Zusammenfassung SWE2

René Bernhardsgrütter, 25.12.2013

Agile Manifesto

Individuals and interactions over processes and tools
... because **great software** is made by great individuals
Working software over comprehensive documentation
... because that makes it possible to get **feedback early**
Customer collaboration over contract negotiation
...because agile teams would like all parties to the project to be working toward the same set of goals
Responding to change over following a plan
... because their ultimate focus is on **delivering as much value as possible** to the project's customer

12 Principles (shortened)

- Highest priority: **customer satisfaction** through early and **continuous delivery** of valuable software.
- **Welcome changing requirements**.
- Deliver **working software frequently**.
- **Business** and **devs** must **work together daily**.
- Motivated individuals. Give environment, support and **trust**.
- Best communication way is **face-to-face**.
- **Working software** is the **measure of progress**.
- Agile processes promote **sustainable development**. Sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- **Continuous attention to technical excellence** and good design enhances agility.
- **Simplicity**: art of max. amount of work not done is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**.
- At **regular** intervals, **the team reflects** on how to become more effective, then tunes and adjusts its behavior accordingly

Developing Software

Variable eines Projekts	**traditionell**	**agil**
Zeit	fix	fix
Ressourcen	fix	fix
Qualität	variabel	fix, am wichtigsten
Funktionalität	fix	variabel

XP Practices

Planning Game: Missverständnisse klären und gut schätzen: Manger schätzt Aufwand. Die Entwickler machen selbiges unabhängig davon nochmals. Wenn die Schätzungen auseinander gehen, ist etwas unklar und wir ausdiskutiert.
Small releases: Pro Funktionalität ein Produkt-Release (nightly).
Metaphor: Gemeinsame Domänen-Sprache unter allen Beteiligten.
Simple design: Erst machen, wenn man es braucht → Emerging, growing desing!
Testing: Automatisiert, schnell, absolut notwendig.
Refactoring: Setzt Tests voraus. Auch zwingend.
Pair programming: 1 schreibt, 1 sagt was und reviewed gleich- zeitig (auf selbem Screen). Oft wechseln = Knowledge-Transfer.
Collective ownership: Jeder darf/muss Wert hinzufügen. Das ist gut, denn jeder (ausser der beste) enthält Feedback zum Code und niemand verteidigt *seinen* Code.
Continuous integration: Nightly Builds. Kann direkt in Produktion gehen, wenn das der Kunde will.
40 hour week: Fresh every morning, satisfied every evening.
On-site customer: Je näher, desto besser. Sollte zumindest gut erreichbar sein (Telefon over E-Mail).
Coding standards: Im Team gleich, da collective ownership.

Pyramid of Agile Competence

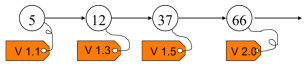
1. **Agile Values**: Selbstgebildetes Team, 40h, gutes Verhältnis zu Kunden
2. **Management Parctices**: SCRUM
3. **Engineering Practices**: XP, CL, TDD, Clean Code, Build automation, Design, Refactoring

Versionskontrolle

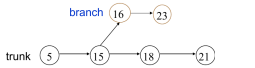
Kontrolliert Änderungen, erlaubt Wiederherstellung eines alten Zustandes, mehrere Versionen parallel zu haben und kann Dateien locken und/oder mergen. Generierbares sollte nicht ins Repo! Nur wenige Branches machen.

Nomenklatur

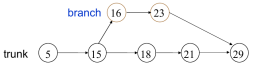
Version	Release mit *n* Funktionen
Release	Etwas, das beim Kunden produktiv läuft
Revision	Bugfix (in der Regel)
Variants	Verschiedene Ausführungen, z. B. plattformabhängig
trunk	Hauptentwicklungsstrang in svn
branch	Nebenentw.zweig für grössere Experimente o. andre Versionen
tags	Markieren einen Stand des Repos als relevant (z. B. „V1.0“)



Releases werden **getagged** (dies nur im trunk machen!)



Branches für Bugfixes oder neue Features.



Zwei Versionen **mergen**. Sollte vermieden werden, wenns geht.

Committing

Oft, mindestens am Abend! Formatierungs-Änderungen unabhängig von Code-Änderungen committen, da es sonst so aussieht, als ob man alles geändert hätte. Beschreiben, ***warum*** man etwas geändert hat, **nicht *was*** (zeigt *diff*). Nur funktionierenden Code committen! SVN = *atomic* Commits.

Continuous Integration

Integration is "making different modules work together".
Wenn verschiedene Personen/Teams miteinander arbeiten, müssen die verschidenen Module ***integriet*** werden, also *Compile*, *Tests*, *Laufenlassen*, *Deployment*.

Broken Integration:

- Integration Server nicht buildet
- Shared Components nicht in allen System korrekt laufen
- Unit Tests, Code-Quality oder Deployment failt

=> Je früher gefixt, desto weniger kostet es!
Manuelle Integration ist schlecht, denn:

- Teuer, da zeitintensiv
- Wenig ausgeführt, denn teuer
- Fehler werden später entdeckt, denn man integriert seltener
- Hemmt Refactoring, denn jedes Mal wäre es teuer, wenn etwas kapput ginge => weniger refactoren => Qualität sinkt => Projekt stribt

Voraussetzungen: VCS, Build/Deployment Server (Jenkins), Ant.
Vorteile:

- Reduziert Risiken!!!
- Aktueller Projektstatus immer für alle klar
- Bugs früher und schneller gefunden
- Beweis, dass System läuft
- Code-Qualität kann erhöht werden, wenn man oft refactored oder Checkstyle nutzt
- Erzeugt Statistiken über Code-Coverage der Unittests, wie oft etwas schief geht, wie oft gebuildet, etc.

Probleme:

- Umständlich, existierende Systeme in CI-Umgebung zu bringen
- Abhängige Systeme machen (DB, Sharepoint, etc.)
- Datenbank muss aktuell bleiben (Schemen, Benutzer, etc.)

Bei Datenbanken vorher Backups erstellen und diese testen!

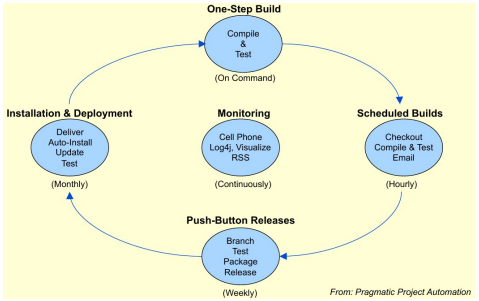
Build automation

Typischerweise ist "Building" oder "integration" immer aufwändiger als make/configure. Z. B: cleanup, checkout von aktuellem Stand, testen, buidlen, deployen, loggen/informieren, Statistiken erfassen.
Ziel: Das alles soll **jederzeit auf Knopfdruck** gehen. Jeder Build-Prozess max. 10 Minuten dauern. Nach jedem Commit builden. Bei Problem => Alarm! Fixen!
Typische Probleme bei Building/Integration:

- Lokal bei verschiedenen Devs anders gebuildet, verschiedene Environments und die können ändern
- Wie Versionierung bei 10 Entwicklern?
- Inkonsistentes/unvollständiges Unittesting
- Intransparentes Deployment wenn nicht nach Schema-F

Wieso CI?

- Manuell ist Fehleranfällig und repetitiv
- Automatisiert = Dokumentation (Code is Doc)
- Vertrauen in das System



CRISP Builds

Complete:	recipe lists all ingredients
Repeatable:	version control time machine
Informative:	radiate valuable information
Schedulable:	complete and repeatable
Portable:	machine-indementdent

Ant

Automatisiert Buildprozess. XML-Config, sehr gut erweiterbar, für verschiedene Sachen nutzbar (auch C usw). Man könnte z. B. VMs starten oder auf verschiedenen Build-Servern für verschidene Achritekturen Builds mit einem Knopfdruck triggern.

`<project name="Test" default="init" basedir=".">`
Beschreibt das Projekt. 1x pro XML-File. Das `basedir` dient als realtives root-dir für weitere Parameter.
`<target name="trgt"><echo message="Hi!"/></>`
Projekt hat Targets, die nacheinander ausgeführt werden. Jedes besteht aus mehreren Tasks (hier: echo).
`<property name="sd" value="src" />=> ${sourceDir}`
Properties sind Variablen. Können in .properties-File sein.

UnitTesting

It's to proof, that your software does what it's expected to, correctly!

Validation: Did you build the right thing?
Verification: Did you build it right?
Fault: static defect
Error: incorrect internal state, caused by a fault
Failure: External incorrect/unexpected behavior

> Testing can only show the presence of a defect but not the absence of defects!

Unit Testing is done on each unit (class,method), in isolation to verify the unit's behavior.
Unit test will establish an artificial environment (called test harness), invoke routines in the unit under test and check the results against expected values.

JUnitTests:
- `assertEquals`: bei Primitive, ob gleiche, sonst `.equals()`
- `assertSame`: vergleicht, ob gleiche Referenzen (==)

Annotationen:
- @Before-/AfterClass: müssen **public static void** sein
- Alle Methoden mit @Before-/AfterClass, @Before/After, @Test müssen **public void** sein und **keine** Parameter nehmen.
- @Ignore werden ignoriert

Null Object Pattern: Anstatt oft auf `null` prüfen: A Null Object is an object with defined neutral ("null") behavior. The Null Object design pattern describes the uses of such objects and their behavior (or lack thereof).

A-TRIP (what good tests are)

Automatic	invoking the tests and checking the results
Thorough	tests all that's likely to braek => code coverarge
Repeatable	run over and over again, producing the same results
Independent	no test relies on another test
Professional	use professional standards as for the production code

Äquivalenzklassen (ÄK)

= Partition the input domain into regions, where you expect the same behavior.
Man will die Anzahl Fälle minimieren, indem man solche ÄK macht, die einen Bereich auf gleiches Verhalten testen. Pro ÄK soll am Schluss ein Test reichen.

$\sqrt{((x-1)*(x+2))}$ x<=2 und x>=1 valid, -2<x<1 nicht.

Nun die ÄK-Grenzen bestimmen und *um diese herum* testen. Zudem auch Extremwerte testen.

ÄK-Anforderungen:
- 1 zusammenhängender Wertebereich = 1 gültige und 2 ungültige ÄQ (Randwert 1, gültige Werte, Randwert 2)
- Wenn Elemente einer ÄQ verschieden behandelt werden, sind es unterschiedliche ÄQ
- Testdaten sollen genau einen gültigen Wert haben (nicht eine Menge von gültigen Ergebnissen) und immer selbes ergeben

RIGHT-BICEP

Right	Are the **results right**?
B	**Boundary** conditions correct? Most of the bugs generally live at the boundaries
I	Can **inverse relationships** be checked? sqrt(4) == 2*2 ?
C	Can results be **corss-checked** using other means? Z. B. eigene sqrt() mit Math.sqrt() vergleichen
E	Can **error conditions/exceptions** be forced to happen?
P	Are tests **performant**? Nicht proaktiv!

TestDoubles in UnitTesing

Dummy	objects are passed around but never actually used. Usually they are just used to fill parameter lists.
Stubs	are minimal implementations of interfaces or base classes. Methods returning `void` will typically contain no implementation at all, while methods returning values will typically return hard-coded values.
Spys	similar to a stub, but a spy will also record which members were invoked so that unit tests can verify that members were invoked as expected.
Fakes	contain more complex implementations, typically handling interactions between different members of the type it's inheriting.
Mocks	objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Wann mocken?

Wenn das echte Objekt kompliziert zu erzuegen, langsam, ein GUI ist oder noch nicht existiert. Immer dann, wenn viel Aufwand.

Wie mocken?

```
Song mock = createMock(Song.class)
expect(mock.getTitle()).andReturn("My Title");
replay(mock); ...; verify(mock);
```

.createNiceMock vs .createMock: Bei createNiceMock werden bei nicht definierten Methodenaufrufen `false` bzw. `null` zurückgegeben. Bei .createMock gibt es eine Exception.

Refactoring

Refactoring "is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet ***improves*** its internal structure."

Benefists

- Improves the design of the system
- makes software easier to understand (see CleanCode)
- helps to find bugs because one can/has to understand the code
- saves development time, because good code evolves better and code/design is more robust

Voraussetzungen

Automatisiert Testen, CI machen und Coding Standards prüfen.
Im Team Collective Code Ownership, Pair Programming, Simple Design mit Rested Programmers machen.

Entweder/Oder

Bei grösseren Änderungen entweder entwicklen ODER refactoren. Kleines natürlich on-the-fly machen.

> Don't try to clean the code when wearing the function hat.
> Don't try to add features when wearing the refactoring hat.

Wann?

Bevor neue Funktionalität hinzugefügt wird, wenn man etwas neues über den Code lernt, als Konsequenz eines Bug-Fixes oder eines Code-Reviews. Allerdings **NICHT, wenn die Tests nicht passen** oder man Teile von dem Code reimplementieren sollte.

Probleme/Umstände

- Refactoring nicht perfektionieren!
- Datenbanken sind langlebig, daher dort vorsichtig sein.
- Veröffentlichte Interfaces möglichst kompatibel halten.

Code Smell

A Code Smell is a ***hint*** that something ***might be*** wrong w/t code.

Ist auf Level *Code*, *Klassendesign* oder *Architektur*.
Manuell (durchsehen) oder mit Checkstyle finden (prüft, ob Code dem definierten Style entspricht).

Typische Code Smells

"Too Much" Code Smells:
- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Feature Envy
- Switch Statements
- Parallel Inheritance Hierarchies

Code Change Smells:
- Divergent Change
- Shotgun Surgery

"Not enough" Code Smells:
- Empty Catch clause

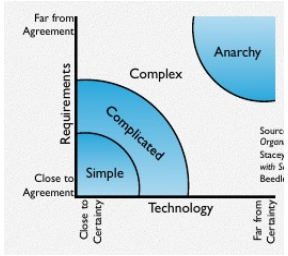
Comment Smells:
- Need To Comment

Scrum

- Scrum is an agile process that allows us to focus on **delivering the highest business value** in the shortest time.
- It allows us to rapidly and repeatedly inspect **actual working software** (every two weeks to one month).
- The business sets the priorities. Teams **self-organize** to determine the best way to deliver the **highest priority features**.
- Every two weeks to a month anyone can see **real working software** and decide to release it as is or continue to enhance it for another sprint.

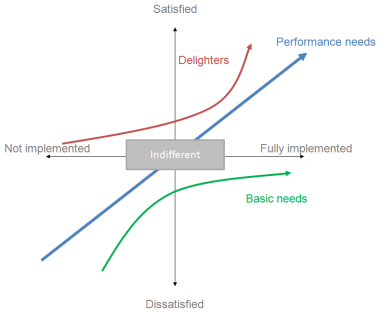
Project Noise

Sagt aus, welcher Entwicklungsprozess genutzt werden sollte.
Simpel geht mit Wasserfall, es gibt aber fast keine solche Projekte.. *Complicated* sollte mit SCRUM gemacht werden können. *Anachy* niemals machen, scheitert eh.

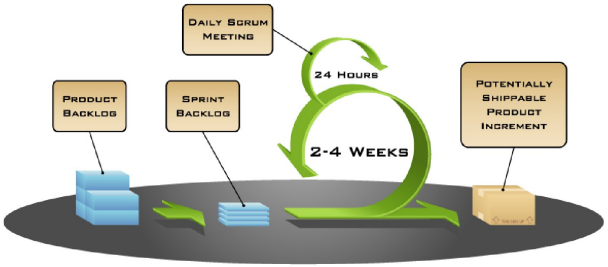


Kano-Modell

Modell zu Analyse von Kundenwünschen/-zufriedenheit. Sollte linear wachsen, die *Basic needs* sollten voll erfüllt sein.



Workflow

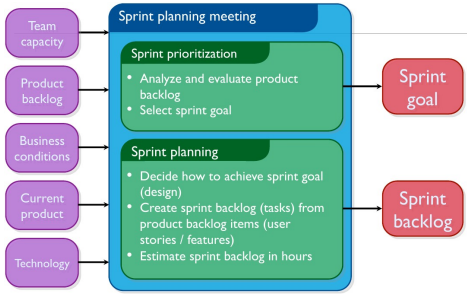


Product Backlog: Sind die priorisierten Requirements, die in einem Projekt erfüllt werden sollen. Product Owner repriorisiert vor Sprints. Bsp: “Allow a guest to make a reservcation”, Prio 8/10.
Sprints: Dauern 2-4 Wochen wo effektiv entwickelt (design, code, tests) wird **ohne Änderungen der Requirements**.

Rollen

Product Owner: Definiert die Features, Release-Datum und priorisiert. Er sagt, ob er mit dem Produkt zufrieden ist.
Scrum Master: Sorgt dafür, dass Scrum-values umgesetzt werden. Entfernt Hindernisse, damit das Team arbeiten kann. Schirmt das Team von externen Problemen ab. Organisiert Meetings.
Team: 5-9 Leute, alle entwickeln (dev, test, design) ± alles, arbeiten Vollzeit und organisieren sich selbst.

Zeremonien



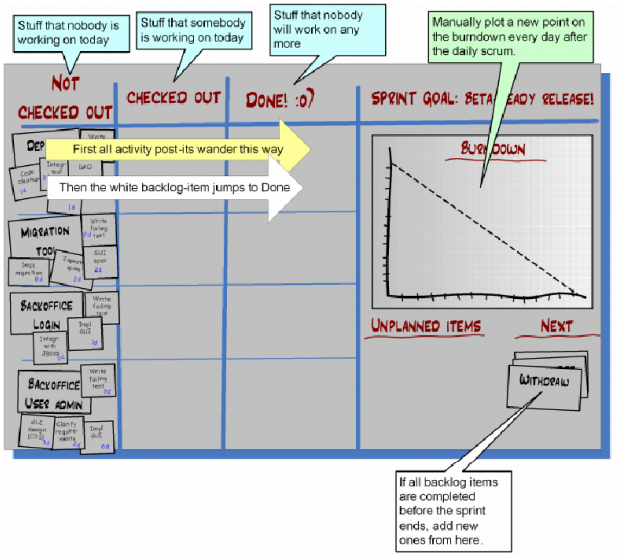
Sprint: Es wird ein Feature oder ein grösseres Stück Arbeit implementiert, und zwar *fix fertig*.
Sprint Planning: **Das Team alleine** wählt Sachen von dem Product Backlog aus, die es machen können sollte im nächsten Sprint. User Stories werden in Tasks umgewandelt (ausdiskutiert). Wichtig ist das Sprint Goal (was mach schaffen möchte), wie viele Leute dabei sind (und zu welchem Kontingent), ein Sprint Demo Date festgelegt und Daily Scrup-Zeit/Ort festgelegt.
Sprint Review: Team präsentiert, was es gemacht hat (oft als Demo). Informal, keine Slides. Das ganze Team nimmt teil, jeder (auch Product Owner) sind willkommen.
Sprint retrospective: Periodisch schauen, was wie läuft. Während 15-30 min nach jedem Sprint mit dem ganzen Team machen. Das Team diskutiert *(Start|Stop|Continue) doings*.
Daily Scrum: 15 Minuten Stand-up-Meeting. Alle können zuschauen, nur ScrumMaster, Product Owner und Team Members dürfen sprechen (checken/pigs). **Nicht um Probleme zu lösen**, sondern um Fragen zu beantworten:

1. Was habe ich gestern gemacht?
2. Was werde ich heute tun?
3. Gibt es ein Problem?

Board/Workflow: Auf dem Board werden alle Tasks eingetragen, wo sie stehen und es wird auf der Burndown-Chart aufgezeichnet, wieviele Punkte man holt, gewichtet mit dem Planning Game. Je höher ein Task ist, desto höher ist er auch priorisiert.

Artifacts

Sprint Backlog: Individuen wählen selbst, was sie machen (es wird nie Arbeit zugewiesen!), es wird täglich aktualisiert und jeder aus dem Team kann den Sprint Backlog ändern. Wenn etwas unklar ist, dafür mehr Zeit reservieren und es, wenn man es macht, aufteilen und genau schauen, was es ist.
Burndown-Chart: Trackt/verfolgt, wie viel Arbeit übrig ist. Sollte möglichst linear nach unten rechts verlaufen.Wenn es nicht so ist, weniger bzw. mehr Tasks zuweisen.



Scrum vs XP

Scrum – Project Delivery Approach:

- Incremental & Iterative: Sprints mit production quality code
- Analysis, design, development, testing in jeder iteration
- Team design and architectural input
- Self organizing cross-functional teams - including the customer
- Minimal creation of “Interim” documents - focus on code delivery

XP – Engineering Practices :

- Continual Refactoring
- Simplest solution that fulfills non-/functional requirements
- Automated Unit testing & Code Coverage checking
- Test driven development
- Continuous integration
- Peer Code reviews / pair programming

TDD

Test-driven dev means that you write an automated test, then you write just enough code to make that one test pass and refactor the code primarily to improve readability and remove duplication.

Incremental Design

Keep the design simple from start and continuously improve it, rather than trying to get it all right from the start and then freezing it.
This is mostly an automatic side effect of doing TDD.

Collective Code Ownership

Teams with a high level of collective code ownership are very robust, for example their sprint doesn’t die just because some key person is sick.
Pair programming with frequent rotation of pairs automatically leads to a high level of collective code ownership.

Estimating and Planning

Man macht das, um zu sagen, *was* man entwickeln soll.

Gute Planung reduziert Risiken, fördert bessere Entscheidungen, baut Vertrauen auf, transportiert Informationen zu allen Beteiligten.

Aglie Planning, nicht Aglie Plans

Pläne werden oft geändert, denn in der Zukunft weiss man mehr oder die Anforderungen haben sich geändert. Pläne sind oft fehlerhaft und unsicher, daher haben *sie selbst* nicht viel Wert.

Definition: Agile planning is focused more on the planning than on the creation of the plan, encourages change, results in plans that are easily changed, and is spread throughout the project .
Schlüsselidee von Aglie Planning: A project rapidly and reliably generates a flow of useful new capabilities and new knowledge.
Mit neuem Wissen können bessere Entscheidungen getroffen werden/besser geplant werden.
Plan Levels: Daily, Iteration, Release.

Feedback von Agile Planning

Man hat immer **Conditions of Satisfacton** beim Planen: täglich, wöchentllich, vom Product Owner.

Estimating Story Points

Story Points: Relative Messgrösse von der Komplexität eine Story.
Geschwindigkeit eines Projekts misst in Story-Point-Druchsatz eines Teams pro Interation.

Das Team schätzt am besten. Auch wenn es immer noch daneben liegen kann. Menschen schätzen gut in kleinen Bereichen von 1..10. Man schätzt also z. B. mit 1, 2, 3, 5, 8 oder 1, 2, 4, 8

Granularität

User-Story: Feingranular bis 8 Punkte.
Epic: Grosse User-Story.
Theme: Mehrere User-Stories, die z. B. etwas miteinander zu tun haben und daher lieber zusammengenommen werden.

Um abzuschätzen, was man hat, kann man zur Schätzscala noch die Werte 13, 20, 40, 100 hinzufügen.

User Stories

Example: As a user, I want to be able to cancel a reservation.

Describes a **WHO**, **WHAT**, and **WHY** scenario from user perspective and delivers **value to the user**. Muss klein genug sein, um gut zu schätzen (1..8 Punkte) und genau genug, um zu testen. Schneller als Use Cases, kompakter und einfacher handhabbar.

Software requirements is a communication problem. User müssen mit Entwicklern reden. Face-to-Face am besten, dann Telefon. Wenn es Änderungen gibt, diese auf die Karte schreiben.

Rollen

Die Rollen möglichst gut bezeichnen, z. B. “der Neukunde” statt einfach “der User”, damit die Sicht der Rolle klarer wird.

3 Cs

Card: A written description of the user story for planning purposes and as a reminder.
Conversation: A section for capturing further information about the user story and details of any conversations.
Confirmation: A section to convey what tests will be carried out to confirm the user story is complete and working as expected.

Details = Conditions of Staisfaction

Stehen auf der Rückseite, damit es möglichst simpel und günstig ist. Die Conditions of Stisfaction werden erst geschrieben, wenn die User Story implementiert wird.

User Stories erzeugen

Verschiedene Möglichkeiten:

- **Fragen**: Umfragen, Fragebogen, Interviews (sehr teuer).
- **Beobachten**: Z. B. bei SSB-Automat oder mit Kamera auf PC-Monitor und User. Letzteres ist allerdings fragwürdig.
- **Story-Writing-Workshop**: Alle sind dabei (Devs, Kunden, Product Owner) und machen Brainstroming. Ziel ist, **möglichst viele** User Stories zu schreiben. Es wird nicht priorisiert.

Gute User Stories: INVEST

Independent: Jede Story kann isoliert entwickelt werden.
Negotiable: Flexibel gehalten und änderbar.
Valueable: Für Users oder Product Owner, nicht für Devs. Wenn doch für Devs, dann sollte sie umgeschrieben werden, dass es für die Users oder Customers relevant wird.
Estimatable: Nötig, da man damit planen muss.
Small: Sollten klein und kompakt sein. Sonst aufteilen in mehrere.
Testable: Ist klar, müssen testbar sein.

Planning Poker

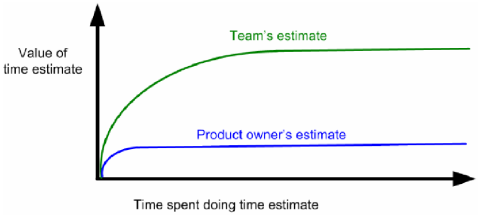
1. Jeder Schätzer erhält Karten mit den Werten
2. Alle lesen die Story-Card und diskutieren sie kurz.
3. Jeder schätzt die Punkte und wirft die Karte.
4. Wenn die Schätzungen gross abweichen, diskutieren und wiederholen bis ausgeglichen.

Man schätzt dabei die Komplexität und nicht die Zeit!
Zeit = Stroy Points / Velocity

Release Planning

= Was bis wann gebaut sein soll. Auf 6 bis 9 Monate planen.

Vor Planning müssen Kriterien bekannt sein: Geld, Zeit, Leute... . Es wird sofort klar, ob das Projekt date- oder feature-driven ist.
Das Team schätzt alle User-Stories, die der Product Owner will. Er muss dabei die Fragen beantworten.



Iterationslänge definieren: Typischerweise 2 bis 4 Wochen. Das hängt ab von: Zeit bis zum Release, Unsicherheiten, wie schnell man Feedback erhalten kann, Overhead der Iteration, etc..
Velocity schätzen: Erfahrung verwenden, 1,2 oder 3 Iterationen durchlaufen lassen, oder eine grobe Prognose machen, wie es etwas sein könnte (das ist aber unzuverlässig!).
Priorisieren und Release Date / Features setzen: Der Product Owner sortiert die User-Stories nach Relevanz (aus seiner Sicht). Dies ist der **Release Plan**. Man kann sie dabei kategorisieren und sagen, was bis zu welchem Release fertig sein soll. Dann ist es f**eature-driven**! Wenn man sagt, man möchte an Tag X releasen, ist es **date-driven**, man weiss halt noch nicht so genau, was im Release enthalten sein wird.
Update: Der Release Plan wird typischerweise vor jeder Iteration aktualisiert.

Iteration Planning

Konzentriert sich auf die Iteration (Sprint) also auf 1..4 Wochen.
Die **User-Stories** werden in **Tasks** aufgesplittet und jeder Task wird auf **ideale Studnen** geschätzt.

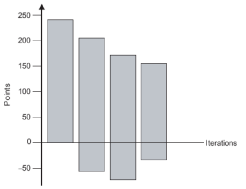
Ideal Hours: An hour of work where you can solely focus on the task at hand **without any interruptions** like calls, emails,... In Scrum Team, Members never estimate real hours but only ideal work hours because experience showed that this matches the way developers tend to think about problems. An 8 hour day could have **6 ideal hours** for every team member.
Ideal Day: Relativ zu Ideal Hours, aber zu grob (Task zu gross).

4 primäre Schätzfaktoren

- Wieviel Geld das Feature bringt.
- Entwicklungskosten (und Support?) vom Feature
- Menge neuen Wissens, das die Entwicklung bringt
- Menge an Risiko, die durch die Entwicklung entfernt wird

Release Burndown Bar Chart

Gibt an, wie weit das Release ist. Alle erledigten Punkte werden oben abgezogen. Wenn Punkte erhöht werden müssen (bei Neuschätzung), kommen diese oben hinzu. Hinzugefügte Features kommen unten hinzu.



No Silver Bullet

Frederick P. Brooks, 1987

Software-Qualities:

- **Essential (bzgl. Design of SW):**
  Complxity: Ideen in Code umwandeln (genau Verstehen). Wenn SW grösser wird, wird Komplexität grösser! Verständnis-, Lese-, Schreibprobleme, schwer änderbar.
  Invisible: SW unsichtbar, es ist ein Ram-Zustand. Macht es kompliziert, zu kommunizieren.
  Changeable: 2 Aspekte: Software oder Maschine ändert.
  Conformity: Wenn etwas nicht passt, soll SW sich ändern.
- **Accidental (Impl and Testing):**
  High Level Langs: Abstraktion mit OO, aber nicht unendl.
  Time Sharing & better OS support: wird besser

Promising areas to build better SW:

- **Buy vs build**: Man zahlt was läuft
- **Incrementeal, iterative dev with feedback from end users**: helps to redefine requirements, prove design and improves morale
- **Identify, retain and coddle great desinger**: Gute Leute finden & halten.