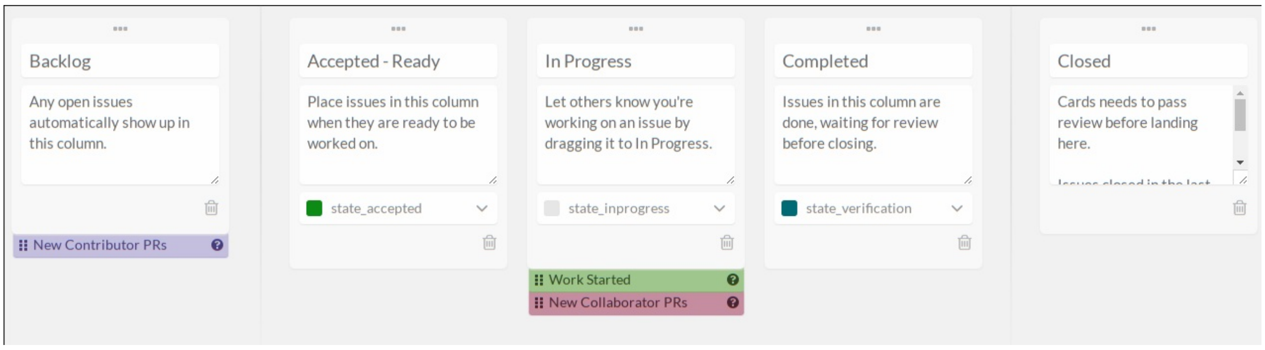
Visualize over github (only relevant for product related repo's)

- We suggest to use https://waffle.io/ to visualize this agile methodology
- This gives you a Kanban method on top of all your stories/bugs/fr's/tickets/...

How to use Waffle

Waffle present the issues and pull requests from one or more Github repository

We use 5 columns to categorize our works:



Backlog
Any open issues automatically show up in this column.
New Contributor PRs

Accepted - Ready
Place issues in this column when they are ready to be worked on.
state_accepted

In Progress
Let others know you're working on an issue by dragging it to In Progress.
state_inprogress
Work Started
New Collaborator PRs

Completed
Issues in this column are done, waiting for review before closing.
state_verification

Closed
Cards needs to pass review before landing here.
Issues closed in the last

Figure: waffle columns

- Waffle can visualize stories, tasks, bugs/feature requests
- Waffle's are attached to milestone projects
- The same Waffle Kanban can visualize all mentioned categories above, use filters to select
- Waffle also allows a user to see which tasks/stories are assigned to him over multiple repo's
- For each Waffle linked to milestone we also link all relevant code repositories so we can use the same waffle to see progress on FR's & BUGS

Stories

- **NEW**: Stories selected for milestone but no work started yet
- **PROGRESS**: Stories which are being worked on
- **QUESTION**: Stories which need feedback and need to be discussed

- **VALIDATION**: When a story is done, the next step is to review it; this is a vey important step in our process
- **CLOSED**: After review when the solution proposed is accepted, the story is closed

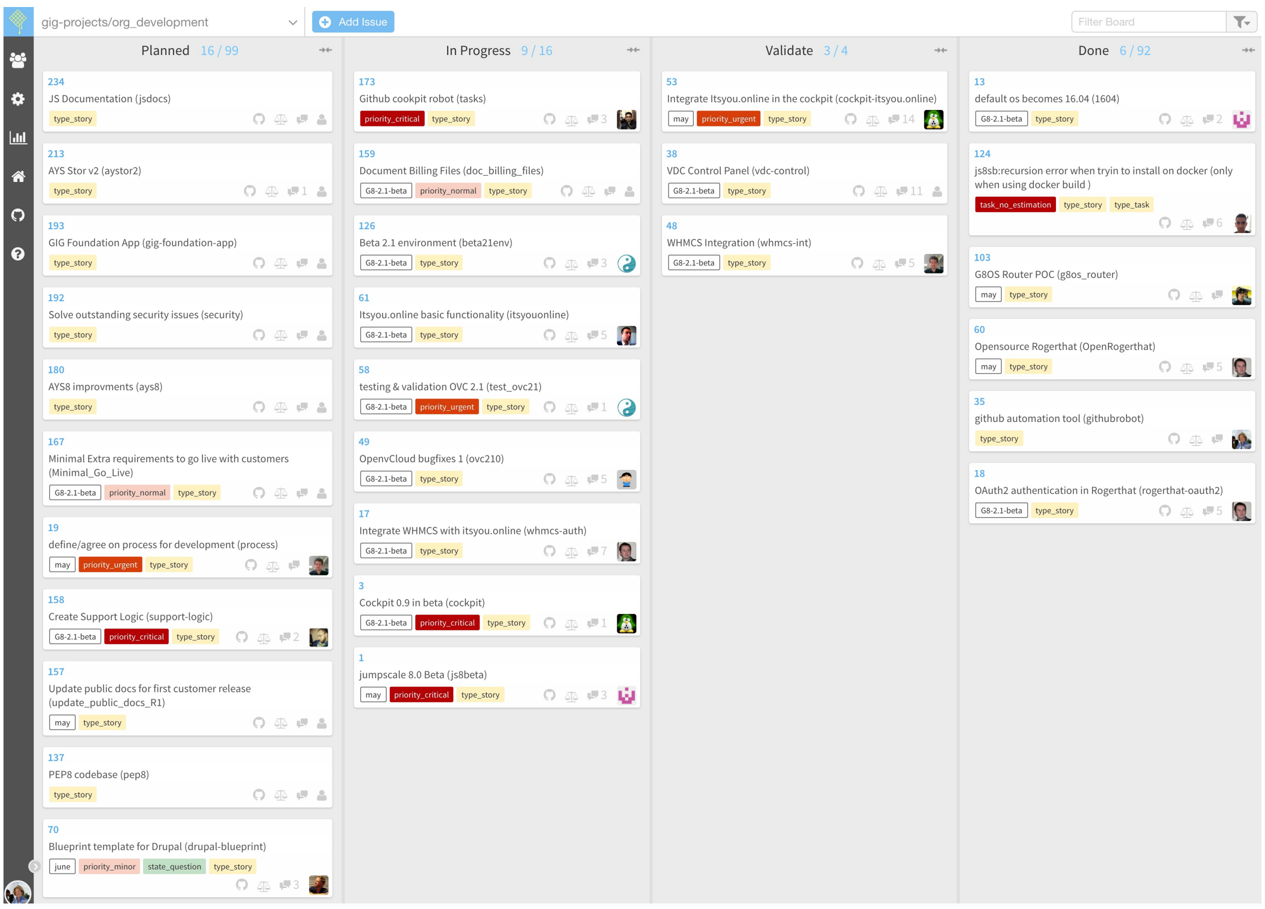
Tasks

- **NEW**: Tasks not started yet
- **PROGRESS**: When someone is working on a task, the task is known as in progress. By looking at that column we can track on what the team is focusing on at the moment
- **QUESTION**: Task is unclear it needs attention
- **VALIDATION**: optional, sometimes a task needs to be validated
- **CLOSED**: Done

Bugs and feature requests

- **NEW**: Unsorted bugs/fr
- **PROGRESS**: Active people are working on it
- **QUESTION**: Need feedback
- **VALIDATION**: Being tested
- **CLOSED**: Done

Example


gig-projects/org_development

Add Issue

Filter Board

Planned 16 / 99

234
JS Documentation (jsdocs)
type_story

213
AYS Stor v2 (aystor2)
type_story
1

193
GIG Foundation App (gig-foundation-app)
type_story

192
Solve outstanding security issues (security)
type_story

180
AYS8 improvments (ays8)
type_story

167
Minimal Extra requirements to go live with customers (Minimal_Go_Live)
G8-2.1-beta priority_normal type_story

19
define/agree on process for development (process)
may priority_urgent type_story

158
Create Support Logic (support-logic)
G8-2.1-beta priority_critical type_story
2

157
Update public docs for first customer release (update_public_docs_R1)
may type_story

137
PEP8 codebase (pep8)
type_story

70
Blueprint template for Drupal (drupal-blueprint)
june priority_minor state_question type_story
3

In Progress 9 / 16

173
Github cookpit robot (tasks)
priority_critical type_story
3

159
Document Billing Files (doc_billing_files)
G8-2.1-beta priority_normal type_story

126
Beta 2.1 environment (beta21env)
G8-2.1-beta type_story
3

61
Itsyou.online basic functionality (itsyouonline)
G8-2.1-beta type_story
5

58
testing & validation OVC 2.1 (test_ovc21)
G8-2.1-beta priority_urgent type_story
1

49
OpenvCloud bugfixes 1 (ovc210)
G8-2.1-beta type_story
5

17
Integrate WHMCS with itsyou.online (whmcs-auth)
G8-2.1-beta type_story
7

3
Cockpit 0.9 in beta (cockpit)
G8-2.1-beta priority_critical type_story
1

1
jumpscale 8.0 Beta (js8beta)
may priority_critical type_story
3

Validate 3 / 4

53
Integrate Itsyou.online in the cockpit (cockpit-itsyou.online)
may priority_urgent type_story
14

38
VDC Control Panel (vdc-control)
G8-2.1-beta type_story
11

48
WHMCS Integration (whmcs-int)
G8-2.1-beta type_story
5

Done 6 / 92

13
default os becomes 16.04 (1604)
G8-2.1-beta type_story
2

124
js8sb:recursion error when tryin to install on docker (only when using docker build)
task_no_estimation type_story type_task
6

103
G8OS Router POC (g8os_router)
may type_story

60
Opensource Rogerthat (OpenRogerthat)
may type_story
5

35
github automation tool (githubrobot)
type_story

18
OAuth2 authentication in Rogerthat (rogerthat-oauth2)
G8-2.1-beta type_story
5

Roles & Responsibilities

- The below roles need to be documented in the relevant proj or org repo's of a PROJORG (GitHub Project organization)
- It's basically describing who owns which role in the organization

Scrum Team Lead

- Owns a team of people working on one or more stories
- Scrum Lead needs to make sure
    - People understand Scrum and our Agile principles
    - People are coached
    - People improve in their career
    - People are happy in their job and are motivated
    - Eliminate all blockers which people might experience
- Scrum leads are often also story owners (of X nr of stories)
- Scrum teams have their own repo: `org_development_$scrumTeamname`
- Scrum teams are well defined in `org_development` repo

Project Owner

- Responsible for a `proj_$customer_$proj` or `proj_$company_$proj` repo

- Responsibility

    - Define milestones
    - Make sure the repo is clean
    - Verify / Check all stories as defined in repo (guarantees quality, priorities, communication around it, ...)
    - Work with product owners to make sure you get the features as required.
    - Organize project meetings (see Meetings to be defined #TODO: *2)
- There is only one project manager per repo

- in readme.md of repo there needs to be clear definition about project & owner & contact info

- They use their telegram group: `$company_proj_$customer_$proj`

Product Owners

- Responsible for a github/gogs account/organization e.g. jumpscale
    - They use the home repo to define the structure in the account
- Important to structure the accounts/organizations well so they represent a product
- Responsibility
    - Define milestones=versions on the repo's, make sure they are consistent
    - Make sure the repos in the account/organization are clean
    - Make sure everyone using the repo's behave well
        - proper defined bugs/fr
        - proper branching
    - Work with Project Managers & other company stakeholders to make sure the repo structure is ok.
    - Organize product meetings (PM) (see Meetings
- There can be more than 1 owner per repo, but try to only have 1.
- The home repo is important to organize the full account/organization.
- They use their telegram group: `$company_product_$name`

Product Manager

- Owns product from a go2market/commercial/marketing perspective.
- Defines roadmap.
- Organizes stakeholder meetings.
- Owns one or more repo's (prod_...) on gogs

Story owners (product related)

- Main responsibilities:
    - Story card is developed in time
    - Story card is clear and understood by all required
    - Requirements for the card are met
    - Done means done! (see Agile Principles)
    - Communicate to product owners about the card
    - Organize story meetup in order to talk with people contributing to the story, follow-up on progress
    - Define tasks that need to be delivered to deliver the story card

What are the duties of a story owner

- Format of story is fully compliant
- Content of story is properly done (description, requirements, ...)
  - Requirements are clear
  - All involved people understand the story properly
- Be part of product meetings (PM's), give info:
  - How far are we
  - What are the blockers
  - What is planned now
  - Do this by commenting on story card
- Escalate to product owner(s) & Scrum Owner when
  - Story has delay
  - There are questions
  - Do this by comments and also use relevant (e.g. Telegram) communication group to escalate to appropriate people
- Guarantee Story gets done which means
  - Code is complete
    - Features are in line with requirements
  - Tests are good enough
  - Good documentation that can be understood also by someone who did not work on the story
  - On validation phase the pull request is properly merged in (after validation of course)
  - Code can easily be build and installed by anyone
  - Everyone relevant understands story is done and declare victory!

Stakeholder

- Anybody who has something to loose or win by the acceptance/progress on a story card
- Stakeholder main responsibility is to be part of product stakeholder meeting (PSM)
- This is a very generic definition & can be quite broad.

Meetings

Organization/Project Stakeholders meeting

- Organized by organization owners (teamleads) or project managers
    - Can be a regular call e.g. once per week
    - He/she feels communication between stakeholders is required to meet the deadlines as defined
    - Update progress towards the deadline
- Who
    - manager of this organization or this project
    - all stakeholders who have something to lose or win by stories discussed on this meeting
- When
    - anytime when company priorities or deadlines need to be discussed
- Use waffle.io kanboard representation to see prio's
- Communicate using telegram in organization/project group
    - in `$company_org_marketing`

Product Demo meeting

- When 1 or more stories are ready do be demo'd then milestone owner or story card owner can decide to hold a demo meeting
- Who
    - Anyone who is interested to learn about progress made on a story card
- Goal
    - Show progress made
    - Demonstrate the story(ies) is(are) "DONE"
    - Show people the starting points so they can repeat it themselves
- Communicate using telegram in product group
    - in `$company_org_product_...`

Scrum meeting

Only relevant for product development.

- meeting done per scrum team
- daily basis
- max 15 min
- they go over all open stories (which is normally max 3 per any point in time)
- Each scrum team member should answer:
    - What did I do since last meeting
    - Did I meet all my estimates and goals
    - What am I planning to work on
    - Estimates regarding my planned work
    - What is blocking me
    - What is remaining effort from my side for completion of this full story card
- Communicate using telegram in product group
    - in `$company_org_development_$scrumTeamName`

Master Scrum Meeting

- the scrum leads come together to discuss progress & remove blocking items.

Product Meeting (PM)

Only relevant for product development.

- Is combination of product stakeholder meeting, story master & scrum meeting from before.
- Is done per product group (see product management)
    - example product groups
        - ovc / g8os / itsyou.online / oneapp / jumpscale
    - these product groups need to be defined in `org_development` repo in github/gogs
        - is important everyone understands which product groups there are
    - each product group needs to have a clear owner
- 3 goals
    - assess progress per product group and see if milestones will be met
    - discuss blocking items (critical path) in relation to progress (any blocking

    potential item)
    - assess blocking bugs (means do we break the current iteration and add some bug to be resolved now)
- When
    - 1-5 days, means daily for fast changing products where deadlines are close
- Who
    - all storycard masters which are linked to product group
    - each stakeholder who wants to be detailed involved in the specific product
    - once every month (at least once): more people are involved to attend the product meeting this to discuss and make sure
- Preparation
    - these meetings need to be well prepared to make them efficient, prepared by product owner
- Communication
    - Call for meetings on telegram channel, when info worth sharing about product group also use this
    - Telegram: `$company_product_$shortnameOfProductGroup`
        - e.g. gig_product_ovc
        - e.g. gig_product_g8os
        - e.g. gig_product_js
- Use waffle.io kanboard representation to see prio's

Product Stakeholders Meeting (PSM)

- normally happens over multiple product groups at once (sometimes can be 1)
- 2 goals:
    - assess how we did and if needed move stories to next milestone (stakeholder meeting)
    - as preparation on next milestone: agree on which stories need to go in which milestone (stakeholder meeting)
- When
    - decided in PM meeting, if they feel its needed to bring all stakeholders together
        - can be because of unforeseen delay, can be planning for next release
    - whenever org owner of company believes this is required
- Who

    - Everyone in company who has something to lose or win by delaying or prioritizing stories
- What this is not!!!
    - No discussions about content: FR/BUGS, Requirements, ... all of this needs to be properly prepared
- Principles
    - These meetings need to be super efficient.
        - Normally lots of people are on these meetings that is why its super important to keep them dense and short.
        - Preparation is done by product owners & in PM meetings, THIS NEEDS TO BE DONE WELL.
            - Don't call for a PSM if no proper preparation done
    - If story not well prepared, it gets skipped and needs to wait for next PSM.
- Communication
    - Telegram: `$company_product_stakeholders`
    - There needs to be a PSM per product group, is ok to do more at once

Usage of telegram

- Telegram (and later Rogerthat) is our main tool to let communication flow
- In this document we describe the main Telegram groups and how we use them
- IMPORTANT: PEOPLE NEED TO USE 2FACTOR AUTHENTICATION !!!!

Product

- `$shortcompanyname_product_$name` , e.g. `gig_product_jumpscale`
- Used for
    - all communication around this product group
    - calling for product stakeholder meetings (PSM)
    - calling for product meetings (PM)
- Who can join
    - anyone who is interested to understand more about a product group

Support

- `$shortcompanyname_support` , e.g. `gig_support`
- Used for
    - Any customer isue which deserves attention from more than 1 person
    - Escalations
    - Information sharing
    - Planning events
- Who is on it
    - Everyone delivering support
    - Everyone who wants to know what is going on
- IMPORTANT
    - On duty support guys
        - Need to announce their duty ON & OFF event on this group (if less than 10 people active, for larger orgs this will not be practical). This will allow people to see how is online & active.
        - Need to use and actively look at this group

Organization

- `$shortcompanyname_org_$name`

- Example groups:

  - e.g. `gig_org_marketing`
  - e.g. `gig_org_products`
  - e.g. `gig_org_sales`
  - e.g. `gig_org_internalit`
  - e.g. `gig_org_development`
  - e.g. `gig_org_research`
  - e.g. `gig_org_proddelivery` : e.g. hardware & shipments of projects
  - e.g. `gig_org_qa` : quality assurance in broad sense
  - e.g. `` `gig_org_operations `` : team which keeps our systems up & running in broad sense
  - e.g. `gig_org_admin`
  - e.g. `gig_org_finance`
  - e.g. `gig_org_legal`
  - e.g. `gig_org_funding`

  - Normally the name used is the same as the name of the git repo

- Used for
  - Communication around the topic
  - Planning, info changes, ...
- Who is on it
  - Everyone which is working on the topic or can needs to know about it

Project

```
- E.g. ```gig_proj_mauritius```
- Normally the name used is the same as the name of the GitHub repo
```

- Used for
  - Communication around such a project
  - Planning, info changes, ...
- Who is on it
  - Everyone which is working on this project

Cockpit

- `$shortcompanyname_cockpit_$name`

- In line with name of Cockpit GitHub repo
- Used for
  - all communicationa around 1 IT environment (can be for 1 customer or internal, ...)
- Also used by Cockpit: AYS Robot
  - Means we can see on this group if there are issues with the environment
  - Or we can give instructions to the AYS Robot to get things done
- E.g. following is done per test environment
  - Can see if someone is doing a test and what the result is
  - Can see monitoring state
  - Can request changes
- Who can use it
  - People with right security clearance, can be partners & internal

0-Bugs

Principle how to sort through bugs and keep the process as light as possible.

Blog article from VMware Israel team

Blog from development team at VMware Israel (small mods to fit in our process)

How many bugs do you manage? 50? 200? 2000?
Maybe you cannot even tell what the exact number is, because it keeps changing.
I know exactly how many bugs we have in our product: zero.
How much time do you spend in bug triage meetings and on bug management (moving them around from version to version)?
I spend half an hour once a month. And I never see the same bug more than once.
Often, we wonder how should we handle bugs when working in scrum/agile?
The answer is 0 bugs policy. It is based on our experience in several of the Scrum teams in VMware Israel.
Other ways we tried failed, with 0 bugs policy it just works.
The following blog is a manifest that describes what is the 0 bugs policy and why it is the only way to go.
The 0 bugs policy is an advanced yet very simple process for handling bugs.
Whenever you encounter a new bug, you should either fix that bug, or close it as "won't fix" and don't think about it again. That's it. Simple.

Now, let’s talk about why this is the only way to go.

Bugs can be generalized into 2 categories:

1. Bugs that were opened during the current iteration for user stories (features you are now implementing) - Fix them right away, otherwise the story/feature is not really DONE.
    - If this concept is not crystal clear, than this blog is not for you.
2. All the other bugs (regression, customer bugs, etc.). Non-sprint bugs.
3. You can either fix them right away (or in the next sprint; Poteto potato).
4. You can close them as "won't fix" - if the value of the fix does not worth the effort to fix it.
5. You can defer the bug.

Let's talk about deferring the bug. Just DON'T. If you don't fix it right now, just close it.

Why?

It will cost you more to fix it later... a lot more... tons more!

Because a sprint from now:

- You will not remember nor understand this bug as good as now.
- The right environment - both for QA and for dev. – might not be available.
- The code might change and the behavior will be slightly different.
- 2-3 sprint from now (not to say the next release) you will not remember it at all and you will have to spend a lot of time understanding the bug and reproduce it.
- You will need to maintain the deferred bugs, either if they are managed in their own backlog or part of the main backlog. You will need to prioritize them, and that takes time.
- Also, it is annoying. Have you ever been part of a bug triage meeting? It’s a day-killer. Trust me.
- You will never – and this is a known secret – fix this bug.
- It will be in your system forever. Why?
    - If you have decided not to fix it now, it means you have more important things ahead. Features. Probably. Trust me. You will always have more features. Even in the next release (-;.
    - As the time will go by, you will defer more and more bugs, and then this bug will have to compete both against new and cool features and also with new and not cool bugs. So, if you didn't fix it now, when you don't have all the other important bugs as well, why the hell do you think you would do it later?

OK. So we have agreed that the 0 bugs policy is in fact GREAT!

Our conclusions

- Each product/code account has milestones defined = product versions
- Per code repo there are normally not more than 3/4 milestones
    - Version number 1 (nearest to today)
    - Version number 2 (next release)
    - Roadmap (always use this name!)
- **Bugs are NEVER put on roadmap**, following this page they are in version 1 or 2 after now.
- FR can be put on roadmap when not in one of 2 next releases, but no specific timing yet.

Planning (#TODO: *1 document needs review, not ok)

NOTOK

Introduction

Planning always happens in an **organization** repository, such as:

- org_development (engineering)
- org_support
- org_quality
- org_internalit
- org_marketing
- org_product
- org_finance
- org_legal
- org_hr

Remember that organization repositories should only be created in GitHub organizations of type "projorg" such as gig-projects.

Also in case of planning related to product development, the planning should only happen in an organization repository, even while the actual code exists in a **code** repository, which can only exist in GitHub organizations of type "codedorg" such as jumpscale.

So in other words, for all product development there will always be two repositories:

- One code repository in a "codedorg" organization where the actual **code** exists
- One organization repository in a "projorg" organization where the **planning** happens

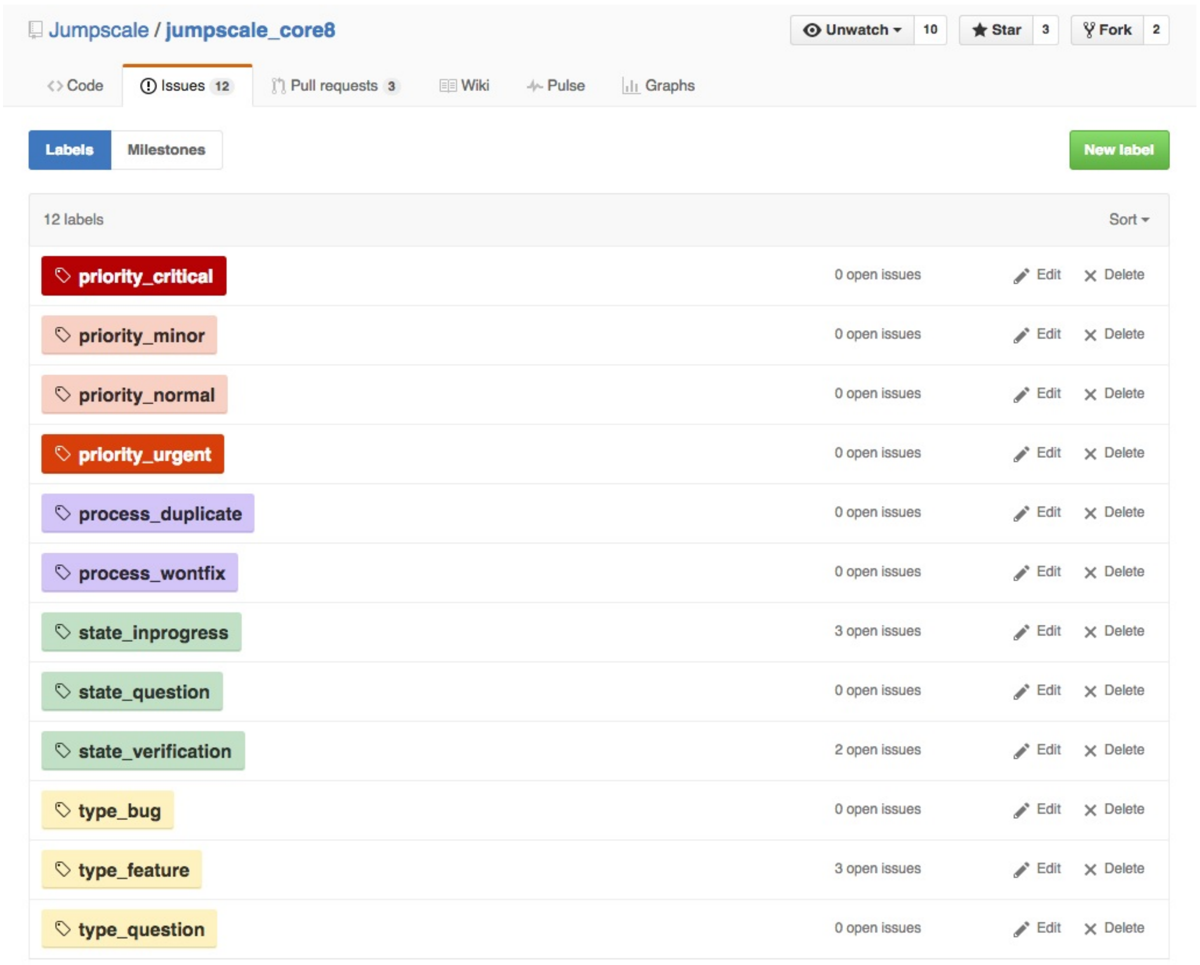
As a consequence you will have two "types" of milestones involved:

- A **non-time based** milestone in the code repository - here we typically use a release number such as "8.1"
- A **time based** milestone in the organization repository - here we use specific names like `nov_end`

In what follows we discuss the steps for planning related to product development.

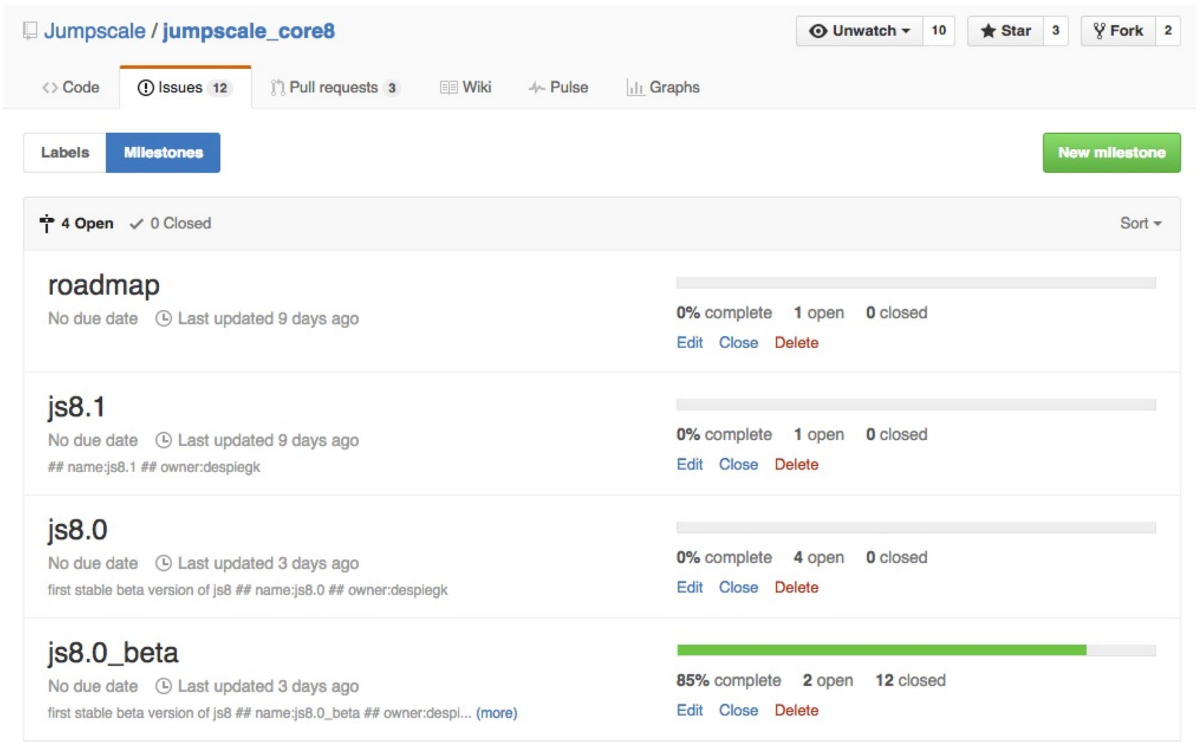
Step 0: Prepare

- Per company there is a GitHub organization of type "projorg", e.g. gig-projects
  - Here you will have an organization repository, e.g. gig-projects/org_development
- Per group of products there is GitHub organization of type "codedorg", e.g. jumpscale
  - Here you will have/create one or more code repositories, e.g. Jumpscale/jumpscale_core8
  - Make sure all your code repositories are well prepared, having the right labels and milestone:



12 labels

priority_critical	0 open issues	Edit	Delete
priority_minor	0 open issues	Edit	Delete
priority_normal	0 open issues	Edit	Delete
priority_urgent	0 open issues	Edit	Delete
process_duplicate	0 open issues	Edit	Delete
process_wontfix	0 open issues	Edit	Delete
state_inprogress	3 open issues	Edit	Delete
state_question	0 open issues	Edit	Delete
state_verification	2 open issues	Edit	Delete
type_bug	0 open issues	Edit	Delete
type_feature	3 open issues	Edit	Delete
type_question	0 open issues	Edit	Delete



Jumpscale / jumpscale_core8

4 Open ✓ 0 Closed

Milestone	Progress	Open	Closed
roadmap — No due date, Last updated 9 days ago	0% complete	1 open	0 closed
js8.1 — No due date, Last updated 9 days ago — ## name:js8.1 ## owner:despiegk	0% complete	1 open	0 closed
js8.0 — No due date, Last updated 3 days ago — first stable beta version of js8 ## name:js8.0 ## owner:despiegk	0% complete	4 open	0 closed
js8.0_beta — No due date, Last updated 3 days ago — first stable beta version of js8 ## name:js8.0_beta ## owner:despi... (more)	85% complete	2 open	12 closed

- Use our tools to set the labels, ...

- Make sure all people have access

Step 1: Roadmap / strategic planning

- In the home repository of the "projorg" organization
    - Update README.md (`$homerepo/Readme.md`)
        - Link to all documents mentioned below
    - Create terminology document (`$homerepo/terminology.md`)
        - Define all relevant terms related to your products
    - Create components document (`$homerepo/components.md`)
        - Define the components which make up the products
        - Describe what all these components do
        - Include links to the associated code repositories
- In org_development or org_research of the "projorg" organization
    - Define roles
    - Define milestones (time based!)
    - Explain why the milestones
    - Define owners of the milestones
- In proj_... of the "projorg" organization
    - Define owner (only 1 role = project owner)

    - Define milestones (time based!)
    - Explain why the milestones
    - Define team
- Telegram group (optional)
    - Create telegram group with same name as repo prepended with short company name e.g. gig_org_development
    - Make sure right people have access to it
    - All relevant stakeholders & story card owners

Step 2: Story card preparation

- Is an ongoing process
- Stories should only exist in a GitHub organization of type "projorg"
- Create all story cards relevant for 1 or more milestones
    - Put the story cards in the right milestones!
- Choose the story card owners wisely
- Use Telegram: `$shortcompanyname_milestone_$name`

Step 3: Stakeholder meeting (see meetings doc)

- Call stakeholder meeting to discuss milestone(s)
- Use Telegram: `$shortcompanyname_milestone_$name`
- When approved, announce over Telegram
- When not approved do more of these meetings, go back to step 2 when required

Step 4: Milestone meeting (see meetings doc)

- Present milestone and related story cards to everyone involved in the relevant organization
- This is the official start of getting this milestone done
- Use Telegram: `$shortcompanyname_milestone_$name`

Step 5: Story group meetings (see meetings doc)

- The story owners organize meetings to discuss progress
- Happens every day, needs to be very short < 15 min
- Use Telegram: `$shortcompanyname_milestone_$name`
- Use `gig_story_$shortStoryName` to communicate specifically about 1 story when this

becomes relevant (is optional)

Step 6: Demo meeting (see meetings doc)

- Give demo's about what has been accomplished
- Use telegram: `$shortcompanyname_milestone_$name`

Step 7: Milestone acceptance meeting (see meetings doc)

- Discuss how the milestone went, learn for future
- Use Telegram: `$shortcompanyname_milestone_$name`

Coding Cycle

Step 1: Story

- Story needs to be defined in milestone
- First requirements done
- Feature requests/bugs are created on code repositories

Step 2: Issues

- Every FR/Bugs should have been assigned to a specific version/realease by the product owner.
- Product owner might put original estimates on issues.
- On accepting an issue, the developer should amend/set the estimate.
- Estimates can always be updated according to findings. Comments must be made as to why.
- Issue should be put *in progress* (by applying the appropriate label) once started.

Step 3: Branch

- The branch version should have been created by the product owner.
- create a local branch on which you are going to fix a bug or implement a feature.
    - this local branch should branch off from the version branch on the origin remote.
    - the local branch must be rebased often from the base version branch. It's the developer's responsibility to always make sure their branch doesn't steer away from the version branch.

Step 4: Code

Important : See coding style

- Only produce code that is related to the bug or FR you're working on.
- Don't mix two bug fixes or features in your local branch!
- write the tests related to the code you create
- write the documentation of the feature you implement

Step 5: Commit

- Once your code is ready, the **tests are written and the documentation is good enough**, push your local branch to remote and create a pull request. Your pull request should ask to merge your branch to the version branch, never to master. (check branching strategy for more details)
- Include the reference to the issue you're solving in your commit
  - E.g. "Implemented live migration cockpit: #10"

Step 6: Auto test perfome by CI

- If the project you're working on has a CI enabled, make sure that all the test passes and that your branch doesn't have conflicts.
- People won't even start reviewing your code if the tests don't pass or if your branch is out of date.

Step 7: code reviews

- Your code will be reviewed by your peer.
- disscusion can happen on the pull request to see how to solve best a problem or if someone has a better solution to the problem.
- restart the cycle from step 3 to step 5 till everyone agree on the code produced.

Step 8: Validate

- The code normally should be tested already by step 4 and 5
- Now the issue is put on *validate* state
- The story owner will finally accept the pull request
- This will merge the pull request back into the version branch.
- delete your branch.

Coding Style and convention.

Contributing to open source project implies to working with different people that have different habits. In order to not loose time in useless disscusion about coding style we describe here the convention we want to enforce in our code repositories.

Always use a linter

Following the convention of the language you are using helps other people to understand your code and make code easier to read.

Linter does more then just make the code easier to read, more idiomatic but it can often warn you about dangerous construct or buggy code.

For Atom:

- python: https://github.com/AtomLinter/linter-pylint
- go: https://github.com/joefitzgerald/gometalinter-linter

Format your code

Write clean code. Universally formatted code promotes ease of writing, reading, and maintenance.

- For Go Always run `gofmt -s -w file.go` on each changed file before committing your changes. Most editors have plug-ins that do this automatically.
  For Atom: https://github.com/joefitzgerald/gofmt

- For Python we choose to follow the PEP8 style guide
  Always run `autopep8 -a --max-length 120 file.py` on each changed file before committing your changes. Most editors have plug-ins that do this automatically.
  For Atom: https://github.com/markbaas/atom-python-autopep8

Pull request

Pull request descriptions should be as clear as possible and include a reference to all the issues that they address.

Commit messages must start with a capitalized and short summary (max. 50 chars) written in the imperative, followed by an optional, more detailed explanatory text which is separated from the summary by an empty line.

Code review comments may be added to your pull request. Discuss, then make the suggested modifications and push additional commits to your feature branch. Post a comment after pushing. New commits show up in the pull request automatically, but the reviewers are notified only when you comment.

Pull requests must be cleanly rebased on top of the version branch without multiple branches mixed into the PR.

Git tip: If your PR no longer merges cleanly, use rebase master in your feature branch to update your pull request rather than merge master.

Before you make a pull request, squash your commits into logical units of work using git rebase -i and git push -f. A logical unit of work is a consistent set of patches that should be reviewed together: for example, upgrading the version of a vendored dependency and taking advantage of its now available new feature constitute two separate units of work. Implementing a new function and calling it in another file constitute a single logical unit of work. The very high majority of submissions should have a single commit, so if in doubt: squash down to one.

Include an issue reference like Closes #XXXX or Fixes #XXXX in commits that close an issue. Including references automatically closes the issue on a merge.

Make sure your code pass the test suite:

After every commit, make sure the test suite passes. Include documentation changes in the same pull request so that a revert would remove all traces of the feature or fix.

Agile Principles

We distiled our own list of agile principles, we got lots of inspiration & text from here.

Because we are not 100% scrum or 100% kanban not everything is 100% relevant, that is why distilled our own list.

80% rule

Pareto’s law is more commonly known as the 80/20 rule. The theory is about the law of distribution and how many things have a similar distribution curve. This means that *typically* 80% of your results may actually come from only 20% of your efforts!

Pareto’s law can be seen in many situations – not literally 80/20 but certainly the principle that the majority of your results will often come from the minority of your efforts.

So the really smart people are the people who can see (up-front without the benefit of hind-sight) *which* 20% to focus on. In agile development, we should try to apply the 80/20 rule, seeking to focus on the important 20% of effort that gets the majority of the results.

If the quality of your application isn’t life-threatening, if you have control over the scope, and if speed-to-market is of primary importance, why not seek to deliver the important 80% of your product in just 20% of the time? In fact, in that particular scenario, you could ask why you would ever bother doing the last 20%?

Now that doesn’t mean your product should be fundamentally flawed, a bad user experience, or full of faults. It just means that developing some features, or the richness of some features, is going the extra mile and has a diminishing return that may not be worthwhile.

So does that statement conflict with my other recent post: “done means DONE!”? Not really. Because within each Sprint or iteration, what you *do* choose to develop *does* need to be 100% complete within the iteration.

It’s also worth considering the impact on user experience. Google has shown us that users often prefer apps that do just what you want. That’s *just* what you want. And no more. The rest is arguably clutter and actually interferes with the user experience for only a limited benefit to a limited set of users.

So in an agile development world, when you’re developing a brand new product, think very hard about what your app is *really* all about. Could you take it to market with all the important features, or with features that are less functionally rich, in a fraction of the time?

Apart from reduced cost, reduced risk and higher benefits by being quicker to market, you also get to build on the first release of the product based on real customer feedback.

So all of this is really common sense I suppose. But it’s amazing how often development teams, with all the right intentions, over-engineer their solution. Either technically, or functionally, or both.

The really tough question, however, is can you see up-front *which* 20% is the important 20%? – the 20% that will deliver 80% of the results. In very many cases, the answer sadly is no.

original content from here

Change Is Not Bad

This is in stark contrast to a traditional development project, where one of the earliest goals is to capture all known requirements and baseline the scope so that any other changes are subject to change control.

Traditionally, users are educated that it’s much more expensive to change or add requirements during or after the software is built. Some organisations quote some impressive statistics designed to frighten users into freezing the scope. The result: It becomes imperative to include everything they can think of – in fact everything they ever dreamed of! And what’s more, it’s all important for the first release, because we all know Phase 2’s are invariably hard to get approved once 80% of the benefits have been realised from Phase 1.

Ironically, users may actually use only a tiny proportion of any software product, perhaps as low as 20% or less, yet many projects start life with a bloated scope. In part, this is because no-one is really sure at the outset which 20% of the product their users will actually use. Equally, even if the requirements are carefully analysed and prioritised, it is impossible to think of everything, things change, and things are understood differently by different people.

Agile Development works on a completely different premise. Agile Development works on the premise that requirements emerge and evolve, and that however much analysis and design you do, this will always be the case because you cannot really know for sure what you want until you see and use the software. And in the time you would have spent analysing and reviewing requirements and designing a solution, external conditions could also have changed.

So if you believe that point – that no-one can really know what the right solution is at the outset when the requirements are written – it’s inherently difficult, perhaps even practically impossible, to build the right solution using a traditional approach to software development.

Traditional projects fight change, with change control processes designed to minimise and resist change wherever possible. By contrast, Agile Development projects accept change; in fact they expect it. Because the only thing that’s certain in life is change.

There are different mechanisms in Agile Development to handle this reality. In Agile Development projects, requirements are allowed to evolve, but the timescale is fixed. So to include a new requirement, or to change a requirement, the user or product owner must remove a comparable amount of work from the project in order to accommodate the change.

This ensures the team can remain focused on the agreed timescale/milestones, and allows the product to evolve into the right solution. It does, however, also pre-suppose that there’s enough non-mandatory features included in the original timeframes to allow these trade-off decisions to occur without fundamentally compromising the end product.

So what does the business expect from its development teams? Deliver the agreed business requirements, on time and within budget, and of course to an acceptable quality. All software development professionals will be well aware that you cannot realistically fix all of these factors and expect to meet expectations. Something must be variable in order for the project to succeed. In Agile Development, it is always the scope (or features of the product) that are variable, not the cost and timescale.

Although the scope of an Agile Development project is variable, it is acknowledged that only a fraction of any product is really used by its users and therefore that not all features of a product are really essential. For this philosophy to work, it’s imperative to start development (dependencies permitting) with the core, highest priority features, making sure they are delivered in the earliest iterations.

Unlike most traditional software development projects, the result is that the business has a fixed budget, based on the resources it can afford to invest in the project, and can make plans based on a launch date that is certain.

original content from here

How do you eat an elephant? One bite at a time!

Likewise, agile development projects are delivered in small bite-sized pieces, delivering small, incremental *releases* and iterating (we call them milestones which do consist out of X stories).

In more traditional software development projects, the (simplified) lifecycle is Analyse, Develop, Test – first gathering all known requirements for the whole product, then developing all elements of the software, then testing that the entire product is fit for release. In agile software development, the cycle is Analyse, Develop, Test; Analyse, Develop, Test; and so on… doing each step for each feature, one feature at a time.

Advantages of this iterative approach to software development include:

- Reduced risk: clear visibility of what's completed to date throughout a project
- Increased value: delivering some benefits early; being able to release the product whenever it's deemed good enough, rather than having to wait for all intended features to be ready
- More flexibility/agility: can choose to change direction or adapt the next iterations based on actually seeing and using the software
- Better cost management: if, like all-too-many software development projects, you run over budget, some value can still be realised; you don't have to scrap the whole thing if you run short of funds

For this approach to be practical, each feature must be fully developed, to the extent that it's ready to be shipped, before moving on.

Another practicality is to make sure features are developed in *priority* order, not necessarily in a logical order by function. Otherwise you could run out of time, having built some of the less important features – as in agile software development, the timescales are fixed.

Building the features of the software ”broad but shallow” is also advisable for the same reason. Only when you've completed all your must-have features, move on to the should-haves, and only then move on to the could-haves. Otherwise you can get into a situation where your earlier features are functionally rich, whereas later features of the software are increasingly less sophisticated as time runs out.

Try to keep your product backlog or feature list expressed in terms of use cases, user stories, or features – not technical tasks. Ideally each item on the list should always be something of value to the user, and always deliverables rather than activities so you can ‘kick the tyres’ and judge their completeness, quality and readiness for release.

original content from here

Fast But Not So Furious

Agile development is all about frequent delivery of products. In a truly agile world, gone are the days of the 12 month project. In an agile world, a 3-6 month project is strategic!

Nowhere is this more true than on the web. The web is a fast moving place. And with the luxury of centrally hosted solutions, there's every opportunity to break what would have traditionally been a project into a list of features, and deliver incrementally on a very regular basis – ideally even feature by feature.

On the web, it's increasingly accepted for products to be released early (when they're basic, not when they're faulty!). Particularly in the Web 2.0 world, it's a kind of perpetual beta. In this situation, why wouldn't you want to derive some benefits early? Why wouldn't you want to hear real user/customer feedback before you build 'everything'? Why wouldn't you want to look at your web metrics and see what works, and what doesn't, before building 'everything'?

And this is only really possible due to some of the other important principles of agile development. The iterative approach, requirements being lightweight and captured just-in-time, being feature-driven, testing integrated throughout the lifecycle, and so on.

So how frequent is *frequent*?

Scrum says break things into 30 day Sprints. That's certainly frequent compared to most traditional software development projects. We break things up in milestones which also happen each couple of weeks, it has the same goal as classic scrum.

Consider a major back-office system in a large corporation, with traditional projects of 6-12 months+, and all the implications of a big rollout and potentially training to hundreds of users. 30 days is a bit too frequent I think. The overhead of releasing the software is just too large to be practical on such a regular basis.

30 days is a lifetime!

Competitors won't wait. Speed-to-market is a significant competitive edge. The value of first-mover advantage is potentially enormous. Whilst it's not always the case, research shows that those first to market 80% of the time win; and end up clear market leaders.

So how frequent is *frequent enough*?

A regular release cycle also allows you to learn more effectively. Your estimating might be good, it might be bad. Hopefully it's at least consistent. If you estimate features at a granular level (ideally less than 1 day) and track your velocity (how much of your estimate you actually delivered in each Milestone), in time you'll begin to understand your *normal* delivery rate. And when you understand this well, you'll be surprised how predictable you can be.

And let's face it, managing expectations is really all about predictability. If people know what to expect, they're generally happy. If they don't, they're not happy. Maybe even furious!

So, in agile development, focus on frequent delivery of products. And perhaps even more importantly, focus on consistent delivery of products.

original content from here

In agile development, “done” should really mean “DONE!”.

Features developed within an iteration (Milestone in our process), should be 100% complete by the end of the Sprint.

Stories on fixed should be 100% done.

Too often in software development, “done” doesn’t really mean “DONE!”. It doesn’t mean tested. It doesn’t necessarily mean styled. And it certainly doesn’t usually mean accepted by the product owner. It just means developed. THIS IS NOT GOOD!!!

In an ideal situation, each iteration or milestone should lead to a release of the product. Certainly that’s the case on BAU (Business As Usual) changes to existing products. On projects it’s not feasible to do a release after every milestone, however completing each feature in turn enables a very precise view of progress and how far complete the overall project really is or isn’t.

So, in agile development, make sure that each feature is fully developed, tested, styled, and accepted by the product owner before counting it as “DONE!”. And if there’s any doubt about what activities should or shouldn’t be completed within the Sprint for each feature, “DONE!” should mean shippable.

The feature may rely on other features being completed before the product could really be shipped. But the feature on its own merit should be shippable. So if you’re ever unsure if a feature is ‘done enough’, ask one simple question: “Is this feature ready to be shipped?”.

It’s also important to really complete each feature before moving on to the next…

Of course multiple features can be developed in parallel in a team situation. But within the work of each developer, do not move on to a new feature until the last one is shippable. This is important to ensure the overall product is in a shippable state at the end of the Sprint, not in a state where multiple features are 90% complete or untested, as is more usual in traditional development projects.

In agile development, “done” really should mean “DONE!“.

original content from here

Agile Development Teams Must Be Empowered

An agile development team must include all the necessary team members to make decisions, and make them on a timely basis.

Active user involvement is one of the key principles to enable this, so the user or user representative from the business must be closely involved on a daily basis.

The project team must be empowered to make decisions in order to ensure that it is their responsibility to deliver the product and that they have complete ownership.

The team must establish and clarify the requirements together, prioritise them together, agree to the tasks required to deliver them together, and estimate the effort involved together.

It may seem expedient to skip this level of team involvement at the beginning. It's tempting to get a subset of the team to do this (maybe just the product owner and analyst), because it's much more efficient. Somehow we've all been trained over the years that we must be 100% efficient (or more!) and having the whole team involved in these kick-off steps seems a very expensive way to do things.

It ensures the buy-in and commitment from the entire project team from the outset; something that later pays dividends. When challenges arise throughout the project, the team feels a real sense of ownership. And then it's doesn't seem so expensive.

original content from here

Agile Requirements

Agile development teams capture requirements at a high level.

Agile Development can be mistaken by some as meaning there’s no process; you just make things up as you go along – in other words, JFDI! That approach is not so much Agile but Fragile!

Although Agile Development is much more flexible than more traditional development methodologies, Agile Development does nevertheless have quite a bit of rigour and is based on the fairly structured approach of lean manufacturing as pioneered by Toyota.

We believe Agile Development teams can build better products if they have a reasonably clear idea of the overall requirements before setting out on development, so that incorrect design decisions don’t lead the team down dead ends and also so a sensible investment case can be made to get the project funded.

However any requirements captured at the outset should be captured at a high level. At this stage, requirements should be understood enough to determine the outline scope of the product and produce high level budgetary estimates and no more.

An Agile Development team (including a key user or product owner from the business) visualises requirements in whiteboarding sessions and creates requirements and storyboards (sequences of screen shots, visuals, sketches or wireframes) to show roughly how the solution will look and how the user’s interaction will flow in the solution. There is no lengthy requirements document or specification unless there is an area of complexity that really warrants it.

This is a big contrast to a common situation where the business owner sends numerous new and changed requirements by email and/or verbally, somehow expecting the new and existing features to still be delivered in the original timeframes. Traditional project teams that don’t control changes can end up with the dreaded scope creep, one of the most common reasons for software development projects to fail.

Agile teams, by contrast, accept change; in fact they expect it. But they manage change by fixing the timescales and trading-off features.

Stories can of course be backed up by documentation as appropriate, but always the principle of agile development is to document the bare minimum amount of information that will allow a feature to be developed.

Using the Scrum agile management practice, requirements (or features or stories, whatever language you prefer to use) are broken down into tasks of no more than 16 hours (i.e. 2 working days) and preferably no more than 8 hours, so progress can be measured objectively on a daily basis.

One thing I think should certainly be adopted from PRINCE2, the very non-agile project management methodology, is the idea of making sure all items are deliverables rather than activities or tasks. You can see a deliverable and “kick the tyres”, in order to judge its quality and completeness. A task you cannot.

original content from here

Active user involvement.

It’s not always possible to have users directly involved in development projects, particularly if the agile development project is to build a product where the real end users will be external customers or consumers.

In this event it is imperative to have a senior and experienced user representative involved throughout.

A User are people who will use the product.

Not convinced? Here’s 16 reasons why!

- Requirements are clearly communicated and understood (at a high level) at the outset
- Requirements are prioritised appropriately based on the needs of the user and market
- Requirements can be clarified on a daily basis with the entire project team, rather than resorting to lengthy documents that aren’t read or are misunderstood
- Emerging requirements can be factored into the development schedule as appropriate with the impact and trade-off decisions clearly understood
- The right product is delivered
- As iterations of the product are delivered, that the product meets user expectations
- The product is more intuitive and easy to use
- The user/business is seen to be interested in the development on a daily basis
- The user/business sees the commitment of the team
- Developers are accountable, sharing progress openly with the user/business every day
- There is complete transparency as there is nothing to hide
- The user/business shares responsibility for issues arising in development; it’s not a customer-supplier relationship but a joint team effort
- Timely decisions can be made, about features, priorities, issues, and when the product is ready
- Responsibility is shared; the team is responsible together for delivery of the product
- Individuals are accountable, reporting for themselves in daily updates that involve the user/business When the going gets tough, the whole team – business and technical – work together!

original content from here

Agile Testing Is Not For Dummies!

In agile development, testing is integrated throughout the lifecycle; testing the software continuously throughout its development.

Agile development does not have a separate test phase as such. Developers are much more heavily engaged in testing, writing automated repeatable tests to validate their code.

We recommend each story card to go through a validation phase though which is much more than testing, its executing the tests by separate people and make sure that the story has been developed in high enough quality.

Apart from being geared towards better quality software, this is also important to support the principle of small, iterative, incremental releases.

With automated repeatable unit tests, testing can be done as part of the build, ensuring that all features are working correctly each time the build is produced. And builds should be regular, at least daily, so integration is done as you go too.

The purpose of these principles is to keep the software in releasable condition throughout the development, so it can be shipped whenever it’s appropriate.

But testing shouldn’t only be done by developers throughout the development. There is still a very important role for professional testers, as we all know “developers can’t test for toffee!” :)

The role of a tester can change considerably in agile development, into a role more akin to quality assurance than purely testing. There are considerable advantages having testers involved from the outset.

This is compounded further by the lightweight approach to requirements in agile development, and the emphasis on conversation and collaboration to clarify requirements more than the traditional approach of specifications and documentation.

Although requirements can be clarified in some detail in agile development (as long as they are done just-in-time and not all up-front), it is quite possible for this to result in some ambiguity and/or some cases where not all team members have the same understanding of the requirements.

So what does this mean for an agile tester? A common concern from testers moving to an agile development approach – particularly from those moving from a much more formal environment – is that they don't know precisely what they're testing for. They don't have a detailed spec to test against, so how can they possibly test it?

Even in a more traditional development environment, I always argued that testers could test that software meets a spec, and yet the product could still be poor quality, maybe because the requirement was poorly specified or because it was clearly written but just not a very good idea in the first place! A spec does not necessarily make the product good!

In agile development, there's a belief that sometimes – maybe even often – these things are only really evident when the software can be seen running. By delivering small incremental releases and by measuring progress only by working software, the acid test is seeing the software and only then can you really judge for sure whether or not it's good quality.

Agile testing therefore calls for more judgement from a tester, the application of more expertise about what's good and what's not, the ability to be more flexible and having the confidence to work more from your own knowledge of what good looks like. It's certainly not just a case of following a test script, making sure the software does what it says in the spec.

And for these reasons, agile testing is not for dummies!

original content from here

No Place For Snipers!

Agile development relies on close cooperation and collaboration between all team members and stakeholders.

Agile development principles include keeping requirements and documentation lightweight, and acknowledging that change is a normal and acceptable reality in software development.

This makes close collaboration particularly important to clarify requirements just-in-time and to keep all team members (including the product owner) ‘on the same page’ throughout the development.

You certainly can’t do away with a big spec up-front *and* not have close collaboration. You need one of them that’s for sure. And for so many situations the latter can be more effective and is so much more rewarding for all involved!

In situations where there is or has been tension between the development team and business people, bringing everyone close in an agile development approach is akin to a boxer keeping close to his opponent, so he can’t throw the big punch! :-)

But unlike boxing, the project/product team is working towards a shared goal, creating better teamwork, fostering team spirit, and building stronger, more cooperative relationships.

There are many reasons to consider the adoption of agile development, and in the near future I’m going to outline “10 good reasons to go agile” and explain some of the key business benefits of an agile approach.

If business engagement is an issue for you, that’s one good reason to go agile you shouldn’t ignore.

original content from here

More Info

- Very good site which explains 7 principles of agile development

Tools

WIKI/GITBOOK

- suggest to extensively use wiki in markdown format to write
  - stories
  - issues
  - specs
- you can use the amazing gitbook tool to convert the markdown docs to an ebook (see this one)

gitbook editor

- https://www.gitbook.com/editor

how to use gitbook toolchain locally

```
npm update -g
npm install gitbook-cli -g
npm install gitbook-plugin-mermaid2
npm install phantom
gitbook update
```

to serve local content

```
cd $directory_gitbook_md_files
gitbook install
gitbook serve
```

to create a pdf

~~~~
```



- more info see https://github.com/GitbookIO/gitbook/blob/master/docs/setup.md



### plugins to gitbook
~~~~

~~~
book.json file

```json
{
    "gitbook": ">=2.0.0",
    "plugins": ["edit-link","googledocs","atoc","github","gitbookcode","chart","im
age-captions","mermaid2","highlight2","codeblock","zingchart"],
    "pluginsConfig": {
            "edit-link": {
                "base": "https://github.com/Incubaid/dev_process/edit/master/",
                "label": "Edit This Page"
            },
    "googledocs": {
            "rm": "minimal",
            "frameborder": "0",
            "width": "100%",
            "height": "500px",
            "noembed": "new window"
        },
     "atoc": {
            "addClass": true,
            "className": "atoc"
        },
    "chart": {
            "type": "highcharts"
        },
    "github": {
            "url": "https://github.com/Incubaid/dev_process"
        }
    }
}
~~~

is example file to be put in gitbook (in root of folder), enables next plugins.

table of contents

- plugin

gdocs inclusion

see link

example

```
http://spreadsheets.google.com/ccc?key={key}&hl=en
```

mermaid

SEEMS NOT TO BE WORKING ON ALL SYSTEMS (e.g. on my OSX I can't get it to work, but on gitbook site it works)

- plugin
- info about mermaid

```
\{% mermaid %\}
graph TD;
  A-->B;
  A-->C;
  B-->D;
  C-->D;
\{% endmermaid %\}
```

include codeblock

- plugin

```
\[import\](fixtures/test.js)
```

```
\[import, lang-typescript\]\(hello-world.ts\)
```

```
\[import:<start-lineNumber>-<end-lineNumber>\]\(path/to/file\)
```

dont use the '\' in examples above

include

to include other markdown docs, ideal to avoid repetition

- plugin

```
\!\INCLUDE "file.md"
```

dont use the '\' in example above

zingchart

docs see https://www.zingchart.com/docs/

```
\{% zingchart width=300, height=300 %\}
{
    "type":"bar",
    "series":[
        { "values": [35, 42, 67, 89]},
        { "values": [28, 40, 39, 36]}
    ]
}
\{% endzingchart %\}
```

FAQ

What if FR or BUG spans more than 1 repo

For BUG

- A bug can in principle only be linked to 1 repo, its always linked to the code inside 1 repo.
- In case the same bug has impact on 2 repo's, which is exceptionall then file 2 separate bugs & define what needs to be fixed in the relevant repo.
- You can link to related bugs in the other repo if this seems to be useful.
- Sometimes it can be that a bug really requires a feature request.

For FR (Feature request)

- The high level FR can be put on e.g. home repo level. This is really more like defining your roadmap.

Customers wanna report a bug, how to do this?

- If the customer (or someone from customer organization) is a contributor on the required repo, they can just file the bugs.
- It the customer is more like an end customer being less involved we suggest to use a ticketing system. At Incubaid we have created some tools to be able to use our Github also as ticketing system but otherwise there are many of them. The support person in your organization can then create the bug report in the appropriate repo and link back from the ticketing system.