Zusammenfassung

**Lernziele**

* Aktuelle Themen aus der Software Entwicklung kennen, anwenden und kritisch einschätzen
* Werkzeuge und Techniken professioneller Software Entwicklung kennen und anwenden
* Pragmatische Prinzipien der Software Entwicklung kennen und anwenden

**Unterlagen / Bücher**

* Keine, nur Vorlesungsfolien und Übungsaufgaben

**Lerninhalte**Der Lerninhalt dieses Kurses richtet sich stark an die aktuellen Themen der Software Engineering. Die Themen werden jedes Jahr auf deren Relevanz geprüft und bei Bedarf angepasst. Es wird daher stark empfohlen, die Prüfung im gleichen Semester zu belegen. Hier ist eine Liste der in der Vergangenheit behandelten Themen:

* Project Planning
* Project Automation
* Test Driven Development
* Pragmatic Software Engineering Practices
* Error Handling Design
* Concurrency Design
* Design by Contract
* Software Architecture
* Code Smells
* Design Patterns
* Refactoring
* Software Metrics
* Software Reviews
* Cost Estimation (Aufwandschätzung)
* Performance Profiling
* Agile Software Development
* Programing in the functional style
* Scripting Languages
* Software Failure Analysis

Übersicht des Modules

[Erste Woche 3](#_Toc475179144)

# Projektplanung

## Drei Themenkreise

Je mehr Leute an einem Projekt mitarbeiten, umso mehr müssen Sie organisieren und planen, insbesonders wenn nicht alle Leute an einem Ort sitzen.

Wie plant & organisiert man ein Software-Projekt?

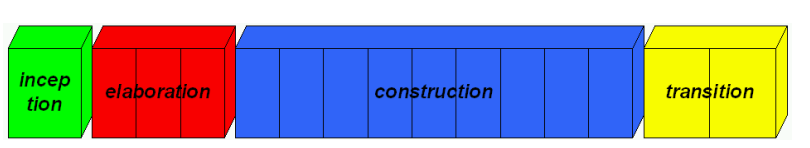
Dieses Kapitel deckt folgende drei Themenkreise ab:

* Agil, RUP, Scrum, Wasserfall, iterativ?
* Von Use Cases zu Arbeitspaketen
* Diagramme und Dokumentationen in Software-Projekten

## Agil, iterativ

Ganz klar: mit dem Wasserfall-Modell (auch V-Modell) gewinnt man keinen Blumentopf, da sind sich praktisch alle einig. Heute gilt:

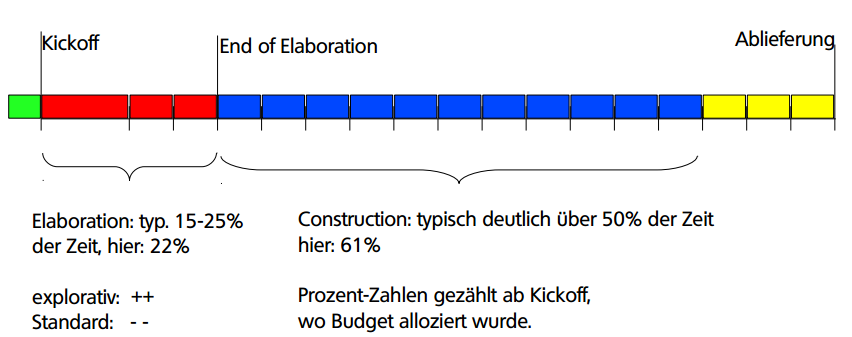
* Agil (Scrum; eXtreme Programming XP)
* Iterativ (RUP, Iterative Development, Spiral Development...)



### Ein Beispiel-Projekt mit 18 Iterationen

Total 8.5 Monate (37 Wochen) in 4 Phasen. Eine Inception von 10 Tagen, eine Elaboration von 8 Wochen in 3 Interationen, eine Construction von 11 Iterationen à 2 W. und ein Transistion von 3 Iterationen à 2 Wochen.

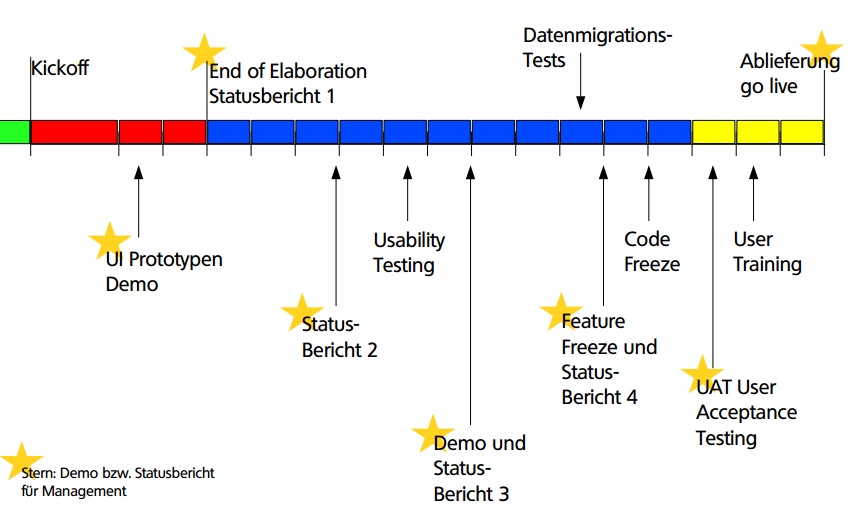
**Anzahl Personen**  
Personal wird vor allem in der intensiven Constuction-Phase benötigt. Wichtig ist das der Chief Architect von Kickoff bis mindestens Mitte Projekt involviert ist.

Zeitaufteilung  
Achtung: Die Prozent der Zeit ist nicht gleich die Prozent der Kosten.

### Checkliste End of Elaboration

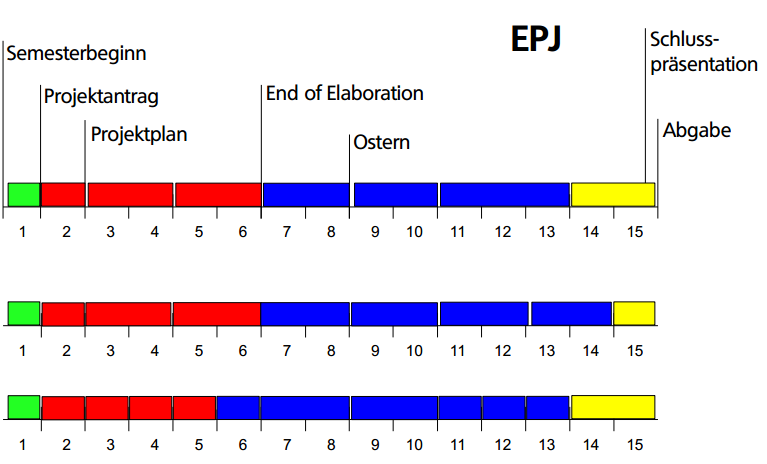
* Anforderungen (Requirements): Haben wir den Kunden verstanden? Funktionsumfang (Scope) ist abgesteckt durch UCs, Domain Model, nicht-funktionale Anforderungen
* User Interface Design: Entwürfe gemacht, dem Kunden gezeigt; wenn möglich Clickable Prototypes plus Grafik-Entwürfe (Farben, Schriften)
* Software Architecture: Entwurf steht, Subsysteme und Interfaces definiert, Prototypen gemacht (Durchstich durch alle Schichten).
* Entwicklungs-Werkzeuge und Methoden: definiert und komplett aufgesetzt (IDE, version control system/server, build server, unit testing, static code analysis tools inkl. Konfig., DEV-TEST-PROD Server, ticketing & bug tracking, user story writing/proofing, etc.).
* Genauere Aufwandschätzung: Liste der Arbeitspakete

### Checkpoints / Meilensteine

**Meilensteine dokumentieren**  
Die Iterationsplanung, d.h. die Beschreibung der Meilensteine "was läuft wann, bzw. was können wir wann zeigen", darf ruhig in einem Dokument erfolgen.

Sobald Sie die Meilensteine gesetzt haben, können Sie diese in Ihrem Arbeitspaket-Verwaltungssystem (Redmine) als Software-Versionen definieren.

### Empfehlungen für Engineering-Projekte

Es gibt verschiedene Varianten für das Engineering Projekt. Version 3 ist eher einem realen Projekt entsprechend mit etwas mehr Risiko. Grundsätzlich gilt die Version 2.

### Meilensteine und Reviews im Eng.-Projekt

### Meilensteine und Versionen in Redmine

### Vorschlag für Umsetzung der Meilensteine in Redmine-Zielversionen.

Die Arbeitspakete werden dann einer dieser Versionen zugeordnet.

P.S.: Die Reviews finden i.d.R. nach Ihren entsprechenden Meilensteinen statt.

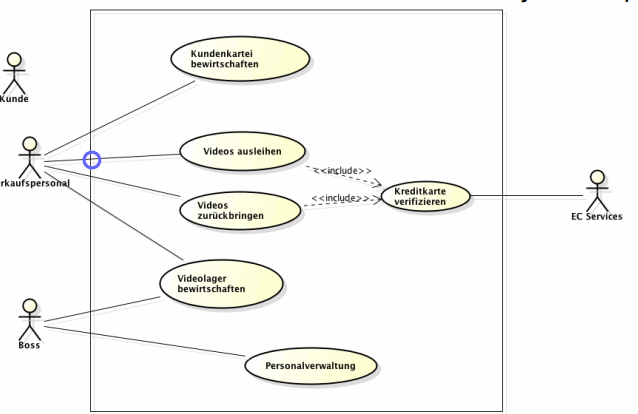
**Redmine-Versionen**

* MS1: Projektplan
* MS2: Anforderungen und Analyse
* MS3: Ende Elaboration
* Alpha
* Beta (Code Freeze?)
* End of Construction
* Optional Features

## Anforderungen: Use Cases etc.

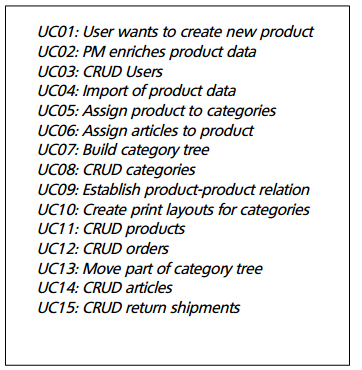
Use Cases sind Beschreibungen klar aus der Benutzersicht, also gut mit dem Kunden kommunizierbar. Use Case beschreiben die Funktionalität im Kontext (z.B. warum, was in welcher Reihenfolge und wie oft?) Es gibt verschiedene Detaillierungsgrade. Von Brief (3 Zeilen), über casual (eine halbe Seite) bis hin zu fully dressed (ein bis zwei Seiten mit vorgegeben Abschnitten). Use Cases können auch gut zum Abstecken des Funktionsumfangs (Scope) verwendet werden.

### Use Case Diagramm

Ist gut für eine Übersicht, was im Scope ist (was nicht), wer darf was machen.

Ein Systemsequenzdiagramm würde den blauen Kreis abbilden.

### Use Cases eines filtiven Beispieles

Insgesamt 15 Use Case. 4 wichtige, komplexe Use Cases. 5 mittel-komplexe Use Cases und 6 CRUD.

### Nicht funktionale Anforderungen

Use Cases beschreiben nur die Funktionalität. Nicht-funktionale Anforderungen ergänzen Use Cases und Domainmodell. Typische Beispiele für nicht-funktionale Anforderungen sind Mengen- und Qualitätsanforderungen:

* Performance ("Antwortzeiten für eine Produkt-Suche bei 100'000...")
* Mengengerüst ("50'000 Artikel; 200 gleichzeitige Besucher; ...")
* Sicherheit (Firewalls, Intrusion Detection, Logging, Plausibility checks...)
* Erweiterbarkeit ("Später automatischer Import von Lieferanten-Daten...)
* Benutzerfreundlichkeit ("Produkt-Manager Einführung in 2 Tagen...")

### Domain-Model

Das Domain-Modell ist das Komplementär zu den Use Cases. Ein Domain-Modell beschreibt, was wir uns während der Laufzeit des Programmes merken, und was wir evtl. darüber hinaus speichern. Vereinfacht ausgedrückt: Use Cases = Dynamik, Domain Modell = Persistenz.

## Anforderungen 🡪 Arbeitspakete

Jetzt haben wir die Anforderungen (zumindest grob) definiert. Warum macht man jetzt Arbeitspakete? Was muss in Arbeitspaketen drinstehen? Wie gross sollen Arbeitspakete sein? Wie viele Arbeitspakete müssen entstehen?

**Inhalt von Arbeitspaketen**  
ID, Titel, Kurze Beschreibung (In der Form «Als AAA möchte ich BBB, weil CCC»), Akzeptanz-Kriterien, Schätzung des Aufwands, Priorität für Kunden, Geleistete Stunden (Zeiterfassung), Status und Arbeits-Kategorie.

**Grösse der Arbeitspakete**  
Maximal 50 – 70% von dem, was eine Person in einer Iteration schafft, damit die Chance gross ist, dass das Arbeitspaket innerhalb der Iteration fertig wird, denn Arbeitspakete dürfen nur «nicht angefangen», «angefangen» oder «fertig» sein. Und wenn ein Arbeitspaket nicht fertig ist, kommt es in die nächste Iteration. Im Falle des Engineering-Projektes mit Iterationen von 2 Woche und 17 Arbeitsstunden pro Person ist die empfohlene maximale Grösse 10 Arbeitsstunden.

**Anzahl Arbeitspakete im Projekt**  
In einem EPJ gibt es typischerweise ca. 80 – 160 Arbeitspakete. 4 Credits für das ganze Projekt ergibt 120 Arbeitsstunden pro Person. Bei einem Team von 4 Personen wären es dann 480 Arbeitsstunden pro Team.

**Arbeitspakete organisieren**  
Arbeitspakete müssen an einem Ort zentral gespeichert sein. Arbeitspakete müssen von allen eingesehen und editiert werden können. Zudem müssen Sie priorisier bar sein (sie werden häufig herumgeschoben). Die Pakete sollten sowohl Schätzungen als auch Ist-Zeiten (Zeiterfassungen) enthalten.

**Workflow Arbeitspakete**  
Definiere Rollen und Zustände/Übergänge für Arbeitspakete in einem Zustandsdiagramm-

Grundsätze zur Arbeitsaufteilung  
Wenn man Arbeit auf verschiedene Teammitglieder verteilen will

(insbesonders wenn das Team noch geografisch verteilt ist), dann muss man wissen, was der Kunde will/braucht, bevor man die Arbeitsaufteilung machen kann. Wenn man Arbeit auf verschiedene Teammitglieder verteilen will (insbesonders wenn das Team noch geografisch verteilt ist), dann muss die Architektur allen Beteiligten klar sein, bevor man die Arbeitsaufteilung machen kann. Das heisst, dass man bis Ende Elaboration eng zusammen arbeitet (kleines Team an einem Ort) und dass die Arbeitsaufteilung erst nach Ende Elaboration klappt: erst danach kann man verteilt loslegen.

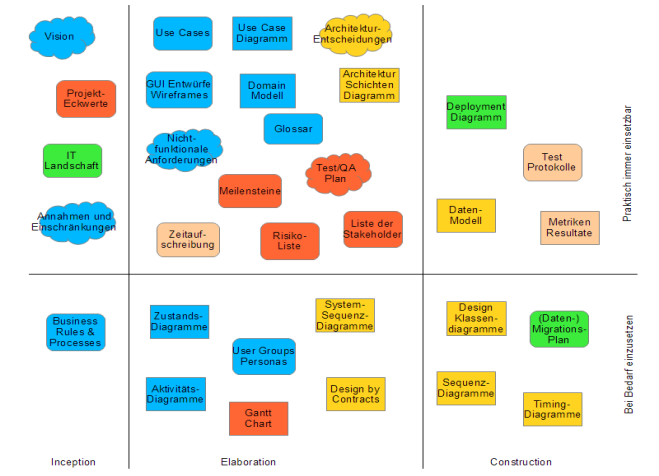
**Planung pro Iteration**  
Genug Arbeitspakete, damit alle im Team während der nächsten Iteration beschäftigt sind. Genau soviele Arbeitspakete wie die Schätzungen zulassen, dass sie auch innerhalb der Iteration fertig werden. Innerhalb des Teams werden die Arbeitspakete eigenverantwortlich zugeordnet, evtl. auch dynamisch verteilt.

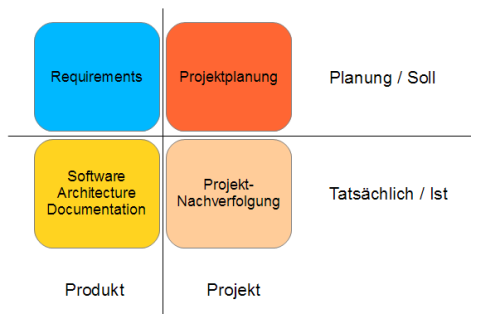
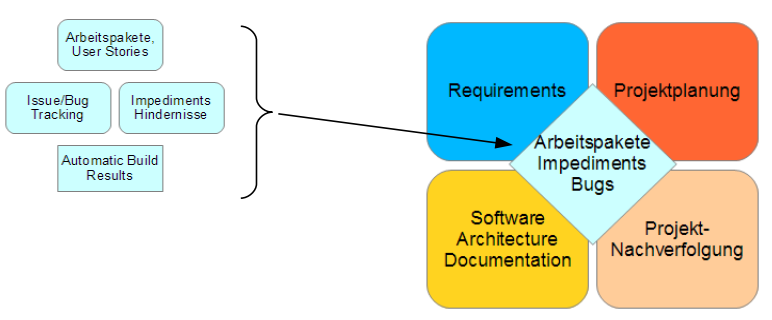
**Regeln bei der Planung**  
Der Entwickler schätzt den Aufwand für die Arbeitspakete, während der Kunde die Arbeitspakete priorisiert. Und nicht umgekehrt. Die einzige Ausnahme sind die architekturrelevanten Arbeitspakete. Diese können vom System-Architekten in bestimmte Iterationen gesetzt werden, weil sonst das System nicht schlau gebaut werden könnte. Dies muss aber den Kunden erklärt werden und der Kunde muss es absegnen.

**Fallstrickte bei Arbeitspaketen**  
Nicht nur generische Arbeitspakete aufführen. Generische Arbeitspakete sind solche, welche in jedem Projekt vorkommen, z.B. 'Domainmodell machen', 'Use Cases schreiben'. Nicht-generische sind z.B. 'Funktionalität für Speichern des Warenkorbs definieren', oder 'Entwurf Level-Editor'. Auch Arbeitspakete für unproduktive Tätigkeiten erstellen, wie z.B. für Besprechungen, Einrichten des Servers, Schreiben des Testplans (werden manchmal separat als ‚Chore‘ geführt, nebst ‚Ticket‘ und ‚Bug‘). Formulieren Sie ihre Arbeitspakete (wie generell auch alle Anforderungen) 'abhakbar', d.h. tabellarisch und so, dass sie abgehakt werden können => klein genug und gute Akzeptanzkriterien. Und nicht vergessen, Arbeitspakete schreiben kostet auch Zeit.

## Diagramme und Dokumente

Erste Spalte Inception, zweie Spalte Elaboration und dritte Spalte Constuction.



Was gehört in welches Dokument?  
Diese Frage ist zweitrangig. Hauptsache wir haben alles dokumentiert, was es wert ist, festgehalten zu werden. Wichtig ist m.E. nur die Zuordnung zu einem der vier Quadranten der Dokumentation. Tatsächlich sind es aber 5 Dokumenten-«Kübel».  
 

## Projektdokumentation

Sie schauen, dass von all Ihrem Code und Ihren Dokumenten laufend Backup gemacht wird (git/SVN Backup), damit nichts verloren geht. Klar. Zusätzlich solle aber bei jedem Meilenstein die Doku eingefroren und gespeichert werden. Das heisst:

* Ein PDF von jedem relevanten Dokument mit Datum im Dateinamen
* Exporten von allen Web Tools, inbesondere Redmine (alle Arbeitspakete, Bugs und Zeitaufschreibungen als CSV Dateien mit Datum im Namen).
* Screenshots von allen wichtigen Funktionen (mit Funktion und Datum im Dateinamen)

**Es ist ihr Gedächtnis**Speichern Sie die Projektdokumentation wie oben beschrieben, denn Sie wissen nicht, was Sie in Zukunft aus den alten Projektdaten (zum Vergleich, zum Abschätzen) herausholen wollen.

Deshalb: speichern Sie alles mögliche über das Projekt in einem lange haltbaren und portablen Format (TXT, CSV, PDF), damit Sie später alles noch lesen, nachvollziehen und nachberechnen können.

Beispielsweise: „Wieviele Stunden haben wir damals für die Vorbereitung und Durchführung der Usability Tests gebraucht? Wieviele Probanden waren dabei? Was für Szenarien haben wir benutzt? Was ist dabei herausgekommen?“

Projektdokumentation ist Ihr Erfahrungsschatz

## «End of Elaboration» = Wendepunkt

Der Zeitpunkt ‘End of Elaboration’ ist auch ein Wendepunkt für die Dokumentation und Kommunikation. Vor ‘End of Elaboration’ liegen die Haupt-Anstrengungen bei der Dokumentation darauf, zu zeigen, dass man den Kunden verstanden hat. Diese Dokumentation ist zum grössten Teil für die Kommunikation mit dem Kunden gedacht, sollte also auch in seiner Sprache (z.B. Deutsch) gehalten sein. Nach ‘End of Elaboration’ ist der Fokus auf dem Bauen der Lösung, d.h. die Dokumentation, die entsteht, ist hauptsächlich für die Entwickler. Da kann es sein – bei einem ausgelagerten Entwicklungsteam – dass die Dokumentation nach Ende Elaboration überwiegend auf Englisch gemacht werden muss.

**Diagramme sind Kommunikation**  
Diagramme sind oft besser als Worte, da Sie präziser/formaler sind. Sie müssen eine normierte Bedeutung (UML) haben. So spart man sich Erklärungen. Zudem sollten Sie kommunizierbar sein d.h. beschränke Grösse. (A3 Druck oder 3x FullHD Bildschirm).

**Je früher desto besser**  
So früh wie möglich, so formal wie möglich (Datenmodell, Zustandsdiagramme). Zudem so früh wie möglich so komplett wie möglich (Use Cases brief, Prototypes). Denn wenn man Fehler, Inkonsistenzen oder Auslassungen früh entdeckt, spart das viel Geld und Ärger.

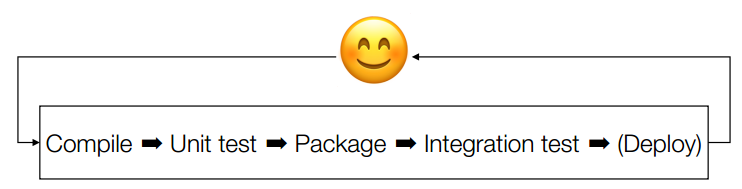
# Projektautomatation

## Wie entwickeln wir Software?

Ein Entwickler führt immer wieder folgende Aktivitäten aus: Kompilieren, Unit Testing, Paketieren, Integrationstests und «Deployen/Veröffentlichen». Die Aktivitäten werden pro meist in sehr kleinen Abständen wiederholend ausgeführt. Dabei drehen wir als Entwickler fast durch, da wir so vieles machen müssen. Ein schlauer Mensch hat einmal gesagt: «Automatisiere alles, was du mehr als einmal brauchst».

### Idee 1 – Build Skript

Die erste Idee. Ein einfaches Skript, welches diese Schritte ausführt. Von den Kompilierung über das Testing bis hin zur Paketierung.

**Workflow**  


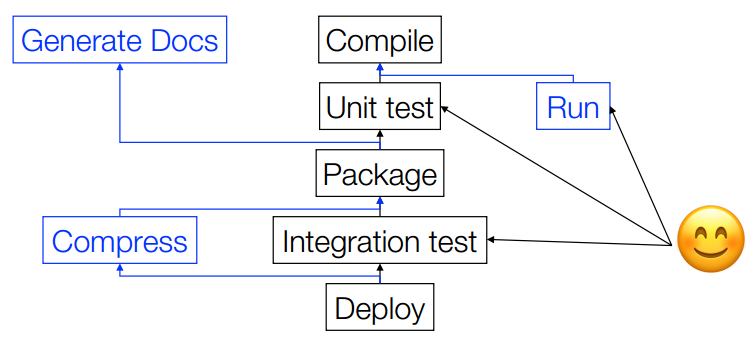
**Vorteile**  
Die ganze Sache ist nun automatisiert in einem nicht interaktiven Prozess. Das Skript kann mehrere Male ausgeführt werden, es ist also repetierbar. Zudem ist es unabhängig von der IDE. Nicht zuletzt können zeitintensive Tasks terminiert werden.

**Nachteile**  
Der Prozess ist zwar automatisiert, aber eben nicht interaktiv. Die Wartung und Erweiterung ist aufwändig und zudem sind die Skripts platformabhängig (Powershell, Bat, SH).

### Was wir möchten (Wunschliste)?

* Single Command Build (CRISP)
  + Complete – Jeden Build von neu aufbauen
  + Repeatable – Immer wieder anstossen, auch älteren Code wieder auschecken und builden können
  + Imformative – Testresultate von Unit- und Integrationstests
  + Schedulable – Zeitlich terminierbar
  + Portable – An verschiedenen Orten ausführen können
* Flexibel
* Leistung – der Build sollte nicht allzu lange dauern
* Erweiterbarkeit

Die Lösung für die Wunschliste ist ein Build Tool, ein spezialisiertes System welches den ganzen Build Prozess verwaltet. Das Core Konzept schaut wie folgt aus.

Zu den weiteren Features gehört z.B. das Dependency Management oder die Optimierung des Build Prozesses (Parallel) sowie Anpassungen beim Testing oder beim Processing.

## The Beginning

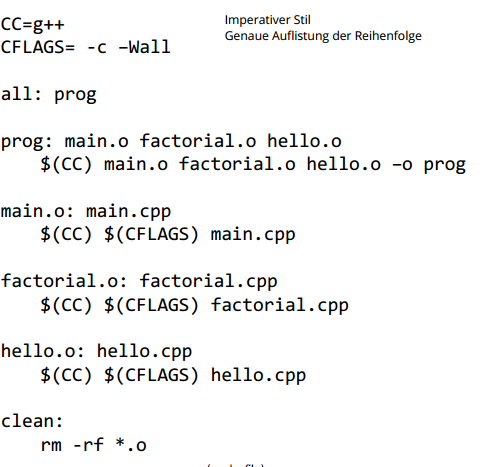
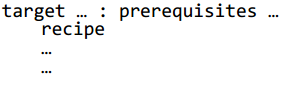
### Der Beginn der Build Automatisierung – GNU Make (Imperativ)

Der Beginn war Make für die Sprache C auf UNIX. Die erste Version wurde in 1976 von den Bell Labs entwickelt. Es lässt sich wie ein «flexibles» Build Skript vorstellen. Make hat das DAG Konzept mit Targets und Abhängigkeiten eingeführt.

Der Stil ist imperativ (Shell-Scirpts). Die ganze Sache ist leider Plattform abhängig und hat kein automatischen Dependency Management. Gearbeitet wird mit Targets / Dependencies und Variables. Die Auflistung muss in der richtigen Reihenfolge erfolgen.

Der Build Author definiert explizit das DAG. Die Targets sind mit einer Skriptsprache implementiert und meistens basiert es auf einem externen Dependency Manager.

**Beispiel**



Neben Make für C gibt es auch Jake (Javascript), nmake(.NET) oder Psake für Powershell. Um 2000 wurde dann Ant eingeführt, welches auf XML basiert. 2003 entstand Rake, in welchem die Targets mit Ruby definiert werden.

### Apache Ant (Imperativ)

Es ist ein XML-basiertes Skripting mit bereits integrierten Taks wie mkdir oder jar oder condition. Der Fokus liegt auf der Portability. Eigene Tasks können in Java geschrieben werden.

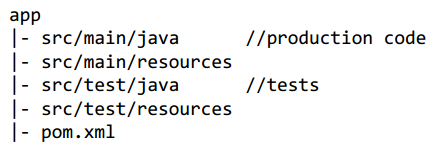
**Vorteile**  
Es umfangreich und flexibel.

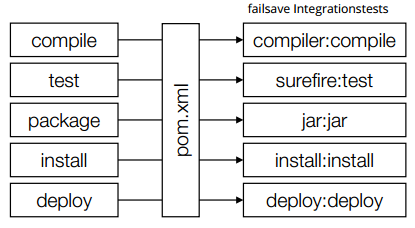
**Nachteile**  
Die Build Definitionen tendieren dazu sehr komplex zu werden. Zudem ist es schwierig die Build Logik wieder zu benutzen. Meist wird Copy Paste angewendet.

### Deklarative Builds mit Apache Maven

Der erste Prototyp erschien im 2001, 2004 wurde Maven 1.0 lanciert. Damit soll aufgehört werden das Rad neu zu erfinden. Es ist entworfen für Konsistenz über mehrere Projekte. Zudem beinhaltet es ein automatisches Dependency Management.

Beispiel  
Es ist deklarativ und im XML gehalten. Die Konvention steht über der Konfiguration (Default Build, nur Abweichungen angeben). Die DAG’s sind die vordefinierten Lifecylces. Zudem setzt Maven eine gewisse Projektstruktur voraus.





Im Gegensatz zu anderen Tools gibt der Build Author an, wie das Build Resulat sein sollte. Erweiterungen und Anpassungen findet über Plugins statt.

**Vorteile**  
Kleinere Build-Dateien, Wiederverwendbare Build Logik (Plugins), Automatisches Dependency Management

**Nachteile**  
Weniger generell und flexibel als «Imperative» Tools, macht mir Vorschriften wie meine Projektstruktur sein sollte.

### Post-Maven Tools

Deklarative Tools sind ein guter Ansatz, aber sind teilweise zur restrektiv. Die aktuellsten Tools probieren das Beste aus den beiden Welten zu verwenden. Zudem haben jene auch Fortschritte in Performance und User Experience gemacht.

2008 Apache Buildr  
2008 SBT  
2009 Gradle

All diese Tools sind mit dem Dependency Management von Maven kompatibel.

## The Future

In der Zukunft wird immer mehr Automatisierung gebraucht. Es lohnt sich also die Zeit zu investieren, ein Build Tool zu erlernen. Beginner sollten sich am besten mit Maven befassen. Die automatisierten Build sollten ab Tag 1 eingesetzt werden.

### Zusammenfassung der Vorteile

* Reduktion der repetitiven Tasks
* Unabhängigkeit von der DIE
* Reproduzierbare Resulate
* Zeit sparen
* Basis für Continuous Integration (nächstes Kaptiel)

### Continuous Integration

«Team-members integrate their work frequently. Usually, each person integrates at least daily, leading to multiple integrations per day.”

#### Ziele

Wir möchten zur jeder Zeit ein lauffähiges Produkt haben. Feedback möchten wir sicher im Falle von Fehlern (Automatisierte Tests, Analysis Tools).

#### 10 CI Praktiken

**1. Maintain a single source repository**Nutzen Sie ein Source Code Management System, so Weiss jeder wo der Code abgelegt wird. Gearbeitet werden soll nicht auf dem Master-Branch.

**2. Automatisierte Build**

**3. Machen Sie Build selbst-testbar**  
Erstellen und pflegen Sie eine automatisierte Test Suite.

**4. Jedermann commitet täglich auf die Mainline**  
Sonst kann ich keine tägliche Builds machen. Zudem reduziert es den Merging Aufwand und neue Bugs können schnell gefunden werden.

**5. Jeder Commit auf die Mainline soll gebuildet werden**Jeder Änderung auf der Mainline sollte als kompletter Build auf dem CI Server ausführbar sein.

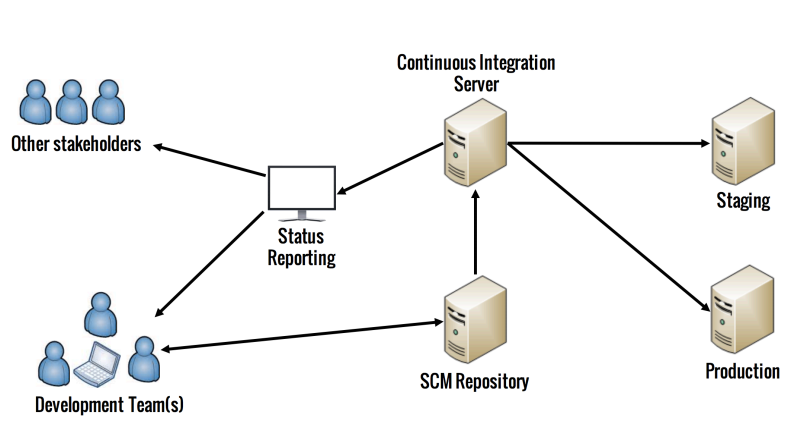
**6. Der Build sollte schnell gehalten werden**Um schnelles Feedback zu bekommen. Meist geschieht dies in mehreren Schritten (1. Build, 2. Integrations Tests und 3. Performance Tests).

**7. Testen in einem Klone der Produktionsumgebung**  
Test und Produktion sollten so ähnlich wie möglich sein, damit das Test-Feedback so genau wie möglich ist. Hier könnte zum Beispiel Docker verwendet werden.

**8. Machen Sie es einfach den die letzten Produkte zu erhalten**  
Jeder welcher in der Entwicklung des Projekt involviert ist sollte auf die aktuellste Version des Produktes Zugriff haben.

**9. Jedermann kann sehen was passiert**  
Status des Builds mit Rot und Grün.

**10. Automatisches Deployment**  
Automatisches Deployment vom Produkt nach den Tests auf eine Testumgebung.



#### Beste Entwicklerpraktiken

* Der Code sollte häufig commited werden
  + Nur kleine Änderungen machen
  + Commiten nach jedem Task
* Commiten Sie keinen fehlerhaften Code
* Kaputte Builds sollten sofort repariert werden
* Schreiben Sie automatisierte Tests
* Lassen Sie Builds lokal laufen (bevor der Code commited wird)
* Setzen Sie ein CI ab Tag 0 auf

#### CI Server

Ein CI Server lässt automatisierte Builds und Tests laufen. Über ein Web Interface oder ein Chat System publiziert dieser Resulate. Ein CI Server ist event-gesteuert. Entweder für Intervalle, manuell oder über Änderungen am Code (Commits). Es gibt noch weitere Features wie Quality Analysis oder IDE Integration. Dort sind keine Grenzen gesetzt.

**Beispiele**  
Open Source (Jenkins, Go, BuildBot, Strider), Commercial (Bamboo, TeamCity) und Cloud Based (Travis CI, Drone.io und GitLab CI)

# Vorlesung Woche 3