ISTQB - Agile Extension

1. Agile SW Entwicklung

1.1 Die Grundlagen der agilen Softwareentwicklung

- Whole-Team Approach
- Entwickler, Fachbereichsvertreter und Tester = gleichberechtigte Mitglieder des Teams
- Regelmässige Kommunikation untereinander → frühzeitige Fehlerreduzierung & qualitativ hochwertiges Produkt

1.1.1 Agile Softwareentwicklung und das agile Manifest

Hoher Stellenwert:		**dennoch wichtig:**
• Individuen und Interaktionen	sind wichtiger als	Prozesse und Werkzeuge
• Funktionierende Software	ist wichtiger als	umfassende Dokumentation
• Zusammenarbeit mit dem Kunden	ist wichtiger als	Vertragsverhandlungen
• Reagieren auf Veränderungen	ist wichtiger als	das Befolgen eines Plans

12 Prinzipien [agilemanifesto Prinzipien]

1. Prio1: Kunden durch frühe und kontinuierliche Auslieferung wertvoller SW zufrieden stellen
2. Anforderungsänderungen, selbst spät in der Entwicklung, werden begrüßt. Wettbewerbsvorteil!
3. Funktionierende SW wird regelmäßig innerhalb weniger Wochen (oder Monate) geliefert
4. Fachexperten und Entwickler müssen während des Projektes täglich zusammenarbeiten
5. Projekte werden rund um motivierte Individuen aufgebaut
6. Effizienteste und effektivste Kommunikation: von Angesicht zu Angesicht
7. Funktionierende Software ist das wichtigste Fortschrittsmaß
8. Agile Prozesse fördern nachhaltige Entwicklung, gleichmässiges Tempo
9. Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität
10. Einfachheit - die Kunst, die Menge nicht getaner Arbeit zu maximieren - ist essenziell
11. Selbstorganisierte Teams → beste Architekturen, Anforderungen und Entwürfe
12. Regelmässige Selbstreflexion des Teams → wie werden wir effektiver? Verhalten anpassen

FIRMTEC SOLUTIONS AG

1.1.2 Agile Softwareentwicklung und das agile Manifest

- Whole-Team Approach → interdisziplinäres, selbstorganisiertes Team
  - Fachbereich, Entwickler, Tester, andere Stakeholder → Auch Vertreter des Kunden
    - Fördert die Kommunikation und Zusammenarbeit innerhalb des Teams
    - Nutzt die unterschiedlichen Fähigkeiten aller als Beitrag zum Erfolg
    - Erhebt die Qualität zum Ziel jedes Einzelnen
- Agiles Team sollte KLEIN sein → Ideal zwischen 3 – 9 Personen
- Selbstorganisation
- Am besten alle im gleichen Raum → Ideale Kommunikation und Interaktion
- Tägliche Stand-Up Meetings → Arbeitsfortschritt? Hinderungsgründe (Impediments)?
- The Power of Three → Tester, Entwickler, Fachbereichsvertreter

1.1.3 Frühe und regelmässige Rückmeldung

- Kurze Iterationen → früher & kontinuierliche Rückmeldungen
- **Sequentiell** → Kunde sieht Produkt spät → Rückmeldungen schwieriger umzusetzen
- **Agil** → regelmässige Rückmeldung → Kurzfristige Anpassungen möglich
- Vorteile früher und regelmässiger Rückmeldung:
  - Vermeiden von Missverständnissen bezüglich der Anforderungen
  - Bessere Abdeckung von Kundenwünschen durch frühe Nutzung der SW
  - Frühes Entdecken, Isolieren und Lösen von Qualitätsproblemen durch Continuous Integration
  - Liefern von Informationen für das agile Team bezüglich seiner Produktivität und seiner Fähigkeiten
  - Fördern einer beständigen Projektdynamik

1.2 Aspekte agiler Ansätze

- Gemeinsame Erstellung von User-Stories, Retrospektiven, Continuous Integration, Agiles Planen für Release/Iterationen

1.2.1 Ansätze agiler Softwareentwicklung

Extreme Programming (XP)

- 5 Werte, die die Entwicklung leiten sollen: Kommunikation, Einfachheit, Rückmeldung, Mut, Respekt
- Liste von Prinzipien:
    - **Humanity (Menschlichkeit)** Schaffen einer den menschlichen Bedürfnissen der Projektmitglieder ausger. Atmosphäre
    - **Economics (Wirtschaftlichkeit)** Die Entwicklung ist sowohl wirtschaftlich als auch wertig
    - **Mutual benefit (beidseitiger Vorteil)** Die Software stellt alle Beteiligten zufrieden
    - **Self-similarity (Selbstähnlichkeit)** Wiederverwendung bestehender Lösungen
    - **Improvement (Verbesserung)** Aus regelmäßig gewonnenem Feedback
    - **Diversity (Vielfalt)** Vielfalt im Team (Fähigkeiten, Charaktere) erhöht die Produktivität
    - **Reflection (Reflexion)** Erkennen besserer Lösungen durch stetige Reflexion
    - **Flow (Gleichmäßig mit hoher Konzentration arbeiten)** Ein stetiger Arbeitsdurchfluss & kurze Iterationen
    - **Opportunity (Gelegenheiten wahrnehmen)** Schwierigkeiten/Fehlschläge bei der Umsetzung = Chancen
    - **Redundancy (Redundanzen vermeiden)** Vermeidung Unnötig wiederholte / manuelle Schritte → Automatisieren!
    - **Failure (Fehlschläge hinnehmen)** Nicht optimale/fehlerhafte Umsetzung wird akzeptiert → Herausforderung
    - **Quality (Qualität)** Hohe Softwarequalität ist wichtig
    - **Baby steps (kleine Schritte)** Zeitnahes Feedback & Fehlschlagkompensation, Flexibilität bei Rahmenbedingungen
    - **Accepted responsibility (akzeptierte Verantwortung)** Übernahme von Verantwortung durch Team → Keine Vorschreibung durch Management
- Die 13 primären Praktiken von XP:
    1. **Sit together (Räumliche Nähe)** Optimieren der Kommunikation durch gemeinsame Anordnung der Arbeitsplätze
    2. **Whole-Team (Interdisziplinäres, selbstorganisiertes Team)** Bewusstsein = nur als Gemeinschaft erfolgreich
    3. **Informative Workspace (Inform. Arbeitsplatz)** Wichtige Infos vom Arbeitsplatz aus sichtbar (akt. Tasks, Projektstand)
    4. **Energized work (Energievolle Arbeit)** Motiviertes Arbeiten in entspannter Atmosphäre. Entwickeln ohne Überstunden.
    5. **Slack (Entspannte Arbeit)** Erläuterung siehe energized work
    6. **Pair programming (Programm. in Paaren)** Selbstregulierende Teams → Programmieren mit wechselnden Partnern
    7. **Stories (Stories)** Die zu entwickelnde Funktionalität wird in Form von User-Stories beschrieben
    8. **Weekly cycle (Wöchentlicher Zyklus)** In wöchentlichem Zyklus wird entschieden, was als nächstes umgesetzt wird
    9. **Quaterly cycle (Quartalsweiser Zyklus)** Das Projekt selbst wird in quartalsweisen Zyklen geplant
    10. **Ten-minute build (10-Minuten-Build)** Build und (automatisierte) Tests in max. 10 min. durchgeführt
    11. **Continuous integration (Kontinuierliche Integration)** Bereitstellung aller Änderungen in kurzen regelmäßigen Zyklen seitens der Entwickler für die Integration
    12. **Test-first programming (Testgetriebene Entwicklung)** Schreiben des Tests vor Realisierung der Funktionalität
    13. **Incremental design (Inkrementelles Design)** Stetige Verbesserung der SW durch inkrementelles Design, in das Feedback und Erkenntnisse einfliessen

Scrum

- Wesentliche Projektmanagementinstrumente und Praktiken:
    - **Sprint**: Scrum gliedert ein Projekt in 2-4-wöchige Iterationen fester Länge
    - **Produkterweiterung**: Jeder Sprint = potentiell auslieferbares, sprintweise erweitertes Produkt
    - **Product Backlog**: Priorisierte Auflistung der geplanten Produkt-Features
    - **Sprint Backlog**: Zusammenzug der für jeden Sprint geplanter Features aus Product Backlog
    - **Definition of Done**: Um abzusichern, dass am Sprint-Ende tatsächlich ein fertiges Produktinkrement vorliegen wird, formuliert das Team zu Beginn eines Inkrementes gemeinsam Kriterien, anhand derer es überprüfen und entscheiden kann, ob die Arbeit an dem Inkrement abgeschlossen ist
    - **Timeboxing**: Nur im Sprintzeitraum machbare Aufgaben werden aufgenommen und wieder entfernt, falls die Aufgabe während dem Sprint nicht fertig gestellt werden kann
    - **Transparenz**: Der Sprint-Status wird täglich (im Stand-Up) aktualisiert und für alle sichtbar abgebildet
- 3 Rollen in Scrum:
    - **Scrum Master**: Coach (nicht Teamleiter!), verantwortlich für die Umsetzung der Scrum Praktiken

- Produkt Owner: Verantwortlich für das Product Backlog, ist Kundenvertreter
  - **Entwicklungsteam**: Entwickelt und Testet das Produkt, selbstorganisiert, kein Teamleiter
- Im Gegensatz zu XP regelt Scrum nicht welche SW-Entwicklungstechniken einzusetzen sind

Kanban

- Allgemeines Ziel: Arbeitsfluss innerhalb Wertschöpfungskette abzubilden & optimieren. Dafür verwendet es 3 Instrumente:
  - **Kanban Board**: Prozessschritte = Spalten, Aufgaben = Tickets, laufen von links nach rechts
  - **Work-In-Progress-Limit**: Limitierung der Menge an gleichzeitig zu erledigenden Aufgaben
  - **Lead Time**: Kanban wird genutzt, um durch die Senkung der durchschnittlichen Bearbeitungszeit den kontinuierlichen Fluss von Aufgaben durch die gesamte Wertschöpfungskette zu optimieren.
- Dieses Vorgehen ist Scrum sehr ähnlich (Visualisierung auf Boards)
- Noch nicht terminierte Aufgaben warten im Backlog und werden erst ins Kanban Board gezogen wenns Platz hat
- Iterationen / Sprints sind in Kanban optional, Kanban Prozess ermöglicht Auslieferung von Arbeitsergebnissen Stück für Stück, daher auch Timeboxing nur optional, im Gegensatz zu Scrum

1.2.2 Kollaborative Erstellung von User-Stories

- Schlechte Qualität von Specs oft Grund für Fehlgeschlagene Projekte
- Mögliche Ursachen:
  - Fehlender Überblick des Kunden
  - Fehlen einer globalen Vision für das System
  - Redundante / sich wiedersprechende Anforderungen
  - Andere Fehler in Kommunikation
- Agile Lösung: User-Stories
  - **Gemeinsame Beschreibung der Anforderungen** aus Sicht der Fachbereichsvertreter, Entwickler, Tester
  - Anstatt wie in sequentiellen Entwicklungen formale Reviews der Anforderungen zu machen, werden im agilen Umfeld durch **häufige informelle Reviews** während der Anforderungsbeschreibungsphase die benötigte Qualität erreicht
  - User-Stories = **funktionale & nicht-funktionale Eigenschaften**
  - Jede Story soll **Abnahmekriterien** für diese Eigenschaften enthalten, welche in Zusammenarbeit zwischen Fachbereichsvertretern, Entwicklern und Testern definiert werden
  - **Blickwinkel des Testers** = Verbesserung der User-Stories, indem er fehlende Details &/ nicht-funktionale Anforderungen ergänzt &/ durch stellen von offenen Fragen über die Story &/ Vorschläge bringt für die Story-Testmethoden und dessen Abnahmekriterien
- Einzusetzende Techniken für die Erstellung der User-Stories: Brainstorming oder Mind Mapping
- 3C Konzept → User-Story = Verbindung von:
  - **Card**: Physisches Medium, beschreibt die User-Story:
    - Anforderung, Dringlichkeit, Erwartete Dauer für Entwicklung und Test, Abnahmekriterien
  - **Conversation**: Erklärt, wie die SW genutzt werden wird
    - Dokumentiert/verbal, Start während Releasephase, Fortführung bei Story-Umsetzungsplanung
  - **Confirmation**: Verwendung der Abnahmekriterien um den Abschluss einer User-Story zu bestätigen
    - Abnahmekriterien können sich über mehrere User-Stories erstrecken
    - Es sollten positive und negative Aspekte getestet werden
    - Getestet wird durch Entwickler/Experten für Performanz, Sicherheit, Interoperabilität etc.
    - Story ist erst abgeschlossen, wenn die definierten Abnahmekriterien getestet und erfüllt sind
- INVEST-Kriterien = Beurteilungsmöglichkeit der Qualität einer User-Story:
  - **Independent** - unabhängig von anderen User-Stories
  - **Negotiable** - verhandelbar, d.h. bietet noch Gestaltungsspielraum fürs Team
  - **Valuable** - wertvoll, d.h. der Nutzen ist erkennbar
  - **Estimable** - so beschrieben und vom Team verstanden, dass sie schätzbar ist
  - **Small** - von angemessener Größe, zu große User-Stories werden heruntergebrochen
  - **Testable** - testbar, z. B. dadurch, dass Akzeptanzkriterien beschrieben sind

1.2.3 Retrospektiven

- Retrospektive = Teamsitzung am Ende jeder Iteration: Was war gut/schlecht? Umsetzung im nächsten Sprint?
- Augenmerk auf die Erhaltung gut funktionierender Praktiken
- Themen: Prozess, beteiligten Personen, Organisationen und Beziehungen, Werkzeuge

FIRMTEC SOLUTIONS AG

- Mögliche Ergebnisse: Verbesserungen für Testeffektivität, Testproduktivität, Testfallqualität, Teamzufriedenheit, Prüfbarkeit der Anwendungen, User-Stories, Features, Systemschnittstellen
- Besser nur einige wenige Verbesserungen vornehmen pro Sprint
  - Einhaltung möglich, kontinuierliche Verbesserung
  - Steigert die Selbstorganisation im Team & kontinuierliche Verbesserung von Entwicklung & Test
- Zeitliche Planung der Retros = f(Agiler Ansatz)
- Teilnehmer: Fachbereichsvertreter & Team (& Moderator)

1.2.4 Continuous Integration (CI)

- Ziel: Am Ende jeder Iteration eine funktionierende SW zu haben
- Grosse Herausforderung
- Lösung: Continuous Integration
  - Regelmässige Zusammenführung aller SW Komponenten, mindestens 1x/Tag (Build erstellen)
  - Konfigurationsmanagement, Kompilierung, Buildprozess, Verteilung in die Zielumgebung, Ausführung der Tests
    → automatisierter, wiederholbarer Prozess
  - Durch diesen kontinuierlichen Buildprozess und autom. Testing → schnellere Fehlerfindung im Code
- Der CI-Prozess besteht aus folgenden automatisierten Aktivitäten:
  - **Statische Codeanalyse**: Durchführung und Aufzeichnung der Ergebnisse
  - **Kompilieren**: Kompilieren und Linken des Codes, Erstellung ausführbarer Dateien
  - **Unittest**: Durchführung, Prüfung der Codeabdeckung, Aufzeichnung der Testergebnisse
  - **Bereitstellung** (Deployment): Installieren der SW in Testumgebung
  - **Integrations- und Systemtests**: Durchführung und Aufzeichnung der Ergebnisse
  - **Bericht** (Dashboard): Veröffentlichung des Status all dieser Aktivitäten an sichtbaren Ort
- Continuous Integration kann auch für Integration eines grossen Systems verwendet werden
- Gute automatisierte Regressionstests decken so viel ab wie möglich, auch die gerade implement. User-Stories
  → Freiraum für manuelle Tests von neuen Features, Änderungen, Fehlernachtests
- Für diese kontinuierliche Qualitätskontrolle: Unterstützende Buildwerkzeuge nötig
- Zusätzliche Tests: weitere statische/dynamische Tests, Performanz(-Profile), Dokumentation aus Quellcode erstellen, Vereinfachung manueller Qualitätssicherungsprozess
- Ziel dieser kont. Q-Kontrolle: Verbesserung der Produktqualität, Reduktion der Lieferzeit
- Traditionelle Alternative dazu: Q-Kontrolle erst nach Fertigstellung
- Verbindung von Build- mit autom. Deployment-Werkzeugen möglich → Überspielung (Deployment) von Builds in Umgebungen: Entwicklung, Test, QS, Produktion
  - Reduktion von Fehler & Verzögerungen (vgl. manuelle Installation)
- **Mögliche Vorteile von CI**:
  - Frühere/einfachere Ursachenanalyse von Integrationsproblemen und widersprüchlichen Änderungen
  - Regelmässiges Feedback ob Code funktioniert oder nicht
  - Die im Test befindliche Version ist max. 1 Tag älter als die Letzte
  - Zügige Fehlernachtests → Verminderung von Regressionsrisiken durch Refactoring des Codes
  - Bestätigung, dass Tagesentwicklungsarbeit solide ist
  - Macht den Fortschritt der Produkterweiterung sichtbar → Ermutigung von Entwickler und Tester
  - Beseitigung der zeitplanerischen Risiken durch Big-Bang Integration
  - Liefert aktuellste Versionen für Test-, Demonstrations- oder Schulungszwecke
  - Vermindert repetitive, manuelle Tests
  - Liefert schnelle Rückmeldung über Verbesserungsentscheidungen
- **Risiken und Herausforderungen von CI**:
  - Einführung und Pflege von **Werkzeugen für CI**
  - Aufsetzung und Etablierung von **Prozessen für CI**
  - Zusätzliche Ressourcen für Testautomatisierung, welche komplex sein kann
  - Möglichst umfassende Testabdeckung ist essenziell, um Vorteile der automat. Tests zu nutzen
  - Oft wird zu sehr auf Unittests vertraut → Weniger Integrations-, System- und Abnahmetests
- CI erfordert versch. Werkzeuge für: Test, Build-Automatisierung, Versionskontrolle

1.2.5 Release- und Iterationsplanung

- Auch für den agilen Lebenszyklus: Planung = fortlaufende Aktivität
- Für agile Lebenszyklen → zwei Arten der Planung: Releaseplanung & Iterationsplanung

Releaseplanung

- Schaut voraus auf Veröffentlichung einer Produktversion die oft einige Monate vom Projektbeginn entfernt ist
- Erstellung & Aktualisierung des Produkt Backlogs

- Hierin: feinere Granulierung von User-Stories
- Liefert Basis für Testvorgehensweise und Planung der Testaktivitäten aller Iterationen
- Releasepläne sind auf grobgranularem Niveau
- Einführung und Priorisierung von User-Stories durch Vertreter des Fachbereichs, mit Team
- Identifikation von Projekt- und Qualitätsrisiken, basierend auf diesen User-Stories
- Grobgranulare Aufwandsschätzung
- Wichtige Beiträge durch Tester an der Releaseplanung:
    - Definieren testbarer User-Stories, inklusive Abnahmekriterien
    - Teilnehmen an Projekt- und Qualitätsrisikoanalyse
    - Schätzen des Testaufwandes in Zusammenhang mit den User-Stories
    - Planen der Tests für das Release

Nach Releaseplanung → Start 1. Iterationsplanung (inkl. Iterationsbacklog)

Iterationsplanung

- Auswahl der priorisierten User-Stories aus Releasebacklog, Detaillierung, Risikoanalyse, Aufwandabschätzung
- Ablehnung durch Team, wenn User-Story zu ungenau ist und Klärungsversuche gescheitert sind → Nächste User-Story
- Fachbereichsvertreter = Verantwortlich
- #User-Stories pro Iteration = f(Velocity [= Mass des Teams für Produktivität im agilen Projekt] & geschätzte User-Story-Grösse)
- Nachdem Iterationsinhalt definiert ist → Herunterbrechung der User-Stories in Aufgaben und Verteilung an Teammitglieder
- Testermehrwert in der Iterationsplanung für die User-Stories:
    - Risikoanalyse, Testbarkeit, Abnahmetestserstellung, Herunterbrechen User-Stories → Aufgaben (v.a. Testaufgaben), Testaufwandschätzung, Identifikation nicht- VS. -funktionale Eigenschaften, Testautomatisierung
- Releaseplanung kann sich im Projektverlauf ändern, auch User-Stories im Product Backlog, durch Interne / Externe Faktoren
    - **Interne** Faktoren: Liefermöglichkeiten, Velocity, Technische Schwierigkeiten
    - **Externe** Faktoren: Ziele &/ Zieldatenveränderung durch: Entdeckung neuer Märkte/Möglichkeiten, neue Mitbewerber, Geschäftsrisiken
- Veränderungen müssen gemäss agilem Prinzip willkommen geheissen werden
  → Angemessene Testbasis und Testorakel für jede Iteration zu Testentwicklungszwecken nötig
  → Vorsichtige Entscheidungen bei Teststrategie und -dokumentation nötig
- Release- und Iterationsplanung → Entwicklungs- und Testaktivitäten:
    - Testumfang & -intensität, Testziele & Gründe dafür
    - Welches Teammitglied führt welche Testaktivität durch?
    - Benötigte Testumgebung und -daten? Zeitpunkt dafür? Änderungen dessen während Projekt?
    - Zeitliche Abfolge, Abhängigkeiten, Vorbed. für Tests (→ Wie häufig Regressionstests, Abhängigkeiten von Features)
    - Zusammenhang/Abhängigkeiten von Entwicklungs- VS. Testaktivitäten
    - Zu adressierende Projekt- & Produktrisiken
- Zusätzlich: Einschluss von Überlegungen größerer Teamschätzungen zu Dauer & Aufwand der Testaktivitäten

2. Grundlegende Prinzipien, Praktiken und Prozesse des agilen Testens

2.1 Die Unterschiede zwischen traditionellen und agilen Ansätzen im Test

- Entwickeln und Testen stehen eng zusammen → Testaktivitäten = f(Entwicklungsmodell)
  → Tester muss Unterschied zwischen traditionellen Lebenszyklusmodellen (sequentiell &/ iterativ) VS. agilen Ansätzen kennen
- Unterschiede Agil VS. Traditionell bezüglich:
    - Art und Weise, wie Test- und Entwicklungsaktivitäten in das Vorgehen integriert werden
    - Arbeitsergebnisse in einem Projekt
    - Verwendete Begrifflichkeiten
    - Testeingangs- und Testendekriterien, die für die verschiedenen Teststufen verwendet werden
    - Gebrauch und Einsatz von Werkzeugen
    - Effektive Umsetzung von unabhängigem Testen
- Implementierung der agilen Ansätze kann je nach Unternehmen/Projekt erheblich anders sein
  → Anpassungsfähigkeit der Tester an solche Veränderungen/Anpassungen sehr wichtig/vorteilhaft

2.1.1 Test- und Entwicklungsaktivitäten

- Ein Hauptunterschied traditioneller VS. agiler Ansätze = sehr kurze Iterationen

- Jede Iteration → funktionierende SW
- Projektanfang → Releaseplanungsphase, danach → Abfolge von Iterationen, inkl. Vorgelagerte It.-Planungen
  - Sobald Iterationsumfang festgelegt → User-Stories: Entwicklung & Integration & Test (E&I&T)
  - Iterationen = hoch dynamisch → (E&I&T) = parallele Aktivitäten → erhebliche Überlappungen
- Entwickler = Unittests, Tester = Integration- & Systemtests, Product Owner = Testen ganzer Stories, während Implementierung, mit schriftlichen TC's oder explorativ → schnelles Feedback an Entwicklung
- **Hardening Iterations** (Stabilisierungsiterationen) = Behebung von Fehlern & Kompromisslösungen
- «Feature Erledigt» = im System integriert und im System getestet
- **Fix Bugs First** = Fehlerbehebung aus vorhergehender Iteration bei Beginn der nächsten Iteration
  - Nachteil: Verschleierung des verbleibenden Aufwandes
- Risikoorientierte Teststrategie
  - Grobe Risikoanalyse schon während Releaseplanung → Tester = Führende Rolle
  - Details zu Qualitätsrisiken während Iterationsplanung → Feature-Reihenfolge & Test-Priorisierung
    - Beeinflusst die Aufwandschätzung der erforderlichen Tests/Feature
- **Pairing** → paarweises Zusammenarbeiten → z.B. 2xTester / 1xEntwickler & 1xTester
  - Schwierig bei räumlicher Trennung, gibt aber Werkzeuge und Vorgehensweisen dafür
- Tester oft auch in der Rolle des Test- und Qualitätstrainers im Team
  - Testautomatisierung oft auf allen Teststufen → Grösserer Teil des Testaufwandes = manuelle Tests: Erfahrungs- resp. Fehlerbasierte Techniken (SW-Angriff, exploratives Testen, Error Guessing)
- Entwickler → Unittests / Tester → automatisierte Integration, System- und Systemintegrationstests → Tester gesucht mit soliden technischen und Testautomatisierungshintergrund
- Agiles Grundprinzip = Änderungen sind willkommen während gesamten Projektverlauf → Leichtgewichtige Dokumentation bevorzugt

2.1.2 Arbeitsergebnisse des Projekts

- Agile Projekte → Optimierung der Dokumentenmenge (→ just enough documentation)
- Prio1 = Funktionierende SW & Automatisierte Tests für Anforderungsnachweis
- Dokugleichgewicht wird angestrebt → Effizienz (wenig Doku) VS. Projektunterstützung (viel Doku)
- Umfeld = reguliert, sicherheitskritisch, dezentral, hochkomplex → weitergehende Formalisierung nötig
  - User-Stories & Abnahmekriterien → formellere Anforderungsbeschreibungen
  - Tracebility-Reports (vertikal und horizontal) → Für Auditoren & Regulierungsvorschriften

1. **Geschäftsprozessorientiert**: Anforderungsspezifikationen (→ User-Stories, inkl. Akzeptanzkrit.) / Benutzerdokumentation
   User-Stories: Sollte so klein sein, dass sie innerhalb eines Sprints (= Iteration) fertig gestellt werden kann
   Epic: Komplexes Feature → Ansammlung von zueinander verbundenen User-Stories:
   a. User-Stories können von untersch. Entwicklerteams kommen, Bsp.: API-Stufe & Benutzerebene
   b. Können über mehrere Iterationen hinweg entwickelt werden
   c. Jede Epic & ihre User-Stories sollten zugehörige Abnahmekriterien haben
2. **Entwicklungsorient**: Datenbank, ER Diagramme, Code, Installations-Skripte, Konfigurationsdateien
   Code: oft begleitet durch autom. Unittests, Erstellt vor (→ Test first / testgetriebene Entw.) oder nach Codeentwicklung
   = Ausführbare Spezifikationen
3. **Testorientiert**: Teststrategien & -pläne / manuelle und automatisierte Tests / Test Dashboards
   Testdokumente: sollen in möglichst leichtgewichtiger Art und Weise erstellt werden
   Testmetriken: Aus Fehlerberichten und Testergebnisprotokollen → auch hier: Leichtgewicht-Ansatz

2.1.3 Teststufen

- **Sequentielle** Lebenszyklusmodelle → oft Endekriterum untere Stufe = Eingangskriterium nächste Stufe
- **Iterative** Modelle → Teststufen oft überlappend
- **Agile** Modelle → Teststufen auch oft überlappend, da Änderungen an Anforderungen, Design, Code jederzeit möglich
  - **Scrum:** Veränderungen an User-Stories nach Iterationsplanung theoretisch unzulässig, in Praxis kommt es vor
  - Reihe an Testaktivitäten während Iteration:
    - **Unittests,** durch Entwickler
    - **Abnahmetests,** für das Feature, teilweise in 2 Aktivitäten aufgeteilt:
      1. Verifizierungstest (automatisiert), Entwickler / Tester, gegen Abnahmekriterien der User-Story
      2. Validierungstest (manuell), Entwickler & Tester & Fachbereich → Feature einsetzbar?

- **Regressionstests**, während gesamter Iteration, = wiederholter Durchlauf der Unit- und Verifikationstests aus laufender und vorangegangenen Iterationen, üblicherweise innerhalb CI-Framework
- **Systemtests**, nicht immer, beginnt sobald die 1. User-Story bereit ist, funktionale und nicht-funktionale Tests, inkl. Performanz, Zuverlässigkeit, Benutzbarkeit, Stabilität u.a.
- **Abnahmetests**, interne Alpha- / externe Beta-Tests, entweder nach (1-x) Iterationen / Integrationsende
- **Benutzerabnahmetests, Betriebsabnahmetests, regulatorische / vertragliche Abnahmetests**
  - Nach (1-x) Iterationen / nach Abschluss aller Iterationen

2.1.4 Werkzeuge zur Verwaltung von Tests und Konfigurationen

- Agil → durch starke Nutzung von automatisierten Werkzeugen für Entwicklung, Tests und Verwaltung geprägt
- Entwickler → autom. Werkzeuge (Programmier- und Test-Frameworks) für statische Analyse (Codeabdeckung) & Unittests, während Entwicklung und nach Codeeincheckung im Konfigurationsmanagementwerkzeug
  → Ermöglicht CI der SW im System → Wiederholung der statischen Analyse & Unittests bei jeder Codeeincheckung
  → Diese autom. Tests können auch funktionale Tests auf Stufe Integration / System umfassen → via GUI Automation
  → In einigen Fällen: Trennung funktionaler Tests von Unittests & weniger häufige Ausführung der funkion. Tests
  → Z.B.: Unittests bei jedem Check-In, langlaufende Tests 1x/Tag (oder Nacht) oder grössere Intervalle
- Falls ein Fehler auftaucht → Behebung des Fehlers vor dem nächsten Check-In → Erfordert Investition in Echtzeit-Testberichte mit guter Detailsicht auf Testergebnisse
  - Hilft teure Zyklen von «Schreiben-Installieren-Fehlschlagen-NeuSchreiben-NeuInstallieren» zu vermeiden
  - Änderungen, die Builds scheitern oder SW-Installation fehlschlagen lassen, werden schneller gefunden
- Automatisierte Tests- und Buildwerkzeuge → Hilfreich für häufige Änderungen, wie sie in agilen Projekten vorkommen
  - Jedoch nicht zu 100% zuverlässig, da Unittests nur beschränkte Effektivität haben! → Autom. Integr. & Sys.-Tests!

2.1.5 Organisationsmöglichkeiten für unabhängiges Testen

- Unabhängige Tester = effektiver bei Fehlersuche → Folgende 3 Organisationsmöglichkeiten:

1. Entwickler erstellen viele der Tests automatisiert, professioneller Tester kann in Team integriert sein und Teile der Testerei übernehmen → Integrierte Tester = höheres Risiko für Unabhängigkeitsverlust / Fehlen objektiver Beurteilung
2. Unabhängige, separate Testteams, **Zuweisung der Aufgaben auf Abruf** während der letzten Sprinttage
   a. **Vorteil**: höhere Unabhängigkeit, Objektivität/Unvoreingenommenheit
   b. **Nachteil**: Tieferes Verständnis der neuen Features, Unstimmigkeiten mit Fachbereichsvertreter/Entwickler
3. Unabhängiges, separates, **langfristiges** Testteam → Unabhängigkeit & Produktverständnis = hoch
   a. Kurzzeitiger Einsatz von Spezialisten möglich, z.B. für Nicht-Iterationsbezogene Arbeiten wie Entwicklung von Testautomatisierungswerkzeugen, Durchführung nicht-funktionaler Tests, Erstellung von Testumgebungen und -daten, Durchführung von sprintunabhängigen Teststufen (z.B. Systemintegrationstests)

2.2 Der Status des Testens in agilen Projekten

- Durch die ständigen Änderungen in agilen Projekten → Teststatus, -fortschritt, Produktqualität ändert ebenfalls
  - Weg für Tester nötig, diese Informationen so geeignet weitergeben, dass daraus kluge Entscheidungen machbar sind
  - Änderungen können auch bestehende Features von früheren Iterationen betreffen
    → **Ständiges Aktuell halten der Tests nötig, um Regressionsrisiko tief zu halten**

2.2.1 Kommunikation über Teststatus, den Fortschritt und die Produktqualität

- Beschreibung von Fortschritt in agilen Teams = funktionierende SW an Iterationsende
- Dafür: Überwachung jedes Arbeitsschrittes in der Iteration und im Release nötig
- Tester in agilen Teams → unterschiedliche Testfortschrittsmethoden: Burndown-Charts, Wiki Dashboards, dashboard-artige Emails, Stand-Up Meeting (verbal), autom. Statusberichte

- Erhebung von Metriken aus Testprozess für Prozess- und Produktverbesserungen
- **Agile Task Boards** inkl.: Story Cards, Entwicklungsaufgaben, Testaufgaben (oft mit farblich abgestimmten Karten)
  - Mögliche Kolonnen: Zu erledigen / In Bearbeitung / Zu prüfen / Erledigt
  - Oft auch automatisierte Kartenverwaltung im Einsatz
  - Testaufgaben → Bezug auf Abnahmekriterien, definiert in User-Stories
  - Überwachung oft beim täglichen Stand-Up → Bei Stockung von Kärtchen wird der Grund gesucht und gelöst
- **Daily Stand-Up Meeting** bezieht alle Mitglieder des agilen Teams ein, auch Tester
  - Alle sagen ihren aktuellen Status, nach folgender Agenda:
    - Was hast Du seit dem letzten Stand-Up Meeting **abgeschlossen**?
    - Was **planst** Du bis zum nächsten Stand-Up Meeting abzuschließen?
    - Was steht Dir **im Weg**?
  - Alle Probleme, die den Testfortschritt behindern könnten, werden angesprochen und gelöst
- Kundenzufriedenheitsbefragungen → hilft bei Verbesserung der Produktqualität
- Mögliche Metriken: Tests erfolgreich / fehlerhaft, Fehleraufdeckungsrate, Ergebnisse von Bestätigungs- und Regressionstests, Fehlerdichte, Fehler gefunden / behoben, Anforderungsabdeckung, Risikoabdeckung, Codeüberdeckung und -veränderung

2.2.2 Das Regressionsrisiko trotz zunehmender Zahl manueller und automatisierter Testfälle beherrschen

- Agil → Produkt wächst mit jeder Iteration → Testumfang erhöht sich
- Neben den aktuellen Änderungen muss sichergestellt werden, dass auch die älteren Features noch laufen
- Risiko für Verschlechterung von alten Features ist im agilen Umfeld gross, da meist umfangreiche Änderungen pro Iteration

  → Testautomation, so früh wie möglich, auf allen Teststufen & Aktuell halten aller Tests (→ <u>Konfigurationswerkzeug</u>)
- Testfallüberprüfung und Anpassung bei jeder Iteration (→ gibt Freiraum für Tests von neuen Funktionen/Änderungen):
  - Feature-Veränderungen → irrelevante Tests entfernen/anpassen
  - Testfälle auf Eignung zur Automatisierung prüfen
- Prozess für Testfallentwurf soll schon während Releaseplanung erfolgen, gute Vorgehensweisen für Entwurf & Implementierung früh anpassen und durchgängig verwenden
- Kurze Zeitintervalle & permanente Änderungen = Verschärfte Auswirkungen auf Testentwurfs- und Implementierungspraktiken
- Gut geschriebene automatisierte Tests → lebendes Dokument der Systemfunktionalität
- Durch Check-In resp. Commit der automatisierten Tests & deren Ergebnisse in Konfigurationsmanagement inkl. Versionierung

  = Abruf der getesteten Funktionalität & Testergebnisse für jeden Build jederzeit möglich
- **Unittests müssen durchgeführt werden bevor der Quellcode in Baseline (=Ausgangspunkt) eingefügt wird** → Sicherstellung, dass SW-Build nicht beschädigt wird → Bei Failed Unittests: entsprechender Code nicht hinzufügen! Erst wenn alles passed!
- **Automatisierte Unittests = Feedback** zur Code- und Buildqualität, aber **nicht zu Produktqualität**
- **Automatisierte Abnahmetests** laufen regelmässig (mind. 1x/Tag) als Teil von CI bei der Erstellung des vollständigen Systems, aber nicht bei jedem Code-Check-In, da diese Tests länger laufen und Check-In verlangsamen können
  - Abnahmetests liefern Feedback zur Produktqualität in Bezug auf Veränderungen **seit letztem Build**
- **Systemtests** (→ Build-Verifizierungstests) = kontinuierlich, unmittelbar nach Lieferung eines neuen Builds in Testumgebung

  → Durchführung von Mindestbestand an automatisierten Tests zur Abdeckung kritischer Systemfunktionalitäten & Integrationspunkte
- Automatisierte Test im **Regressionstestset** → läuft i.d.R. beim täglichem Haupt-Build in CI Umgebung

  → Falls Fail → Arbeitsunterbruch & Ursachenanalyse: Gewollte Funktionsänderung? → User-Story Update = Abdeckung neuer Abnahmekriterien, ev. alter Test entfernen, da neuer Test vorhanden. **Echter Fail?** → Fehlerbehebung nötig durch Team
- Neben automatisierter Testdurchführung, auch Automation folgender Testaufgaben:
  - Testdatenerstellung, Systembeladung durch Testdaten, Buildauslieferung in Testumgebungen (→ CI), Zurücksetzung der Testumgebung auf Grundzustand, Vergleich von Datenergebnissen

2.3 Rollen und Fähigkeiten eines Testers in einem agilen Team

2.3.1 Fähigkeiten agiler Tester (zus. zu Fähigkeiten definiert im Foundation Level Sillabus)

Benötigte Hard-Skills:

- Kenntnisse in der **Testautomatisierung**
- Vertraut mit Test- (En: TDD, **T**est **D**riven **D**evelopment) und Abnahmegetriebener (En: A-TDD) Entwicklung
- Sicher im Umgang mit White-Box, Black-Box und erfahrungsbasierten Tests sein
- Fähigkeiten besitzen "just enough" zu dokumentieren

Benötigte Soft-Skills:

- Positiv und **lösungsorientiert** gegenüber Teammitgliedern und anderen Mitarbeitern außerhalb des Teams auftreten
- Eine eher kritische, qualitätsorientierte, **skeptische Denkweise** über das Produkt an den Tag legen
- **Informationen aktiv** vom Fachbereich / Auftraggeber einholen (statt nur auf geschriebene Spezifikationen zu verlassen)
- Testergebnisse, Testfortschritte und **Produktqualität** genau beurteilen und darüber berichten
- Zusammen mit Kunden bzw. dem Fachbereich effektiv an Definition **prüfbarer User-Stories** (Abnahmekriterien!) arbeiten
- Im Team **mitarbeiten**, paarweise mit Programmierern und anderen Teammitgliedern
- **Schnell auf Veränderungen reagieren** → Testfälle anpassen / hinzufügen / verbessern
- Ihre eigene Arbeit planen und organisieren
  → Kontinuierliche **Weiterentwicklung persönlicher Fähigkeiten** in Agilen Umfeld besonders wichtig!

2.3.2 Die Rolle eines Testers in einem agilen Team

- Erheben & Bereitstellen von Informationen zu: **Teststatus**, **Testfortschritt**, **Produktqualität**, **Prozessqualität**, zudem:
  - Die **Teststrategie** verstehen, implementieren und aktualisieren
  - Die **Testüberdeckung** über alle anzuwendenden Metriken hinweg messen und berichten
  - Den richtigen Einsatz der **Testwerkzeuge** sicherstellen
  - **Testumgebungen** sowie **Testdaten** konfigurieren, verwenden und verwalten
  - **Fehlerberichte** erstellen und mit dem Team bei der Fehlerbehebung zusammenarbeiten
  - Teammitglieder in die wesentlichen **Prinzipien des Testens** einweisen
  - Sicherstellen, dass die **Tests angemessen** sind & in Release- und Iterationsplanung berücksichtigt werden
  - Mit Entwicklern, Fachbereich, Product Owner → Klärung der Anforderungen in Bezug auf:
    - Testbarkeit, Konsistenz, Vollständigkeit
  - An Team **Retrospektiven** proaktiv teilzunehmen und dort Verbesserungen mit vorschlagen und umsetzen
- In agilen Teams → Jedes Teammitglied verantwortlich für Produktqualität → Jeder hat testbezogene Aufgaben
- Organisationsbezogene Risiken:
  - Tester arbeiten so eng mit Entwicklern zusammen, dass sie ihre **objektive Sicht verlieren**
  - **Tester tolerieren** ineffiziente, ineffektive oder qualitativ schwache Praktiken innerhalb des Teams
  - Tester können mit den **permanenten Änderungen** bei gleichzeitig kurzen Iterationszyklen nicht mithalten

  → Organisationen sollen Alternativen zur Wahrung der Unabhängigkeit in Betracht ziehen

3. Methoden, Techniken und Werkzeuge des agilen Testens

3.1 Agile Testmethoden

- Agil oder nicht, folgende Techniken für Qualitätssicherung sind generell gültig:
  - Vorab Schreiben von Tests (um richtiges Verhalten auszudrücken)
  - Konzentration auf frühe Fehlervermeidung und -behebung
  - Sicherstellen der Durchführung der richtigen Testarten zur richtigen Zeit / auf der richtigen Teststufe

3.1.1 Testgetriebene-, Abnahmetestgetriebene- und Verhaltensgetriebene Entwicklung

- 3 sich gegenseitig **ergänzende Techniken** → in Agilen Teams genutzt um Tests auf verschiedenen Stufen durchzuführen
- Jede Technik = Beispiel für grundlegendes Prinzip des Testens mit **Test- bevor Codeentwicklung**

Testgetriebene Entwicklung (Test-Driven Development, TDD)

- Code wird mit automatisierten Testfällen erstellt, nach folgendem Prozess:
  - Testfall **erstellen** und automatisieren für zu schreibenden Codeabschnitt
  - Testfall **ausführen** → Fehlschlag erwartet, da Code noch nicht existiert

- Testfall so lange **wiederholen** bis Code passt
- Bei **Codeänderungen** (=Refactoring) → Retest → Sollte auch erfolgreich sein

- Überwiegend Unittests & Codebezogen, kann aber auch auf Integrations- und Systemstufe gemacht werden
- Testgetriebene Entwicklung wurde durch Extreme Programming populär
- Entwickler konzentrieren sich auf klar definierte, erwartete Ergebnisse, Tests automatisiert, in CI verwendet

Abnahmetestgetriebene Entwicklung

- Definiert Abnahmekriterien und Tests während der Entwicklung von User-Stories
- Kollaborativer Ansatz, ermöglicht jedem Stakeholder zu verstehen, wie die SW sich zu verhalten hat und was Entwickler, Tester und Fachbereichsvertreter dafür tun müssen
- Es werden wiederverwendbare Regressionstests erstellt, unterstützt durch spezifische Werkzeuge, oft innerhalb CI
- Tests Können mit Daten und Serviceebenen der Anwendung verbunden sein → Tests auf System- oder Abnahmeniveau möglich
- Ermöglicht schnelle Lösung von Fehlern und Bewertung des Verhaltens des Features

Verhaltensgetriebene Entwicklung (Behaviour Driven Development, BDD)

- Ermöglicht es dem Entwickler sich auf das Testen des Codes auf das erwartete Verhalten hin zu konzentrieren
- Tests basieren auf dem erwarteten Verhalten der SW → leichter zu verstehen für Teammitglieder & Stakeholder
- Spezifischen BDD Frameworks → Definition der Abnahmekriterien auf folgender Basis (→ Gherkin-Format):
    - **Gegeben** ein Einstiegskontext,
    - **Wenn** ein Ereignis auftritt,
    - **Dann** werden bestimmte Wirkungen sichergestellt.
- Daraus erstellt das Framework ausführbaren Testcode → BDD unterstützt/hilft bei Testautomatisierung
- Diese sind auch für Personen ohne Programmierkenntnisse lesbar und verständlich

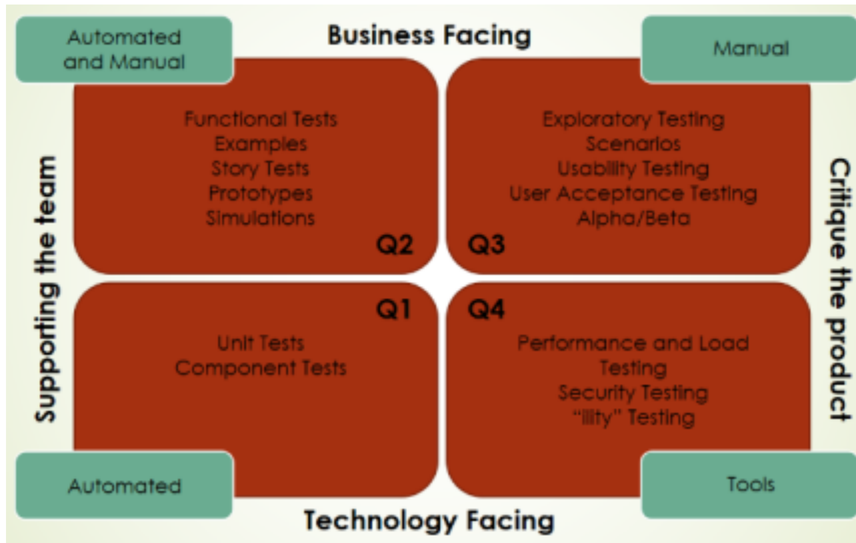
FIRMTEC SOLUTIONS AG

3.1.2 Die Testpyramide (Prinzip: frühestmögliche Qualitätssicherung)

Komponente → Integration → System → Abnahme (= Teststufen)

- Viele Tests → Weniger Tests
- API-Automatisiert → GUI-Automatisiert (& Manuell)

3.1.3 Testquadranten, Teststufen und Testarten (haupts. für dynamische Tests)



3.1.4 Die Rolle des Testers

Teamwork - Best Pratices:

- **Funktionsübergreifend**: Jedes Teammitglied trägt mit seinen speziellen Fähigkeiten zum Team bei. Teamarbeit bei Teststrategie, Testplanung, Testspezifikation, Testdurchführung, Testbewertung, Testergebnisberichten

- **Selbstorganisierend**: Das Team bestimmt selbstständig den nächsten anzugehenden Schritt aus Product Backlog
- **Ortsverbundenheit**: Tester sitzen mit den Entwicklern und dem Product Owner zusammen
- **Zusammenarbeit**: zwischen: Tester, Teammitgliedern, andere Teams, Stakeholdern, Product Owner, Scrum Master
- **Bevollmächtigt**: Gemeinsame (Wer alles? siehe Punkt oben) technische Entscheidungen bezüglich Design und Test
- **Engagiert**: Der Tester ist engagiert im Hinterfragen und Bewerten des Produktverhaltens und der Produktcharakteristika, in Bezug auf die Erwartungen und Bedürfnisse der Kunden und Nutzer
- **Transparent**: Der Vorgang des Programmierens und Testens ist auf dem agilen Task Board sichtbar
- **Glaubwürdig**: Der Tester muss die Glaubwürdigkeit der Teststrategie, seine Implementierung und Ausführung sicherstellen, da die Stakeholder ansonsten den Testergebnissen nicht trauen werden. Oft geschieht dies durch regelmäßige Berichte über den Testprozess an die Stakeholder
- **Offen für Rückmeldungen**: Rückmeldungen sind ein wichtiger Aspekt für den Erfolg jedes Projekts, insbesondere aber in agilen Projekten. Retrospektiven ermöglichen es den Teams aus Erfolgen und Misserfolgen zu lernen
- **Flexibel**: Tester müssen auf Veränderungen reagieren können

Sprint Null – Erste Iteration im Projekt → Vorbereitungen treffen, Arbeiten an folgenden Massnahmen:

- Identifikation des **Projektumfangs** (→ Product Backlog)
- Erstellen einer initialen **Systemarchitektur** und ggf. erster Prototypen
- Planung, Erwerb und Installation der notwendigen **Werkzeuge**
- Erstellung initiale **Teststrategie** aller Teststufen, u.a.: Testumfang, -arten, technische Risiken, Überdeckungsziele
- Durchführung einer initialen **Qualitätsrisikoanalyse**
- Definition von **Metriken** zur Messung des Testfortschritts und zur Produktqualität
- Spezifikation der **Definition of Done**
- Festlegen der **Struktur des Task Boards** und initiales "befüllen" des Boards
- Definition von **Testende-Zeitpunkt**, bevor das System an den Kunden geliefert wird

Integration → kontinuierlicher Mehrwert → Identifikation der Abhängigkeiten zw. Funktionen und Features

Testplanung → Testen vollständig in Team integriert → Testplanung bereits während Releaseplanung

- Sprintplanung → Ergebnis = Reihe von Aufgaben (<= 2 Tage) in Taskboard → Testprobleme nachverfolgen

Agile Praktiken → Folgende Praktiken sind von Nutzen:

- **Pairing**: Zwei Teammitglieder bearbeiten gemeinsam Test- oder andere Sprintaufgabe
- **Inkrementelles Test Design**: Testfälle & Chartas schrittweise aus User-Stories und anderen Testgrundlagen aufbauen, dabei mit einfachen Tests starten und dann übergehen zu komplexeren Tests
- **Mind Mapping** → nützliches Werkzeug für das Testen, Bsp.: Um zu bestimmen welche Testsitzungen durchzuführen sind, Teststrategien demonstrieren, Testdaten beschreiben

3.2 Qualitätsrisiken beurteilen und Testaufwände schätzen

- Typisches Testziel → Risiko von Produktqualitätsproblemen vor Inbetriebnahme auf akzeptables Niveau reduzieren
- In Agilen Projekten → Gleiche Techniken wie bei traditionellen Projekten nutzbar für Qualitätsrisikoreduktion:
  - Identifizieren, Risikoniveau abschätzen, Aufwand für Risikoreduktion abschätzen, Milderung des Risikos durch Testentwurf, Implementierung und Durchführung
  - In einigen Bereichen im Agilen Umfeld müssen Techniken angepasst werden, um den kurzen Iterationen & Grad an Veränderungen gerecht zu werden

3.2.1 Die Produktqualitätsrisiken in agilen Projekten einschätzen

- Risiko = Möglichkeit eines negativen Ergebnisses
- Risikoniveau → Einschätzung von Eintrittswahrscheinlichkeit & Wirkung des Risikos
- Betrifft Hauptwirkung die Produktqualität? → Qualitäts- (Produkt-) Risiko
- Betrifft erster Effekt des potentiellen Problems den Projekterfolg? → Projekt- (Planungs-) Risiko
- In Agilen Projekten → Qualitätsrisikoanalyse an zwei Stellen:

FIRMTEC SOLUTIONS AG

    - **Releaseplanung** → Fachbereichsvertreter liefern groben Überblick über Risiken, gesamtes Team kann zur Risikoidentifikation und -beurteilung beitragen
    - **Iterationsplanung** → Gesamtes Team identifiziert und bewertet Qualitätsrisiken
- Beispiele für Qualitätsrisiken eines Systems:
    - Falsche Berechnungen in Berichten → funktionales Risiko, betrifft Genauigkeit
    - Langsame Reaktion auf Nutzerangaben → nicht-funktionales Risiko, betrifft Effizienz und Antwortzeit
    - Schwierigkeit, Bildschirme und Felder zu verstehen → nicht-funktionales Risiko, betrifft Benutzbar- & Verständlichkeit
- Beispiel - Prozess der Qualitätsrisikoanalyse in Agilem Projekt während Iterationsplanung:
    1. Besprechung aller Teammitglieder, einschließlich der Tester
    2. Auflistung aller Backlog Themen für aktuelle Iteration (z. B. auf einem Whiteboard)
    3. Identifikation der Qualitätsrisiken für jedes Thema unter Berücksichtigung aller produktrelevanter Qualitätsmerkmale
    4. Beurteilung jedes identifizierten Risikos (→ Kategorie & Risikostufe = f [Wirkung & Wahrscheinlichkeit])
    5. Bestimmen des Ausmaßes an Tests in Abhängigkeit von der Risikostufe
    6. Auswahl der passenden Testtechniken für Risikoverminderung. Grundlage: Risiko, Risikostufe, Qualitätsmerkmale
- Risiken können über Projektverlauf ändern (agiles Grundprinzip!) & Risikos können auch vor Teststart vermindert werden (→ statische Tests)

3.2.2 Schätzung des Testaufwands auf Basis des Inhalts und des Risikos

- **Planungspoker** → Product Owner / Kunde lesen User-Story vor, jeder Schätzer hat einen Satz Karten mit Werten, z.B. mit Fibonacci-Sequenz (0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, … Vorteil: Unsicherheit wächst proportional mit Grösse der Story), hohe Schätzung → Story zu gross & Aufteilung nötig, Schätzer diskutieren das Feature und klären Fragen (Aspekte: Entwicklungs- und Testaufwand, Komplexität, Testumfang (→ Risikostufe), Feature vollständig besprochen → Jeder Schätzer wählt geheim eine Karte aus, alle Karten werden gleichzeitig offen gelegt, sind alle Karten gleich → offizielle Schätzung, Falls ungleich: Diskussion, & Wiederholung der Pokerrunde bis Einigung erreicht (→ Konsens / Anwendung bestimmter Regeln [Median, höchster Wert]) zur Begrenzung #Pokerrunden → Verlässliche Schätzung des Aufwandes

3.3 Techniken in agilen Projekten

3.3.1 Abnahmekriterien, angemessene Überdeckung und andere Informationen für das Testen

- Benutzbarkeit & Performanz → Auch als User-Stories definieren
- User-Stories = Wichtige Testbasis! Andere Testbasis-Informationsquellen:
    - Erfahrung aus aktuellen oder vorangegangenen Projekten
    - Bestehende Funktionen, Features und Qualitätsmerkmale des Systems
    - Code, Architektur und Entwurf
    - Nutzerprofile (Kontext, Systemkonfiguration und Nutzerverhalten)
    - Informationen zu Fehlern aus aktuellen und vorangegangenen Projekten
    - Eine Kategorisierung der Fehler in einer Fehlertaxonomie
    - Anwendbare Standards (z. B. [DO-178B] für Avionik Software)
    - Qualitätsrisiken (siehe Abschnitt 3.2.1)
- Themen die abzudecken sind für Abnahmekriterien:
    - **Funktionales Verhalten**: Das von außen zu beobachtende Verhalten mit Nutzeraktionen als Input
    - **Qualitätsmerkmale**: Wie das System das spezifische Verhalten ausführt. Die Eigenschaften werden häufig auch als Qualitätsattribute oder nicht-funktionale Anforderungen bezeichnet. Verbreitete Qualitätsmerkmale sind Performanz, Verlässlichkeit, Benutzbarkeit, usw. (siehe auch [ISO25000])
    - **Szenarios** (Use Cases): Eine Abfolge von Aktionen zwischen einem externen Akteur (oft ein Nutzer) und dem System, um ein bestimmtes Ziel zu erreichen oder eine bestimmte Aufgabe zu erfüllen
    - **Geschäftsprozessregeln**: Aktivitäten, die nur unter bestimmten Bedingungen im System ausgeführt werden können, wobei die Bedingungen durch externe Vorgehensweisen und Beschränkungen bestimmt sind. (Bsp.: Verfahrensweise die von einem Versicherungsunternehmen verwendet wird, um Versicherungsansprüche zu behandeln)
    - **Externe Schnittstellen**: Beschreibungen der Verbindungen zwischen dem zu entwickelnden System und der Außenwelt. Können in verschiedene Arten unterteilt werden (Benutzerschnittstelle, Programmierschnittstelle, etc.)
    - **Einschränkungen**: Jegliche Entwurf- und Implementierungsbedingungen, die die Möglichkeiten der Entwickler begrenzen. Geräte mit eingebetteter SW → Berücksichtigung von physischen Grenzen (Größe, Gewicht, Schnittstellen)

    - **Datendefinitionen**: Der Kunde kann das Format, den Datentyp, erlaubte Werte und Standardwerte sowie die Anordnung des Datensatzes in einer komplexen Unternehmensdatenstruktur beschreiben (z. B. PLZ in US-Adressen)
- Weitere Informationen zu User-Stories & Abnahmekriterien für Tester:
    - Wie das System arbeiten und genutzt werden soll
    - Systemschnittstellen, die genutzt werden können/die zugänglich sind, um das System zu testen
    - ob die aktuell verfügbare Werkzeugunterstützung ausreicht
    - ob der Tester über genügend Wissen und Fähigkeiten verfügt, um die notwendigen Tests durchzuführen

Definition of Done (DoD)
- **Unittests**:
    - 100% Entscheidungsüberdeckung wo dies möglich ist und mit Reviews der nicht abgedeckten Wege
    - Statische Analyse über den gesamten Code
    - Keine ungelösten schweren Fehler (Rangfolge gemäß Risiko und Schwere)
    - Keine bekannten inakzeptablen technischen Schulden im Entwurf oder im Code
    - Reviews des gesamten Codes, der Unittests und der Ergebnisse der Unittests sind abgeschlossen
    - Alle Unittests sind automatisiert und vollständig durchgeführt
    - Wichtige Qualitätsmerkmale befinden sich innerhalb der vereinbarten Grenzen (z. B. Performanz)
- **Integrationstests**:
    - Alle funktionalen Anforderungen sind getestet, **inklusive Positiv- und Negativtests**. Die Anzahl der Tests basiert dabei auf der Größe, Komplexität und der Kritikalität der Applikation
    - Alle **Schnittstellen** zwischen den Komponenten sind getestet
    - Alle **Qualitätsrisiken** sind gemäß vereinbarten Ausmaß an Tests abgedeckt
    - Keine ungelösten schweren Fehler (priorisiert gemäß Risiko und Bedeutung)
    - Alle gefundenen Fehler sind **dokumentiert**
    - Alle **Regressionstests** sind soweit möglich automatisiert und in gemeinsamer Ablage gespeichert
- **Systemtests:**
    - End-to-End Tests der User-Stories, Features und Funktionen sind durchgeführt und dokumentiert
    - Alle Nutzeridentitäten sind abgedeckt
    - Die wichtigsten Qualitätsmerkmale des Systems abgedeckt (z. B. Performanz, Robust- und Zuverlässigkeit)
    - Tests werden in einer produktionsähnlichen Umgebung durchgeführt
    - Alle Qualitätsrisiken sind gemäß vereinbarten Testumfang abgedeckt
    - Alle Regressionstests sind soweit möglich automatisiert und in gemeinsamer Ablage gespeichert
    - Alle offenen Fehler sind dokumentiert, priorisiert (gemäß Risiko und Bedeutung) und im aktuellen Status akzeptiert
- **User-Story (US):**
    - Die für Iteration ausgew. US sind: vollständig, vom Team verstanden, haben detaillierte, testbare Abnahmekriterien
    - Alle Elemente der User-Story sind spezifiziert und einem Review unterzogen worden
    - Die Abnahmetests der User-Stories sind (als Testbeschreibung und/oder Testscript) bereitgestellt
    - Die notwendigen Aufgaben für Implementierung & Testen sind identifiziert und vom Team geschätzt worden
- **Feature (kann mehrere User-Stories &/ Epics umfassen):**
    - Jede wesentliche User-Story und ihre Abnahmekriterien sind vom Kunden definiert und abgenommen
    - Der Entwurf ist vollständig
    - Der Code ist vollständig
    - Unittests wurden durchgeführt und haben den definierten Überdeckungsgrad erreicht
    - Integrations- und Systemtests für das Feature wurden anhand der definierten Überdeckungskriterien durchgeführt
    - Alle schweren Fehler sind korrigiert
    - Vollständige Feature-Dokumentation: Release Notes, Bedienungsanleitungen, Online Hilfe-Funktionen
- **Iteration:**
    - Alle Features der Iteration sind fertig entwickelt und gemäß der Feature-Level Kriterien individuell getestet
    - Nicht-kritische Fehler (→ nicht behoben während Iteration) → dem Product Backlog hinzugefügt und priorisiert
    - Die Integration aller Features der Iteration ist erfolgt und getestet
    - Die Dokumentation ist geschrieben und nach dem Review abgenommen
- **Release:**

- **Überdeckung**: Alle relevanten Testbasiselemente für den gesamten Inhalt des Release sind durch Tests abgedeckt. Die Angemessenheit der Überdeckung wird dadurch bestimmt, was neu oder geändert ist, sowie durch die Komplexität, Größe und das damit verbundene Risiko für einen Misserfolg
    - **Qualität**: Die Fehlerhäufung (z. B. wie viele Fehler werden pro Tag oder pro Transaktion gefunden), die Fehlerdichte (z. B. die Anzahl der gefundenen Fehler im Vergleich zur Anzahl der User-Stories und/oder Qualitätsattribute), die geschätzte Anzahl verbleibender Fehler bewegt sich in einem akzeptablen Rahmen, die Folgen der nicht behobenen und verbleibenden Fehler (z. B. die Schwere und Priorität) sind verstanden und akzeptabel, das Restrisiko, das mit jedem identifizierten Qualitätsrisiko verbunden ist, ist verstanden und akzeptabel
    - **Zeit**: Wenn das zuvor festgelegte Lieferdatum erreicht ist, müssen die geschäftlichen Belange bezüglich der Veröffentlichung oder Nicht-Veröffentlichung abgewägt werden
    - **Kosten** – Die geschätzten Lebenszykluskosten sollten verwendet werden, um die Rendite des gelieferten Systems zu berechnen (d.h. die berechneten Entwicklungs- und Wartungskosten sollten weit geringer sein als der erwartete Gesamtumsatz des Produkts). Der Hauptteil der Lebenszykluskosten entsteht oftmals durch die Wartung des Produkts nach dem Release, da eine gewisse Anzahl von Fehlern unbemerkt in die Produktion übernommen wird

3.3.2 Anwendung der abnahmetestgetriebenen Entwicklung

- Test-First-Ansatz → Testfälle werden vor Implementierung erstellt: Vom gesamten Team, manuell / automatisiert
- 1. User-Story wird analysiert, diskutiert, geschrieben → Korrektur jeglicher Unvollständigkeit, Mehrdeutigkeit, Fehler
- 2. Tests werden erstellt, gemeinsam im Team / vom Tester allein → Beurteilung der Tests durch Fachbereichsvertreter
  Tests = Beispiele, welche spezifischen Eigenschaften der User-Story beschreiben → Hilft Team, US korrekt zu implementieren
- I.d.R.: die ersten Tests = **Positivtests** → Testet das Standardverhalten, ohne Ausnahme- oder Fehlerbedingungen
- Umfasst die Abfolge der Aktivitäten die ausgeführt werden, wenn alles nach Plan läuft
- Nach den Positivtests → **Negativtests**, welche auch nicht-funktionale Attribute abdecken (Performanz, Benutzbarkeit)
- **Test so ausdrücken, dass jeder Stakeholder sie versteht** (Vorbedingungen, Inputs, Ergebnisse → in normaler Sprache)
- Tests/Beispiele → Abdeckung aller Eigenschaften der US → **nichts hinzufügen, das nicht in US drin ist!**
- Keine 2 Tests/Beispiele für das gleiche Merkmal der US

3.3.3 Funktionales und Nicht-Funktionales Black-Box Test Design

- Agile Projekte → Viele Tests von Testern entwickelt während Entwickler programmieren, **beide auf Basis von User-Stories**
- Nicht-Funktionale Anforderungen auch als/innerhalb User-Stories beschreiben
- Black-Box Testentwurfsverfahren: Äquivalenzklassenbildung, Grenzwertanalyse, Entscheidungstabellen, zustandsbasierte Tests

3.3.4 Exploratives Testen und agiles Testen

- In Agilen Projekten wichtig wegen begrenzter Zeit für Testanalyse & begrenzter Detailgenauigkeit der User-Stories
- Für beste Testergebnisse → **Kombination von Explorativem Testen mit anderen erfahrungsbasierten Verfahren**:
    - Analytisches (Risiko- oder Anforderungsbasiertes) Testen, modellbasiertes Testen, Regressionsvermeidendes Testen
- Exploratives Testen → Testentwurf & -durchführung gleichzeitig, unterstützt durch Test-Charta (→ Liefert Testbedingungen, welche während zeitlich begrenzter Testsitzung abgedeckt werden müssen)
    - Auf Basis der Ergebnisse des letzten Tests werden nachfolgende Testfälle entworfen
    - Testentwurf = Black- or White-Box Verfahren verwendbar
- **Test-Charta** kann folgende Informationen enthalten:
    - **Akteur**: der vorgesehene Nutzer des Systems
    - **Zweck**: Angabe von Testziel und Testbedingungen
    - **Set-up**: Was muss vorhanden sein, um die Testdurchführung zu beginnen

- **Priorität**: der relative Stellenwert dieser Charta, auf Grundlage der Priorität der zugehörigen User-Story oder Risikostufe
- **Referenz**: Spezifikationen (z. B. User-Story), Risiken, oder andere Informationsquellen
- **Daten**: Jegliche Daten, die benötigt werden, um die Charta durchzuführen
- **Aktivitäten**: Liste von Ideen, was Tester tun will (z. B. als „Super User" ins System einloggen) und was für Tests interessant wären (sowohl Positiv- als auch Negativtests)
- **Orakel Notizen**: Wie soll das Produkt beurteilt werden, um zu bestimmen, was korrekte Ergebnisse sind (z. B. um zu erfassen, was auf dem Bildschirm passiert und dies mit dem zu vergleichen, was im Nutzerhandbuch steht)
- **Variationen**: alternative Aktionen & Auswertungen, um die Ideen zu ergänzen, die unter Aktivitäten beschrieben sind

- Sitzungsbasiertes Testmanagement (session based testing) → Eine Sitzung = ununterbrochene Zeitspanne von 60-120 min:
    - Überblickssitzung (um zu lernen, wie es funktioniert)
    - Analysesitzung (Bewertung der Funktionalität oder Eigenschaften)
    - Genaue Überdeckung (Ausnahmefälle, Szenarien, Interaktionen)
- Relevante Fragen des Testers entscheidend für Testqualität, Bsp.:
    - Was ist das Wichtigste, das über das System herauszufinden ist?
    - Auf welche Art und Weise kann das System versagen?
    - Was passiert, wenn…? / Was sollte passieren, wenn…?
    - Werden die Bedürfnisse, Anforderungen und Erwartungen des Kunden erfüllt?
    - Ist das System installationsfähig (und wenn nötig deinstallationsfähig) für alle unterstützten Upgrades?
- Bei Testdurchführung nutzt Tester: Kreativität, Intuition, Erkenntnisvermögen, Fachkenntnisse für Fehlerfindung
- Tester braucht: gutes SW Verständnis unter Testbedingungen, Kenntnisse über Fachbereich, wie SW genutzt wird, wie zu bestimmen ist wann SW fehlschlägt
- Reihe von nutzbarer Heuristiken (Anleitung für Tester für Testdurchführung & Bewertung der Ergebnisse), Bsp.:
    - Grenzen
    - CRUD (Create, Read, Update, Delete = Erstellen, Lesen, Aktualisieren, Löschen)
    - Konfigurationsvariation
    - Unterbrechungen (z.B. Abmelden, Schliessen / Neustarten)
- Wichtig das Tester Prozess so genau wie möglich dokumentiert, Bsp.:
    - **Testüberdeckung**: Eingabewerte? Wieviel wurde abgedeckt? Wieviel muss noch getestet werden?
    - **Bewertungsnotizen**: Beobachtungen während des Testens, sind das System und das getestete Feature stabil, wurden Fehler gefunden, was ist als nächster Schritt als Folge der aktuellen Beobachtungen geplant und gibt es weitere Ideen?
    - **Risiko-/Strategieliste**: Welche Risiken wurden abgedeckt / welche verbleiben von den wichtigsten? Wird die ursprüngliche Strategie weiterverfolgt, sind Änderungen nötig?
    - **Probleme, Fragen und Anomalien**: Jegliches unerwartetes Verhalten, jegliche Fragen bezüglich der Effizienz des Ansatzes, jegliche Bedenken bezüglich der Ideen/Testversuche, Testumgebung, Testdaten, Missverständnisse bezüglich der Funktion, des Testskripts oder des Systems unter Testbedingungen
    - **Tatsächliches Verhalten**: Aufzeichnung des Systemverhaltens, (z. B. Video, Screenshots, Ergebnisdateien)
- Aufgezeichnete Infos → in Statusmanagementwerkzeug erfassen (Stakeholdergerecht → aktueller Status leicht ermittelbar)

3.4 Werkzeuge in agilen Projekten

- Werkzeuge: Anforderungs-, Test- und Abweichungsmanagem. → agil: Anwendungslebenszyklus-, oder Aufgabenmanagem.
    - Task Boards, Burndown Charts, User-Stories
    - Wichtig für Agile Tester: **Konfigurationsmanagementwerkzeug** → wegen vielen Autom. Tests auf allen Stufen

3.4.1 Aufgabenmanagement- und Nachverfolgungswerkzeuge (→ Elektronische Task Boards) dienen den folgenden Zwecken:

- Aufzeichnung der **Stories**, ihrer relevanten Entwicklungs- und Testaufgaben, um sicherzustellen, dass während des Sprints nichts verloren geht
- Erfassen der **Aufwandschätzungen** der Teammitglieder und automatische Berechnung des notwendigen Gesamtaufwands für die Implementation der Story, um effiziente Iterationsplanungssitzungen zu unterstützen
- **Verbindung von Entwicklungs- und Testaufgaben** derselben Story, um ein vollständiges Bild des notwendigen Aufwands des Teams für die Implementierung der Story bereitzustellen

FirmTec Solutions AG

- Erfassen des **Aufgabenstatus** nach jedem Update der Entwickler und Tester, was automatisch einen aktuell berechneten Snapshot des Status jeder Story, der Iteration und des gesamten Releases liefert
- Bereitstellung einer **visuellen Darstellung via Schaubildern** und Dashboards für Status von: User-Story, Iteration, Releases
  → Gut für geografisch dezentralisierte Teams
- **Integration mit Konfigurationsmanagement-Werkzeugen**, was die automatische Aufzeichnung von Code Check-ins und Builds gegenüber Aufgaben und, in einigen Fällen automatisierte Statusupdates für Aufgaben ermöglicht

3.4.2 Kommunikations- und Informationsweitergabe-Werkzeuge

- Alt: Email, Dokus, verbal → Neu (→ zusätzlich): Wikis, Instant Messenger, Desktop Sharing → Vorteile von:
- **Wikis**
  - Produktfeaturediagramme, Featurediskussionen, Prototypdiagramme, Whiteboard-Diskussions-Fotos u.a.
  - Werkzeuge und/oder Techniken für Entwicklung und Test, die andere Teammitglieder nützlich finden
  - Metriken, Tabellen und Dashboards zum Produktstatus, die insbesondere dann wertvoll sind, wenn das Wiki mit anderen Werkzeugen verbunden ist, wie z. B. mit dem Build Server und dem Aufgabenmanagementsystem, da das Werkzeug den Produktstatus automatisch aktualisieren kann
  - Besprechungen zwischen Teammitgliedern, ähnlich denen per Instant Messenger oder Email, aber so, dass sie mit allen anderen Teammitgliedern geteilt werden
- **Instant Messenger, Telefonkonferenzen, Video Chat Werkzeuge:**
  - Ermöglichen direkte Kommunikation in Echtzeit, insbesondere bei dezentralisierten Teams
  - Sie beziehen dezentralisierte Teams in Stand-Up Meetings ein
- **Desktop Sharing & Erfassungswerkzeuge:**
  - In dezentralen Teams können damit Produktdemonstrationen, Code Reviews und sogar Pairing stattfinden
  - Aufzeichnen der Produktdemonstrationen am Ende jeder Iteration → Dann Einfügbar ins Team-Wiki
- **Direkte Kommunikation → Nach wie vor die Beste Variante, alles andere Notlösungen**

3.4.3 Werkzeuge für Build und Distribution → CI & Build-Distributionswerkzeuge → Siehe Kapitel 1.2.4

3.4.4 Werkzeuge für das Konfigurationsmanagement

- Für Speicherung von Quellcode, automatisierten Testskripts, manuelle Tests und anderen Arbeitsergebnissen
- Häufig im selben Repository hinterlegt wie Quellcode → Nachverfolgung SW Version VS. Testversion möglich
- Ermöglicht schnelle Änderungen ohne Verlust historischer Information
- Versionsverwaltung über **V**ersions**C**ontroll**S**ysteme (VCS) → Untersch. Zwischen centralized (CVCS) VS. distributed VCS (DVCS)
  - Welches VCS = f (Teamgrösse, Teamstruktur, Einsatzort, Integratiosanforderungen anderer Werkzeuge)

3.4.5 Werkzeuge für Testentwurf, Implementierung und Durchführung:

- **Testentwurf**: Z.B. Mind Maps beliebt um schnell Tests für neues Feature zu umreißen & entwerfen
- **Testfallmanagement** → können Teil des Werkzeugs für Anwendungslebenszyklusmanagement oder Aufgabenmanagement des gesamten Teams sein
- **Testdatenvorbereitung und –Erstellung**: Werkzeuge, die Daten generieren, um die Datenbank einer Anwendung zu bestücken, sind sehr von Vorteil, wenn viele Daten und Datenkombinationen nötig sind, um die Anwendung zu testen. Diese Werkzeuge können auch dabei helfen, die Datenbankstruktur neu festzulegen, wenn das Produkt geändert wird, und um die Skripts zu Generierung der Daten zu refaktorieren. Dies ermöglicht eine schnelle Aktualisierung der Testdaten im Fall von Änderungen. Einige Werkzeuge zur Testdatenvorbereitung nutzen Produktionsdatenquellen als Rohmaterial und verwenden Skripts, um sensible Daten zu entfernen oder zu anonymisieren. Andere Werkzeuge können dabei helfen, große Daten ein- oder -ausgaben zu bewerten
- **Testdatenladewerkzeuge**: Nachdem die Daten für die Tests erstellt sind, müssen sie in die Anwendung geladen werden. Manuelle Dateneingabe ist oft zeitaufwändig und fehleranfällig, aber es gibt Datenladewerkzeuge, die den Prozess verlässlich und effizient machen. Viele Datengenerierungswerkzeuge enthalten sogar bereits eine integrierte Komponente zum Laden der Daten. Es ist manchmal auch möglich, große Datenmengen über das Datenbankmanagementsystem hochzuladen
- **Automatisierte Testdurchführungswerkzeuge**: Es gibt Testdurchführungswerkzeuge, die tendenziell besser für agiles Testen geeignet sind. Spezielle Werkzeuge für Test First Ansätze, wie verhaltensgetriebene Entwicklung, testgetriebene Entwicklung und abnahmetestgetriebene Entwicklung gibt es sowohl von kommerziellen Anbietern als auch als Open Source Lösungen. Diese Werkzeuge ermöglichen es Testern und

FIRMTEC SOLUTIONS AG

Fachbereichsspezialisten das erwartete Systemverhalten in Tabellen oder in einfacher Sprache unter Nutzung von Schlüsselwörtern zu formulieren

- **Werkzeuge für exploratives Testen**: Für Tester und Entwickler nützlich sind Werkzeuge, die alle Aktivitäten in einer explorativen Testsitzung aufzeichnen und speichern. Wird ein Fehler gefunden, so ist ein solches Werkzeug von Nutzen, da alle Aktivitäten vor dem Auftreten des Fehlers nachvollziehbar aufgezeichnet sind. Dies ist für das Berichten des Fehlers an die Entwickler sinnvoll. Die Protokollierung der in einer explorativen Testsitzung durchgeführten Schritte kann von Vorteil sein, wenn der Test schließlich in der automatisierten Regressionstestsuite integriert ist

3.4.6 Cloud Computing und Virtualisierungswerkzeuge

- Virtualisierung → eine einzelne Ressource (z.B. Server) → Darstellung als separate, kleinere Ressourcen
- Bei Benutzung von virtuellen Maschinen / Cloud → Grössere Anzahl von Ressourcen für Teams (Entw. & Test)
- Kann Verzögerungen verhindern durch Wartungsarbeiten
    - Neuer Server zur Verfügung stellen / Wiederherstellen ist einfacher durch Snapshot-Funktion
    - Snapshot → Abspeicherung einer aktuellen Situation → kann jederzeit wiederhergestellt werden
- Einige Testmanagementwerkzeuge → nutzen Virtualisierungstechniken, um Server-Snapshot im Fehlermoment anzufertigen

FIRMTEC SOLUTIONS AG