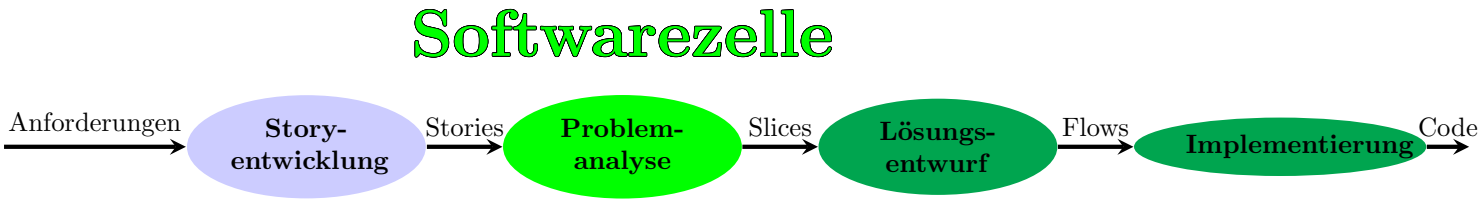
Anforderungsanalyse

Motivation

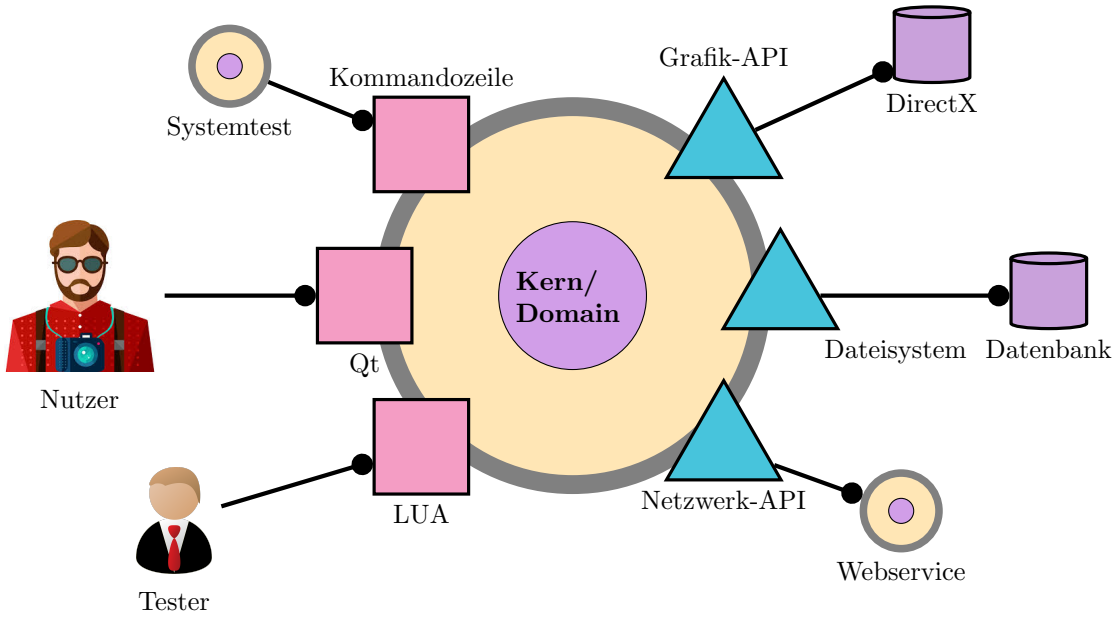
Der Anfang jeder Aufgabe ist das Verstehen des Problems/ der Anforderung. Hierbei ist die Absicht, wie bei dem Lösungsdesign, effizient zu einem sauberen Ergebnis zu kommen. Außerdem soll das Ergebnis der Analyse einen nahtlosen Einstieg in das Design ermöglichen. Das hier vorgestellte Verfahren orientiert sich an einem erfolgreichen Konzept der Natur, dessen Eigenschaften, skalierbar, Trennung System von Umwelt, funktional voneinander unabhängig, Kommunikation mittels Nachrichtenaustausch und inkrementelle Entwicklung ebenso hilfreich für die Softwareentwicklung sind.

Überblick



1 Die Softwarezelle

Zellen in der Natur sind **unabhängige** Einheiten, die durch eine Membran von der **Umwelt getrennt** sind und mittels **Botenstoffen** mit ihr kommunizieren. Viele Zellen **bilden** Organe, Knochen, Haare, usw, aus vielen kleinen Teilen wird etwas anderes Großes und das **skaliert** sehr gut. Da all diese Eigenschaften auch für die Softwareentwicklung wünschenswert sind, verwenden wir die Konzepte in der **Softwarezelle**.



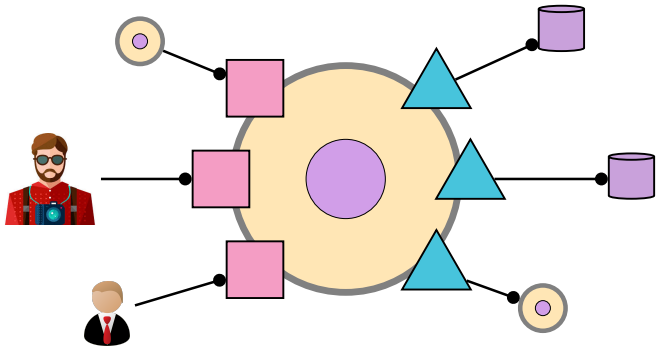
- Die Softwarezelle bietet der Umgebung Dienste an, auf die über Portale **(Adapter)** zugegriffen werden kann.
- Die Softwarezelle nutzt Dienste der Umgebung die über Provider **(Adapter)** zur Verfügung gestellt werden.
- Der Kern der Softwarezelle enthält die Domainlogik, die vom Nutzer als Funktionalität wahrgenommen wird.
- Adapter kapseln die unterschiedlichen APIs, damit der Kern unabhängig davon bleibt.
- Die Funktionsbereiche Kern, Provider und Portale sind getrennt und wissen nichts voneinander.

1.1 Anforderung zerlegen (Slicing)

Verständnis wird durch Funktionssignaturen mit den jeweiligen Akzeptanztests eindeutig zum Ausdruck gebracht. Erst dann kann man mit dem Lösungsentwurf sinnvoll beginnen. Im folgenden werden wir eine strukturierte Zerlegung sehen, um Funktionssignaturen und zugehörige Testfälle zu finden.

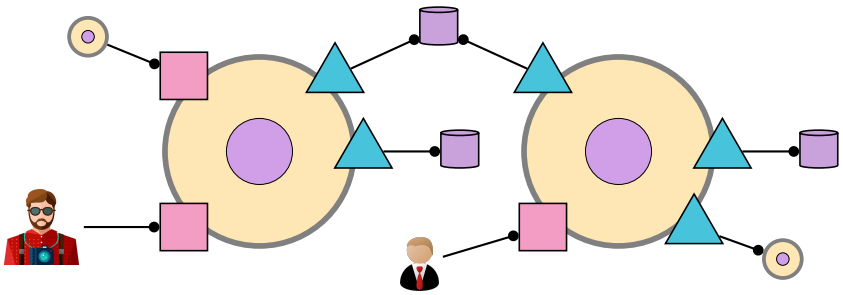
Gesamtsystem

- Finde alle Systemabhängigkeiten
- Wer hängt vom System ab (Benutzer)?
- Wovon hängt das System ab (Ressourcen)?



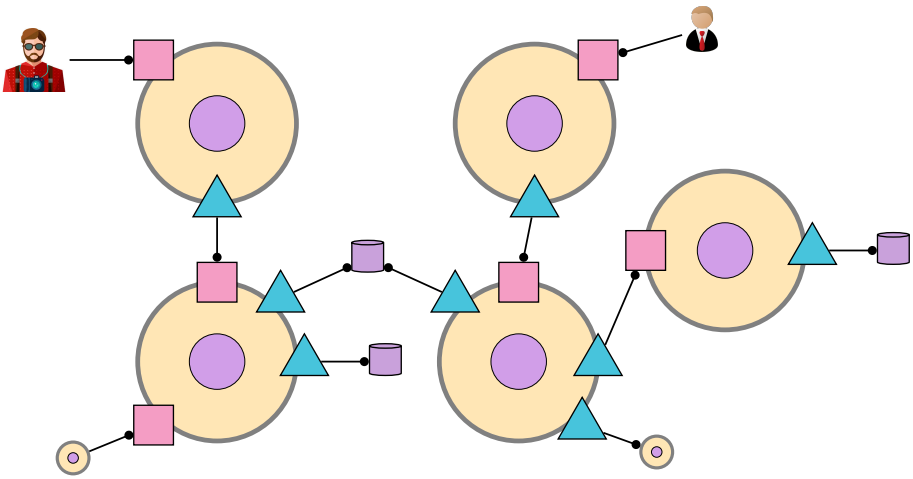
Anwendungen

- Funktionale Anforderungen: Nutzergruppen benötigen unterschiedliche Funktionalität des Systems (Use Cases)
- Unterteilung des Systems um Benutzererlebnis zu verbessern (z.B. Administrator und Benutzer)



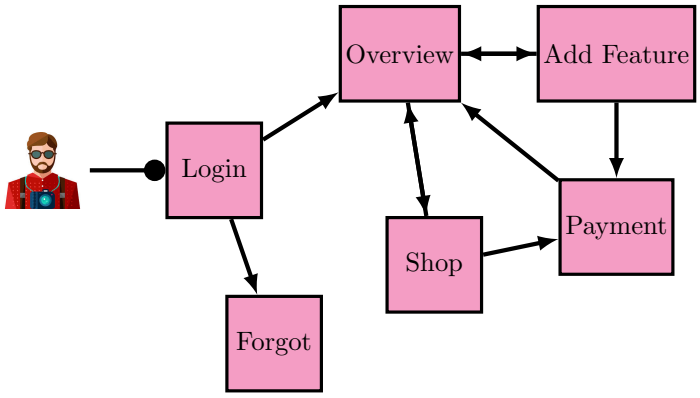
Co-Workers/Module

- Nicht funktionale Anforderungen: Ressourcen, Paradigmen, Optimierungen, örtliche Trennung, können zu Unterteilung führen
- Zum Beispiel unterschiedliche Programmiersprachen (UI in C#, Logik in C++) oder Teile des Systems sollen später einzeln verkaufbar bzw. nachkaufbar sein, Client - Server Verteilung, usw.



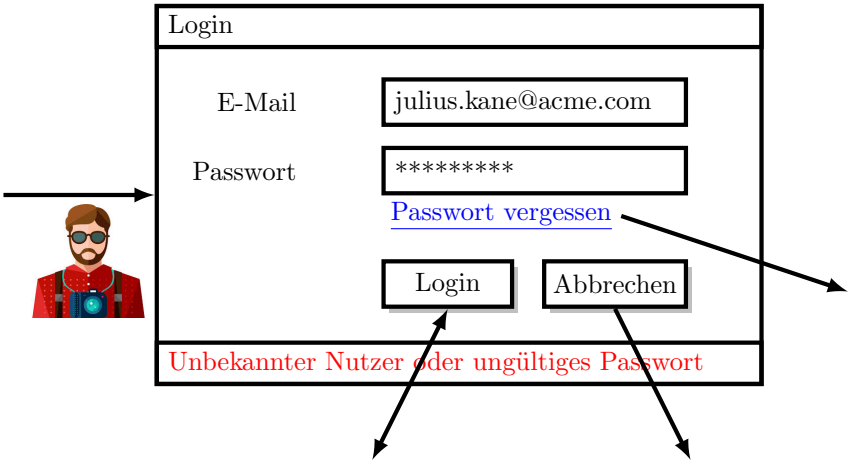
Dialoge

- Für jeden Co-Worker wird jedes Portal analysiert
- Bestimme alle Dialoge die zu dem Portal gehören
- Bestimme alle Dialogübergänge
- Nur kurz den Zweck des Dialogs nennen, keine Details



Interaktionen

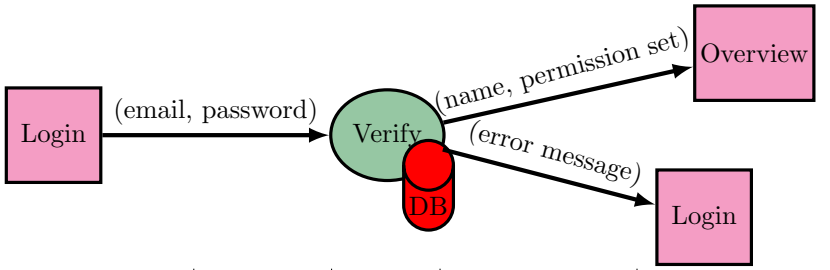
- Für jeden Dialog die Aktionen bestimmen
- Wie/Wo kann ein Benutzer ein Verhalten auslösen?
- Wann muss Logik hinter einem Auslöser ausgeführt werden, um ein bestimmtes Verhalten zu erzeugen?
- Die Interaktionsanalyse erfolgt am besten mit Spezialisten für Benutzeroberflächen.



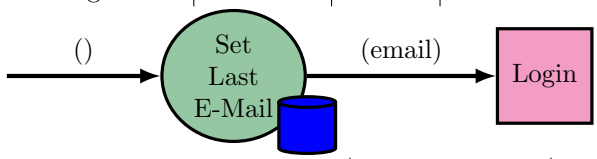
1.2 Entwurf in die Breite

Nachrichten

- Für jede Interaktion den genauen Nachrichtenfluss definieren.
- Was sind relevante und erwartete Übergänge von Eingabe (Anforderung, Zustand) zu Ausgabe (Antwort, Neuer Zustand)?
- Ergebnis sind Funktionseinheiten, wo anhand der Liste von **Akzeptanztests** die Eingaben, Ausgaben, das Verhalten und möglich Seiteneffekte klar abzulesen sind.



email	password	name	permission set	error message
das@gmx.de	123456	Darius	RWX	-
du@google.de	x	-	-	unbekannte E-Mail
das@gmx.de	123	-	-	falsches Passwort



Zustand	Aufruf
erster Start	{}
letzte E-Mail: das@gmx.de	{das@gmx.de}

1.3 Auswahl einer Interaktion

Reihenfolge legt der Produkt-Owner grob fest. Entwickler sollte technisch schwierige Interaktion zuerst angehen.
Die Umsetzung einer ausgwählten Nachricht wird als **Inkrement** bezeichnet.

1.4 Entwurf in die Tiefe (siehe Kapitel Flow Design)

In diesem Schritt wird der Lösungsentwurf **einer** Interaktion vorgenommen.

1.5 Arbeitsorganisation

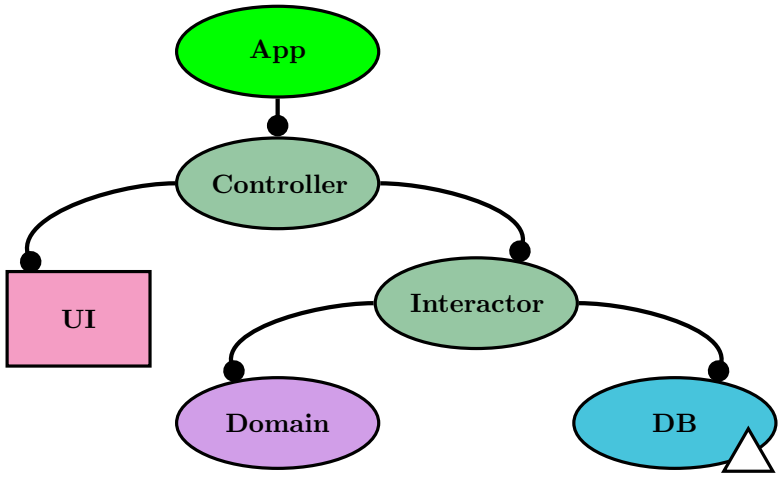
Da die Arbeit im Team erfolgt muss klar sein wer woran arbeitet und wie man vermeidet sich gegenseitig zu behindern.

1.5.1 Übergang zur Umsetzung

1. Ist jede Funktionseinheit nur für genau einen Aspekt zuständig?
2. Kann jede Funktionseinheit in maximal vier Stunden implementiert werden?
3. für **jede Funktionseinheit** entscheiden ob diese als *Funktion* oder *Klasse* umgesetzt wird
4. zugehörige Klassen für Funktion bzw. Dateiname für Klasse festlegen
5. Klassen mit Verhalten als Schnittstelle auslegen und Klassen ohne Verhalten als Datenklassen (einfaches struct)
6. die Schnittstellen und Datenklassen im Ordner Contracts sammeln und von dort referenzieren

1.5.2 Empfohlene Struktur für Desktopanwendungen

Die unterschiedlichen Funktionsbereiche sollen unabhängig und testbar bleiben. Ein nützliches Muster dafür verteilt die Integration auf mehrere Schichten:



- **App** Ist der Einstiegspunkt in die Anwendung.
  Sie erzeugt alle benötigten Funktionseinheiten, löst Abhängigkeiten auf und gibt die Kontrolle an die Benutzerschnittstelle ab.
  Bei größeren Anwendungen wird das Erzeugen auf die Controller verteilt.
- **Controller** verknüpft die **UI** mit ihrem **Interactor**.
  Für das Testen der gesamten App ohne die UI ist es sinnvoll die UI dem Controller als Schnittstelle zur Verfügung zu stellen, damit man diese im Test gegen ein Fake austauschen kann.
- **Interactor** verknüpft die **Domänenlogik** mit den **Ressourcenzugriffen**.

1.6 Umsetzung Einzeln (Implementation + Tests First)

Durch Unittests gestützte Entwicklung der jeweiligen Logik (UI, Domain, DB).

1.7 Umsetzung Team (Integration + Tests First)

Durch Integrations-/Akzeptanztests gestützte Integration/Verknüpfung der jeweiligen Logik (UI, Domain, DB) im Interactor/Controller. Die Akzeptanztests wurden bereits bei Entwurf in die Breite gefunden.

1.8 Code Review im Team

- Wurden alle Anforderungen umgesetzt?
- Genügt die Codequalität den Ansprüchen?
- Ist die Testabdeckung hoch genug?
- Sind die Tests verständlich genug?