"MAP - Compliant Agile Development – using iterative approaches to develop and/implement computer systems in a compliant environment"

Christopher Cieslok, intilias AG Zurich, Switzerland
Ian Francis, Business&Decision Life Sciences, Basel, Switzerland
Prof. Dr. Eckhart Hanser University of Cooperative Education, Lörrach, Germany

ABSTRACT
The first part of this paper introduces a new meta model for agile processes: MAP. Following the rules of MAP gives a better chance to have a successful software project. The most important "ingredients" of a successful software development are identified. MAP is the result of a software project over 5 years in a course of experimental software engineering at the students' lab of the Baden-Wuerttemberg Cooperative State University in Loerrach, Germany. The second part of this paper introduces MAP to the compliant environment. MAP is enhanced to MAP-CAD.

The second part is about CAD, a software development and implementation methodology which is intended to improve software quality and responsiveness to changing customer requirements based on MAP. Agile software development advocates frequent "releases" in short development cycles (timeboxing), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Other elements of CAD include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, frequent communication with the customer and among programmers and last but not least – documentation. The methodology takes its name from that idea that the beneficial elements of traditional software engineering practices are taken to agile but compliant levels.

PART I

I. AGILE META PROCESS MODEL

MAP (Meta Agile Process Model) is a representative of the class of *agile (meta) process models*. In the "Manifesto for Agile Software Development" Kent Beck, Alistair Cockburn, Ward Cunningham, Martin Fowler et.al. (Beck et al., 2001) define four principles for a successful software development:

- *Individuals and interactions* over processes and tools.
- *Working software* over comprehensive documentation.
- *Customer collaboration* over contract negotiation.
- *Responding to change* over following a plan.

Agile process models act as contrast to the so-called "heavy-weighted" process models as e.g. the "Unified Software Development Process" (UP) of Jacobson, Booch and Rumbaugh (Jacobsen et al., 1999). The aim of agile process models is a good communication with the customer to get a good software product, not (only) a good documentation.

Agile process models make sense in projects where

- the specifications of the project change rapidly,
- the customer has no final notion of the functionality of the software,
- the programmers have to be highly productive,
- the project has a high (also temporarily) risk,
- the project team is small enough(about 12 to 20 members, including managers and customer).

In opposition to agile process models as Extreme Programming (XP) (Beck, 2000), Crystal (Cockburn, 2006) or agile project management as Scrum (Schwaber, 2004), MAP does not define the process i.e. how the team has to work but defines the ingredients a project needs to be successful. MAP is the result of an "experiment". The experimental software engineering lab at the Baden-Wuerttemberg Cooperative State University in Loerrach, Germany, gives the possibility to test over several years the behaviour of the students. We can test, which agile rules and practices work and which do not.

II. STUDENTS SOFTWARE ENGINEERING LABORATORY

The students' lab takes place in the department of Applied Informatics in Loerrach, Germany. Each spring, on 11 days with 5 hours, the students of one course carry out a software project, developing collectively a software system. About 10 to 20 team members develop software for about 800 project hours per semester (totally about 3.000 hours until today). The lecturer (i.e. the author of this paper) is a member of the team, he plays the role of the customer (and sometimes the role of the coach). The students' lab has existed since 2004 (Hanser et al., 2006).

The laboratory for experimental software engineering has one main aim: To investigate the used process model. Concerning agile models, the following questions are investigated:

(1) Do classical project roles as project manager and quality manager change in an agile environment?

(2) Are there agile practices, which do not work without intervention of a project management? Which agile practices are not the results of the self organization of the team?

(3) Practical experience shows that teams with more than about 5 members subdivide into "mini teams":

- The typical XP mini team is a "pair". A pair consists of two members. Are XP pairs successful in real world?
- Is there an ideal size (perhaps more than 2) for successful mini teams?
- Which is the structure of such (successful) mini teams?

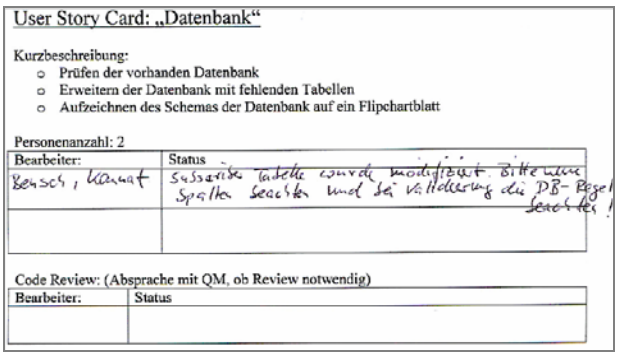
To answer these questions we will take a look at different sessions of the student' software engineering laboratory.

III. PROJECT AND QUALITYMANAGEMENT IN 2005

The "classical" project manager leads his team and supervises the project progress. He writes the project handbook and the project plan to define the project and its schedule. He provides the conditions for the success of the project. In an agile project this classical role is not clearly defined. Therefore we determined a student to take over this role and to define his new, agile role himself. As a result the 2005 project manager characterized his role as

- coordinator of the project,
- problem solver,
- communicator,
- .NET expert,
- co-designer of user stories (Fig. 1 and 2).

Fig. 1: user story card board


User Story Card: „Datenbank"

Kurzbeschreibung:
- Prüfen der vorhanden Datenbank
- Erweitern der Datenbank mit fehlenden Tabellen
- Aufzeichnen des Schemas der Datenbank auf ein Flipchartblatt

Personenanzahl: 2

Bearbeiter:	Status
Benesch, Kramat	Sussanne Tabelle sowie modifiziert Bitte neue Spalten beachten und bei Weiterung die DB-Regeln beachten!

Code Review: (Absprache mit QM, ob Review notwendig)

Bearbeiter:	Status

Fig. 2: user story card

Also the quality manager, who in his "classical" project role is responsible for writing a qualification plan, establishing a risk analysis, implementing test plans and measuring the project quality, is not a well-defined role in an agile project. Therefore, as we did with the project manager, we determined a student to take over this role and to define his new, agile role himself. As a result the 2005 quality manager characterized his role as

- someone who assures the product and project quality
  - defining project rules (Fig. 3),
  - reviews,
  - testing,
- problem solver,

- communicator,
- .NET expert,
- co-designer of user stories.

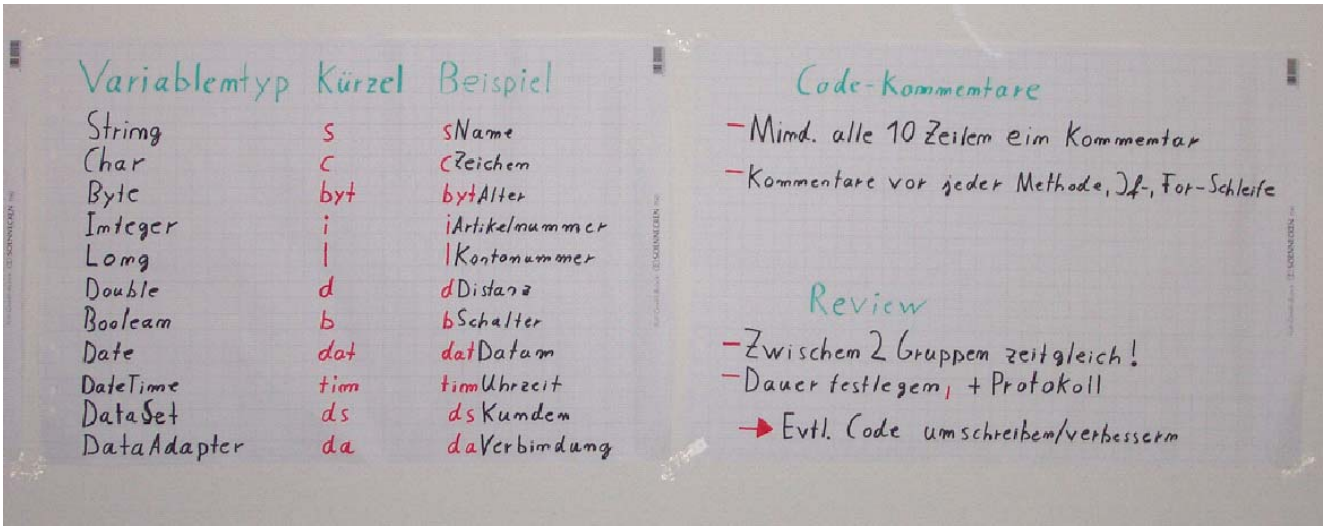


Fig. 3: project rules

It is extraordinary, that both students define their new agile roles very similarly. They differ only in the first point, i.e. in the classical part of their new agile project role. But most of the time both of them do very similar jobs: Both teach the team in (agile) project rules and practices, especially in team communication. Especially they establish a wiki web (Wiki, 2006) as a centralized, simple repository for project documentation. They supervise the team structure and are ready to change the structure of XP pairs, if necessary. Here they "fix XP" for the first time and adapt one of the XP practices: They influence the self-organization of the pairs.

IV. DESIGNING USER STORIES AND COLLECTIVE CODE OWNERSHIP

There are other XP practices which have to be "fixed": In both sessions 2004 and 2005 the designing of user stories is not very popular inside the team. The customer is hardly noticed as a full team member. Again there are the project manager and the quality manager (in their new agile roles) who fill this gap: They communicate with the customer and act as an interface between customer and team. Together with the customer and a few dedicated team members they design the user stories and try to understand the requirements of the customer in order to reach the maximum quality of the software product. Here XP is fixed for the second time. Not the whole team, but the team members mentioned above, create the user story cards (Fig. 1 and 2). Only when the developers mini teams have chosen their story cards do they begin actively to make a detailed low level design.

Also the collective code ownership is not popular in the team. Except of the project manager and the quality manager there are only the few dedicated team members having developed the user stories, who move around to understand the whole software – not only the small piece of code they work on. The readiness for collective designing or "collective code ownership" (XP) is hardly recognizable in both sessions. Therefore, in order to avoid severe failures in the software which often appear only in later project phases, the team nearly doesn't practice these both XP rules.

V. SOFTWARE INTEGRATION

The third activity, where XP does not work very well is the software integration. In session 2004 there was no integration cycle time pretended by the customer. The team was expected to "integrate often" (as required in XP). But the team members complained about weak communication in the project team. Therefore they had a lot of problems with integration in time. The first (and final) complete integration took place on the last (!) project day. These bad experiences caused the customer in session 2005 to require an official integration cycle time of 3 weeks (i.e. every third laboratory day). Unfortunately the team again made no preparations for integration in time. All programmers were busy implementing their detail user stories. Frequent (let alone continuous) integration did not take place. As a result of heavy interventions of the customer, the new project role of an "integration engineer" was created. He is the

only team member allowed to work on a dedicated integration server. He is responsible for solving all integration problems – his (only) aim is to establish a continuous integration. In session 2005 he worked alone – we called him „lone wolf“, because he always kept a distance to the other team members. As a result of his successful work continuous integration was possible within about 5 weeks (i.e. 5 laboratory days) after the project has started.

Both sessions of the students' lab show the same result: Integration is not a result of the "self-organization" of the team (as required in XP), but the result of a centralized project planning of the project management, which therefore is necessary for a successful project! Not only the integration machine has to be unique, there must be one integration engineer fully responsible for the integration of the software. Integration carried out by pairs with equal rights has not worked up to now at the students' lab in Loerrach. This is independent of the exact project team size, we see this behaviour with teams sizes of 10 members up to 20 members (which is the "natural" limit, because it is usually our maximum course size).

VI. THE SIZE OF MINI TEAMS: ARE XP PAIRS SUCCESSFUL?

An interesting parameter for a process model is the size of the subgroups of the team, where the user stories are treated (we call them “mini teams”). XP requires “pairs” (as mini teams): 2 programmers work together at a single computer. One person types and thinks tactically, while the other thinks strategically. In order to analyze the “stability” of these pairs we started with 3 members in a mini team (we called it “triplet”) in session 2004, while in session 2005 and 2008 each mini team was a “classical” XP pair. If XP pairs were an “optimum state” for a mini team, we would expect that the pairs of session 2005 and 2008 would have been stable and in 2004 the mini teams would have transformed into pairs. Therefore we focus to the change of the mini team sizes over the project time:

session 2004	**session 2005**
Mini teams with 3 members as “starting point”.	XP pairs as initial mini teams.
Result: XP pairs do **not** occur.	Result: Number of XP pairs seems to decrease.
Minimal mini team size remains **3**.	Most mini teams grow in size (3, 4 or more).
The most successful mini team has **4** members.	In special phases, e.g. **integration**, we see a team size of **6** members.

We find a few examples in figures 4 and 5, photographed after an interval of two weeks:

Fig. 4: 3 pairs and a super team in front

Fig 5: the number of pairs decreases

Therefore it seems that XP pairs are “unstable”. The size of the mini teams is oscillating, depending on the job to be done. 2008 several pairs are working so close together that they form a 4 persons mini team (Hanser, 2009).

In the author’s experience there is no convincing reason for the existence of pairs. In special project phases, e.g. preparing the integration of the software, also mini teams with 6 (or more) members are possible – we call them “super teams”. If pairs are formed, they consist often of a “stronger” and a “weaker” partner, where the first develops the architecture of the solution, while the second handles the details (without changing their roles as required e.g. in XP).

VII. MAP – Meta Agile Process Model

Organizational psychology differentiates between 6 types of members of a successful team (Hanser, 2009 and Baldegger, 2001):

- communicators,
- creative minds,
- technical experts,
- problem solvers/trouble shooters,
- team workers
- and quality testers.

The mixture of these different types is the basis for the success of the team. “Classical” XP pairs, on the other hand, require similar types, because the partners should alternate in strategical and tactical thinking. Especially with male programmers, this may generate rivalry between the pair members, which increases the risk of failure of the pair. In the students’ lab well-working pairs with members of equal rights could be found mostly between female students!

Mini teams which consist of the above 6 types and which are not too big, are considered as successful in the organizational psychology. The results of the students’ lab, especially concerning the “super team” mentioned above, confirm this statement in the field of software engineering. Especially the successful bundling of all detail activities at the beginning of the first integration in session 2005 was the precondition for the success of this session. Nevertheless the existence of an integration engineer, the “lone wolf” is absolutely necessary. This job can only be done centralized – by one person.

In the students’ lab we had to change four “classical” agile rules and practices to get a successful software project in session 2005. In the authors opinion the process model which is actually used is no longer XP (but nevertheless agile!).

It seems that the approach of process models is proven false although it is very important for software developers to know about process models. Process models are a valuable *toolbox* for development teams. But for “real world project teams” it is more important, to identify the most important prerequisites of a successful software development in a team than to follow a certain process model. As a result of our students’ lab a successful software development project in a small to medium sized team (with up to perhaps 20 members) needs the following “ingredients”:

- A *customer*, who is able to elaborate his project requirements, and who is available to the team if he is needed.
- One or two good and flexible *communication managers*, who share project and quality management tasks, but especially ensure a good communication in the team and with the customer. They must have a good feeling for the project and its psychology and a thorough understanding of process models in order to have the courage to fix the used process model, if it is necessary. (We fixed XP four times!)
- An *integration engineer* who demands and guarantees a continuous software integration. He is fully responsible but supervised by the project management.
- A well balanced project team with a good psychological process (and therefore a good communication), where all 4 types of project members are represented which we know from organizational psychology.

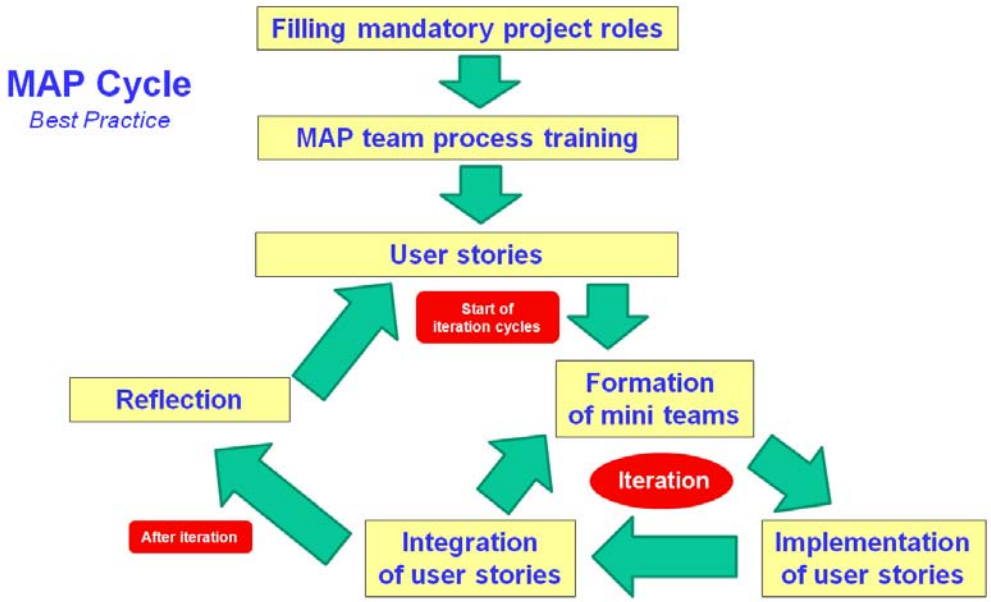


Fig 6: reference process MAP Cycle

Regarding our experiences in the students' lab we can deviate from MAP a reference process model which fulfils the demands of MAP: *MAP Cycle* (Fig. 6). It is important to understand that MAP Cycle is only a *reference* process! In the last 5 years the author could see that the students' teams followed more or less this process regardless of the process they chose in the beginning of their laboratory session. But in the end the experiences of the students' lab of the Baden-Wuerttemberg Cooperative State University in Loerrach show that if the ingredients of MAP are fulfilled, the chosen process model is of secondary importance.

VIII. REFERENCES

BALDEGGER, R., GOTSMANN, L., 2001. *Ganzheitliches Projektmanagement.* Baldegger Verlag.
BECK, K., 2000. *Extreme Programming Explained.* Addison-Wesley.
BECK, K., et.al., 2001. *http://www.agilemanifesto.org*
COCKBURN, A., 2006. http://alistair.cockburn.us/
HANSER E., 2009. *Agile Prozesse – von XP bis MAP.* Springer *(to be published)*.
HANSER, E., BRECHT, U., MICHELBACH, A., 2006. *Agile Software-Entwicklung: Extreme Programming im IT-Labor der BA Loerrach.* BA Dialog, BA Loerrach
JACOBSON, I., BOOCH, G., RUMBAUGH, J., 1999. *Unified Software Development Process.* Addison-Wesley.
SCHWABER, K., 2004. *Agile Project Management with Scrum.* Microsoft Press.
WIKI, 2006. http://en.wikipedia.org/wiki/WIKI

PART II

IX. INTRODUCTION

A common problem in system development projects that is often encountered with traditional waterfall or "V" model approaches is that we want to implement a system that meets the needs of the user BUT the user often cannot express their needs clearly and/or their needs may be dynamic and will often change based on opportunities available in the solution...

"The system can do THAT! Can we add that to our requirements....?"

“We changed our business process, can the project change direction…?”

A common problem in system development projects that is often encountered with traditional waterfall or "V" model approaches is that we want to implement a system that meets the needs of the user BUT the user often cannot express their needs clearly and/or their needs may be dynamic and will often change based on opportunities available in the solution...

"The system can do THAT! Can we add that to our requirements....?"

“We changed our business process, can the project change direction…?”

This causes familiar problems in predictive (waterfall) approaches that are poorly managed, such as requirements being defined in isolation from the system and then being locked from change early in the life cycle, delivering a solution that does not meet user’s real needs, and causing dissatisfaction in the users, and that does not provide a return on investment, etc....

Using an Agile-like approach to develop systems can mitigate against these problems and help to ensure user satisfaction and overall quality.. But there are a few things we need to consider when using such an approach in a regulated environment;

- **Scope Creep:** It is true that the scope of a system development project must be clearly defined, for cost control and time/resource management, but too often the scope is locked early and at a detailed level. We should identify a difference between "scope creep" and "functional evolution". The scope of the system should be defined at a relatively high level in a Business Solution Concept. But detailed specifications should be open to evolution and (managed) change.

  For discrete projects, we should control the overall scope of the system based on the concept, and apply an iterative approach to evolve the system functions, specifications and business procedures to a point where they provide real value to the user.
  This is less of a concern for operational enhancements or continuous improvements, because these are normally relatively small changes with a well defined scope.

- **Software Control:** Key to any well managed development is maintaining control of configuration items. Agile type developments are no different; Good Engineering Practices (GEP) and appropriate tool sets should be used to ensure that efficient configuration, change and release management processes are applied.

- **Documentation:** Beyond GEP, in the Life Sciences industries we need a certain level and quality of documentation that may not be required in other industries; these are derived from many sources of Good Practice (GxP).
  We need to have the process for providing quality documentation **built in** to our approach, combining and scaling documents where appropriate.

- **Technical Foundation:** Any software development has a set of prerequisite foundations; these can include programming languages, compilers, operating systems, hardware, networks, and databases. Normally such technology will be standard components of a solution and these should be appropriately qualified.

  Additionally, software development (i.e. creating custom code) is today normally performed within the framework of a larger application (for example SAP ABAP coding or SAS programs), and as such these applications are also a required foundation before beginning an Agile development.

X. CONCEPT PHASE

ACTIVITY. BUSINESS SOLUTION CONCEPT

This could be a description fewer than 10 pages (preferably 3-5 pages) of the key business concepts that the users have for the system, e.g.

> *Solution must provide an environment to allow for Clinical Data storage, processing, analysis and manipulation*
>
> - *With graphical user interfaces to allow for intuitive use by the end user*

Should also describe regulatory considerations for any solution, e.g.

> *The solution must provide for an interface to regulatory authority systems to allow for online submission of trial data*

Used to define system scope and as an input to package selection/supplier assessment activities. Document serves to open the dialogue between business users and system developers and also to provide the foundation of deploying a framework system to begin the Development activities.

OUTPUT. PROJECT RISK ASSESSMENT

As part of an overall risk management process, an assessment of the Business Concepts to determine the suitability of an agile or plan driven model could be applied, as in this example model:

XI. AGILE COMPATIBILITY ASSESSMENT

We recognize that an Agile approach may not be acceptable or appropriate to every software development; we need to be flexible and apply each methodology based on fitness-for-purpose.

The assessment may be performed at the appropriate level;

- the entire solution,
- individual work streams,
- functional areas etc…

This can provide a rationale for applying different development models to different solution areas.

Output. Technical Solution Concept

This could be a description fewer than 10 pages (preferably 3-5 pages) of architectural constraints or concepts that are applicable, e.g.

- *Server architecture must be based on AIX; client (end user) architecture must be based on commercially supported Linux*
- *The primary application framework will be SAS (latest version)*

This would be a collaborative effort between business users, IT (who could be external or internal) and supplier (who could be IT)

Output. Framework (Vanilla) System

At the end of the concept phase, a basic system should be built (e.g. servers, middleware, databases, COTS apps, possibly minimally configured)
This will be built based on the Technical Solution Concept, which will also be updated if any changes are needed during the build.
This system will be a "controlled development" environment, with freedom to prototype but still with GEP software controls in place. Development items will be "promoted" from this environment into a formal test environment when it is decided the development cycle for the item has finished.

Output. Configuration/Build Specification (Draft)
The activities required to create the framework system must be documented (compliance requirement). This deliverable is started now and will be added to continually for each CI **tested** and **approved**; this also allows this work item to serve ultimately as a functional test summary report, as it will only describe CIs that have been tested and approved. First formal approval (i.e. configuration baseline) will not occur until production deployment readiness review (or similar).

Output. Configuration Management Database (CMDB)
The CMDB will record all CIs (i.e. anything that is a design solution such as configuration settings, code units, SOPs), their status and relation to user stories and other CIs.
This work item will need to serve the role of Traceability Matrix and Design Specification, as well as the traditional CMDB and inventory/asset management purpose.

Output. Quality Assurance and Control Procedure
This will be defined as part of an overall Quality Management System and is analogous to a Quality Plan. It will define the governance processes and enabling tools that will be applied to control the development and associated artifacts.

XII. AGILE DEVELOPMENT PHASE

Agile techniques should be applied appropriately throughout this phase, including team management and project planning principles.

Inputs. Business and Technical Solution Concepts

The individual statements in the Solution Concepts are "deconstructed" into developmental areas. Developmental Areas may be based on roles, functional areas or work streams.

Activity. Create User Stories

Initial "user stories" are created, written such that acceptance test can be directly defined against them; preferably with acceptance criteria being defined at the same time. The team should strive to ensure that the user stories are deconstructed to the smallest meaningful size. This is useful to aid in the commitment phase (i.e. planning).

Output. User Story

User stories should be[1]:	User Stories should contain:
* I - Independent * N - Negotiable * V - Valuable * E - Estimable * S - Small * T - Testable	ID Number... Version/Iteration.... Group… User Story... Development Notes… Acceptance criteria... Approval for Formal Verification... Approval for Production Deployment...

In general, user stories should be short; limited to a single sentence with the structure[2]:

This ensures that developers understand who the functionality is for and why they want it. As noted above, it is suggested that the initial acceptance criteria are stated before the solution is developed. This incorporates the concept and benefits of Test Driven Development. Of course the acceptance criteria may also evolve as a user story evolves.

User Story Examples

- As a customer representative, I can search for my customers by their first and last name.
- As a non-administrative user, I can modify my own schedules but not the schedules of other users.
- Starting Application: The application begins by bringing up the last document the user was working with.

[1] Credit: Bill Wake
[2] Credit: Mike Cohn

- Closing Application: Upon closing the form, the user is prompted to save/cancel (when ANYTHING has changed in the form data since the last save).

A customer will often provide a number of user stories wrapped up in a single statement, for example;

The employee will enter expenses on an expense form. The employee will enter items on the form like expense type, description, amount, and any comments regarding the expense. At any time the employee can do any of the below options.
(1) Once this is completed the employee will "Submit. If the expense is under fifty (<50), the expense will go directly to the system for processes
(2) In the event the employee has not finished entering the expense, the employee may want to "Save for later". This instance should then be displayed on a list (queue) for employee with the status of "Incomplete"
(3) In the event the employee decides to clear the data and close the form the employee will "Cancel and exit". This is instance will not be saved anywhere.

This should be broken into smaller, yet still meaningful, user stories, for example:

- The employee will enter expenses on an expense form. The employee will enter data on the form: expense type, description, amount, free-text comments regarding the expense.
- Once the expense type, description, amount are completed the employee will "Submit". If the expense is under fifty (<50), the expense will go directly to the system for processes, if the expense is over fifty (>50), the expense will go to account manager for approval;
- At any time, in the event the employee has not finished entering the expense data, the employee may want to "Save for later". This expense record should then be displayed on a list (queue) for the employee with the status of "Incomplete";
- In the event the employee decides to clear the data and close the form the employee will "Cancel and exit". This expense record will be discarded and not be saved anywhere.

Output. Story Book
An index of all user stories that is used to manage progress and priority/risk. The initial user stories will typically spawn other more detailed user stories during development, so the Story Book will also grow and evolve.

Activity: Commitment phase
This phase involves the determination of **costs**, **benefits**, and **schedule** impact of the development activities. The process can be performed for a group of stories that will be developed in parallel (e.g. there will be a one month time box to develop the stories decided during the commitment phase) or the process can be performed on a per story basis (e.g. where there are rolling time boxes each with a separate story to develop).

The objective for the end of each time box is to deliver a solution suitable for release to the formal verification environment.

Comparison of Approaches

Time Box Commitment	User Story Commitment
Should result in delivery of a group of functionality.	Should deliver small incremental changes.
Useful for related or interdependent stories.	Useful for delivery of discrete functionality
Useful for completing larger pieces of work e.g. complete functional or business areas.	Useful for continuous improvement or addition of functionality in a system that is largely complete (e.g. operational enhancements)
Length of time box will be dependent on the number and types of user story selected for development and budget allocation.	Length of time box can be uniform (e.g. "not more than 1 month")

Assessment Activities

1. Sort Developments by Value

The business side sorts the user stories by business value, assigning one of three categories:

- Critical: stories that are a regulatory requirement or without which the system or business process cannot function or has no meaning.
- Significant Business Value: Non-critical user stories that have significant or immediate business value.
- Needed: User stories that will provide specific value and are required, but there is no immediate driver to develop and that do not have immediate business value when there are other more important stories.
- Nice to have: User stories that are not important but may provide value to individuals (e.g. changing font styles on reports, or moving the placement of buttons on a dialog).

2. Sort Developments by risk

The developers sort the user stories by risk. The following is an example of an approach to this:

Determine Risk Index: Assign each user story a score for each of the following factors:

Completeness	**Volatility**	**Complexity**
Do we know all of the story details? Can the story further decompose? • Complete (0) • Incomplete (1) • Uncertain (2)	Is the story likely to change? • Locked (0) • Stable (1) • Changeable (2) • Volatile (3)	How hard is it to build? • Simple (0) • Standard (1) • Uncertain (2) • Complex (3)

All scores for a user story are added, assigning the user stories a risk index of negligible (0-1) low (2–3), increased (4–5), or high (6–8).

(Discussion Point: This could also be seen as synonymous with a value for "story points".)

3. Estimate Release scope

Based on the risk and value, an initial estimate of relative confidence that the user story (development) should be included in the next time-box/release can be estimated.

Confidence matrix:.

Risk \ **Business Value**	Nice to have	Needed	Significant or Immediate	Critical
None (0)	75%	75%	100%	100%
Negligible (1)	50%	75%	100%	100%
Low (2–3)	25%	50%	75%	100%
Increased (4-5)	0%	25%	50%	75%
High (6–8)	0%	0%	25%	50%

4. Refine Development scope:

Developers estimate the reality of developing a working solution for the user story with current resource/constraints. They may create proof of concept solutions to better understand this.

(Discussion Point: Developers may use the Planning Poker game; this is a useful way of mitigating against an over-optimistic leader who wants to push certain items through even when they may be improbable to achieve given the current resource/constraints.)

The output should be an estimate of:

- Certainly or easily achievable in time box
- Probably achievable in time box
- Not probable in time box

If a realistic estimate cannot be determined, it is often due to the user story being too complex; the business should review the user story to attempt to deconstruct it into simpler units.

This activity will provide a refined list of user stories indicating candidates for inclusion in the next work period (time box)

Reality / Confidence	Certain / Easy	Probable	Not probable
100%	Develop	Develop	Review resource, constraints
75%	Develop	Review confidence, resource, constraints	Review resource, constraints
50%	Review confidence, reality	Review confidence, resource, constraints	Postpone
25%	Review confidence, reality	Postpone	Postpone
0%	Postpone	Postpone	Postpone

Refinement can be performed for all current user stories (more appropriate for rolling release approach), or performed only for stories >X% from "Step 3. Estimate Release Scope" (more appropriate for sequential release approach where the reality of being able to implement a group of stories can be assessed).

As with any subjective risk assessment process, there will be exceptions to any fixed rule, and as such the final list of user stories may be modified based on justifiable decisions. The development velocity can be a factor, as can other external influences, such as regulatory drivers, known impediments to progress or the budget that can be assigned to the next iteration/time box.

Activity. Agile Development
This is where the work gets done on the user stories selected during the Commitment activity.

During this activity, it is expected that user stories will evolve. An example of this is shown below:

Original User Story	**Evolved User Story**
As a non-administrative user, I can modify my own schedules but not the schedules of other users.	As a non-administrative user in the IT department, I can modify my own schedules and those of my department but not the schedules of users outside my department.
Starting Application: The application begins by bringing up the last document the user was working with.	Starting Application: The application begins by asking the user if they want to start with the last document the user was working with. "Yes" will present the last document the user worked on; "No" presents a blank document.

Note that this is not the same as scope creep; it is an evolution of the existing story within the high-level defined scope.

Evolving Stories, Scope and Cost
By the end of the iteration, a working solution should have been developed, and this will typically have caused to user story to evolve.

There may be two main reasons for this:

1) The user story has evolved and has grown (possibly with conditional scenarios within the same requirements) and it is logical to break off these into discrete user stories. The overall scope has not changed. These extra stories should be added to the Story Book and scheduled accordingly.
2) A completely new requirement has been identified. This will generally mean a change in scope. The new requirement must be examined in detail to determine (a) is it really required and (b) is it already covered by existing entries in the Story Book.

So in some cases this evolution may have identified new user stories, meaning the end result will be a working

solution to meet one user story and one or more new user stories. These will not have been previously planned or budgeted for, leading to the concern of scope (and cost) creep.

As the requirements change the cost must also change, i.e. it isn't clear how much the system will cost up front. However, the initial requirements envisioning (Business Concept, Technical Concept, initial list of User Stories) should aim to provide sufficient information about the project scope to give a reasonable, ranged estimate early in the project.

Also, the customer has control over the budget, scope, and schedule and gets concrete feedback on a regular basis – throughout the development process. In this situation the customer doesn't need an detailed estimate up front because of the increased level of control which they have. Would the customer rather have a detailed, and probably wrong, estimate or requirements and cost up front or would they rather be in control and spend the money wisely?

Output. Updated CMDB
The CMDB must be continuously updated with configuration items created or changed to implement the user story functionality to maintain a list of CIs, their relationships and their status.

Output. Approved (Locked-down) functionality
User stories are developed in the system into functioning software through a number of iterations until there is an agreement that the user story reflects the real needs of the user and the solution appears to provide that need; at this point the user story and related CIs are formally "approved". The CIs may now be released to the formal Test environment as required for formal acceptance. Release planning should be applied as it may be appropriate to transfer a group of functionality representing a larger unit (e.g. business process); the approach taken during the commitment phase will also influence how functionality is released for formal acceptance.

XIII. FORMAL VERIFICATION

As described in the Agile Development Phase, the User Stories and acceptance criteria will have been defined before the solution was developed and then evolved during the iteration.

At the end of the iteration, the user story should be complete (based on the user needs) and the solution should meet the story. At this point an approval to release to the formal verification environment should be made. This is to meet regulatory expectation that requirement/specification/test are Approved.

This is not an approval to test. The scope of testing should be based on risk assessment as described in the appropriate Risk and Verification Management procedures. The risk assessment performed in the Commitment Phase can be re-used to determine the scope of testing, e.g. all Stories with a Confidence Value above 40% will be tested.

The Confidence Value is useful as a measure of overall risk since it combines criticality to the business, risk of the solution being buggy (Complexity) and the risk that the solution will meet requirements (Volatility and Completeness). However, as with any subjective risk-based process, there may be exceptions to any fixed rule and these should be justified.

Note that lower levels of verification (white box, black box testing) are normally built into an Agile process. This must be described in the Quality Assurance and Control Plan (or appropriate procedure).

XIV. MANAGEMENT TOOLS

To successfully manage a large or active agile development lifecycle, appropriate tools must be used – it is not feasible to apply paper-based techniques to managing agile projects. (Note that XP and many agile techniques recommend using index cards for recording and tracking user stories. This may be appropriate in the early stages of defining a Story Book, but should be transferred to an electronic tool as soon as possible; this will make sharing progress and managing CIs more efficient).

The activities that are critical to having the correct technical tools in place are discussed below:

SCM: Software Configuration Management

CT: Collaboration Tools; useful for rapidly sharing and developing user stories and disseminating project progress, i.e. publishing the current status of the Story Book.
RTM: (Requirements) Traceability Management
DM: Document Management
TM: Test Management
IM: Incident/Issue/Problem Management
CM: Change Management
PM: Project Management
RM: Release Management

Example solutions

Name	SCM	CT	RTM	DM	TM	IM	CM	PM	RM
Agilo								□	
Bugzilla						□			
ClearCase	□								~
Confluence (wiki)		□							
IBM Doors		~	□						
HP Quality Center (Mercury Test Director)			□		□	□			
Jira						□		□	
ManageEngine Service Desk	~					□	□		
IBM RequisitePro			□						
ScrumWorks						~		□	□
Sharepoint		□							
Subversion	□								~
Visual SourceSafe	□								~

□ Key Feature
~ Secondary Feature
Free version

Scrum vs. CAD (extract)

CAD	Scrum	Key differences
Story Book	Product Backlog	Risk included in CAD
Time Box	Sprint	Time boxes may overlap; Sprints a sequential
Group (user story)	Theme	

XV. CONCLUSION

With Compliant Agile Development it is possible to use best practices for software development like extreme programming, rapid application development and/or other iterative approaches to implement any software in the heavily regulated pharmaceutical industry. This includes software and application development but is not limit to it as it also can cope with standard software application or custom build software packages.

CONTACT INFORMATION

Christopher Cieslok
Christopher@cieslok.net