# IMAP-Client

Softwaretechnik I SoSe 2021

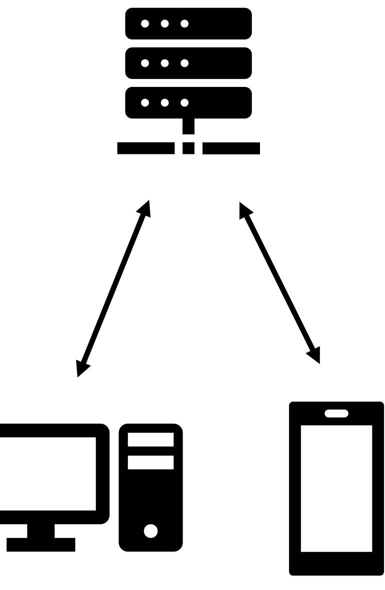
15.07.2021

Smilla Fox, Elena Gensch, Jan Groeneveld, Christian Helms, Lukas Rost, Johann Schulze Tast (Gruppe 1)

# Grundlagen

Internet **M**essage **A**ccess **P**rotocol (IMAP)

- Protokoll für den Zugriff auf E-Mails auf einem Mailserver
- E-Mails bleiben auf dem Mailserver
  - Mehrere Clients können Postfach verwalten
  - Ordnersystem auf dem Server
- Erweiterte Funktionalitäten serverseitig

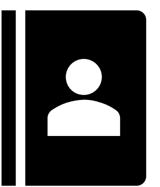


# Live-Demo

# Funktionale Anforderungen

# Offline Modus

- E-Mails offline
  - lesen
  - bearbeiten
    - Flags setzen
    - zwischen Ordnern verschieben
    - Sortieren für die Anzeige (nach Name, Datum, …) (jeweils einzelne Userstory)
- Änderung bei Internetverbindung synchronisieren
- Anzeige, wann zuletzt synchronisiert wurde
- Client öffnet sich ohne Passworteingabe im Offline-Modus



# Adressbuch

- Anzeige von Kontaktname
- Kontakte
  - Hinzufügen
  - Löschen
- Sortieren der Einträge im Adressbuch nach Name, Emailadresse
- Eindeutige Symbole für
  - Kontakt hinzufügen
  - Adressbuch öffnen

# Projektverlauf - User Stories

| Sprint 1 | Sprint 2 | Sprint 3 | Sprint 4 | Sprint 5 |
|---|---|---|---|---|
| 3. Mai - 17. Mai | 17. Mai - 31. Mai | 31. Mai - 14. Juni | 14. Juni - 28. Juni | 28. Juni - 12. Juli |
| Mails offline lesen | Mails offline flaggen | Refactoring | Mails offline verschieben | |
| Offline-Modus | Synchronisationszeitpunkt anzeigen | | Zugriff auf gespeicherte Mails ohne Login | |
| Adressbucheinträge hinzufügen | Kontaktnamen anzeigen | Adressbuch sortieren | | |
| Adressbuch | Designänderungen | | | |

# Nichtfunktionale Anforderungen

**Robustheit**

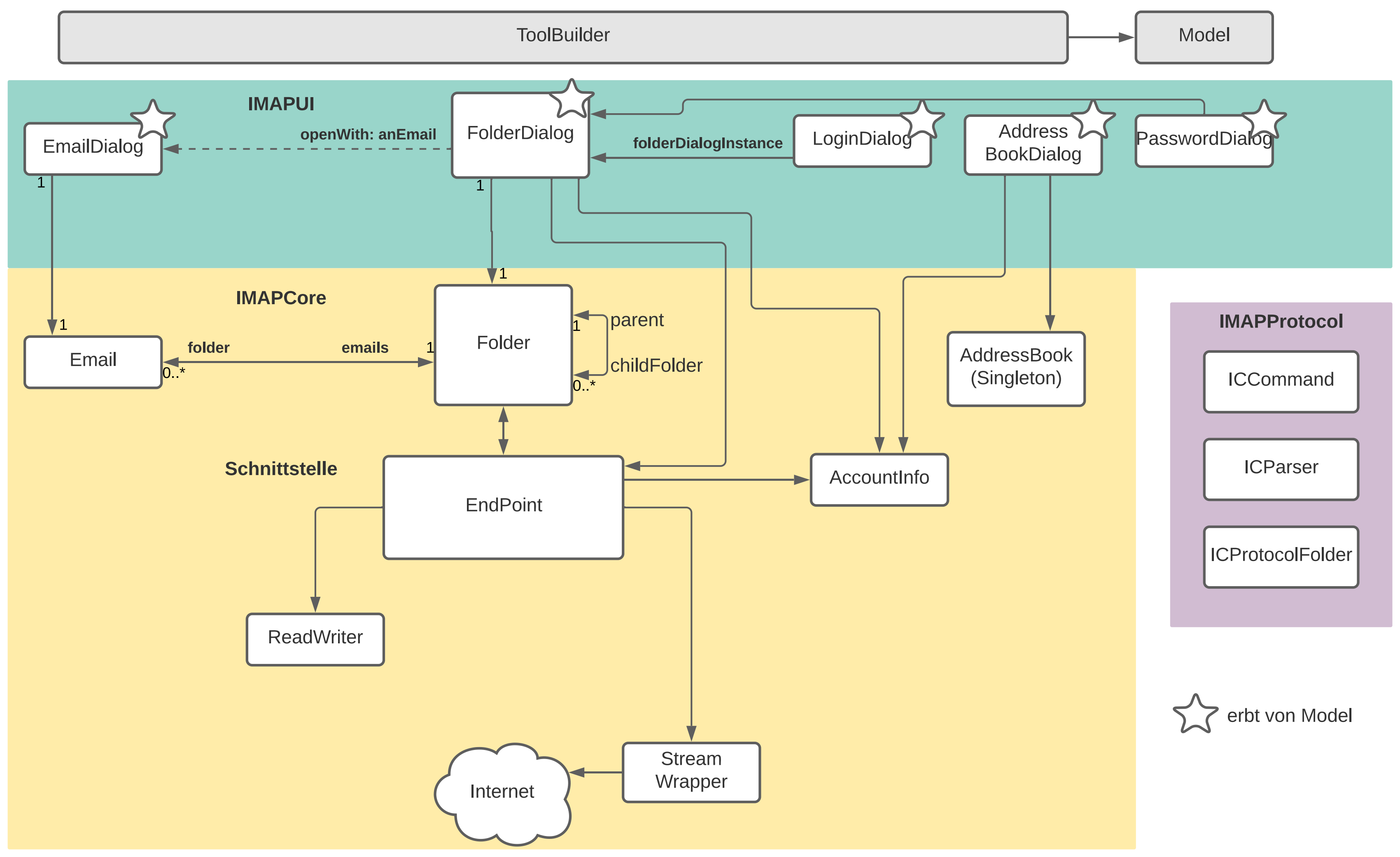
- Durch Failure des IMAP-Clients keine Seiteneffekte, die nach Wiederöffnung des IMAP Clients Infektionen bewirken

**Verlässlichkeit**

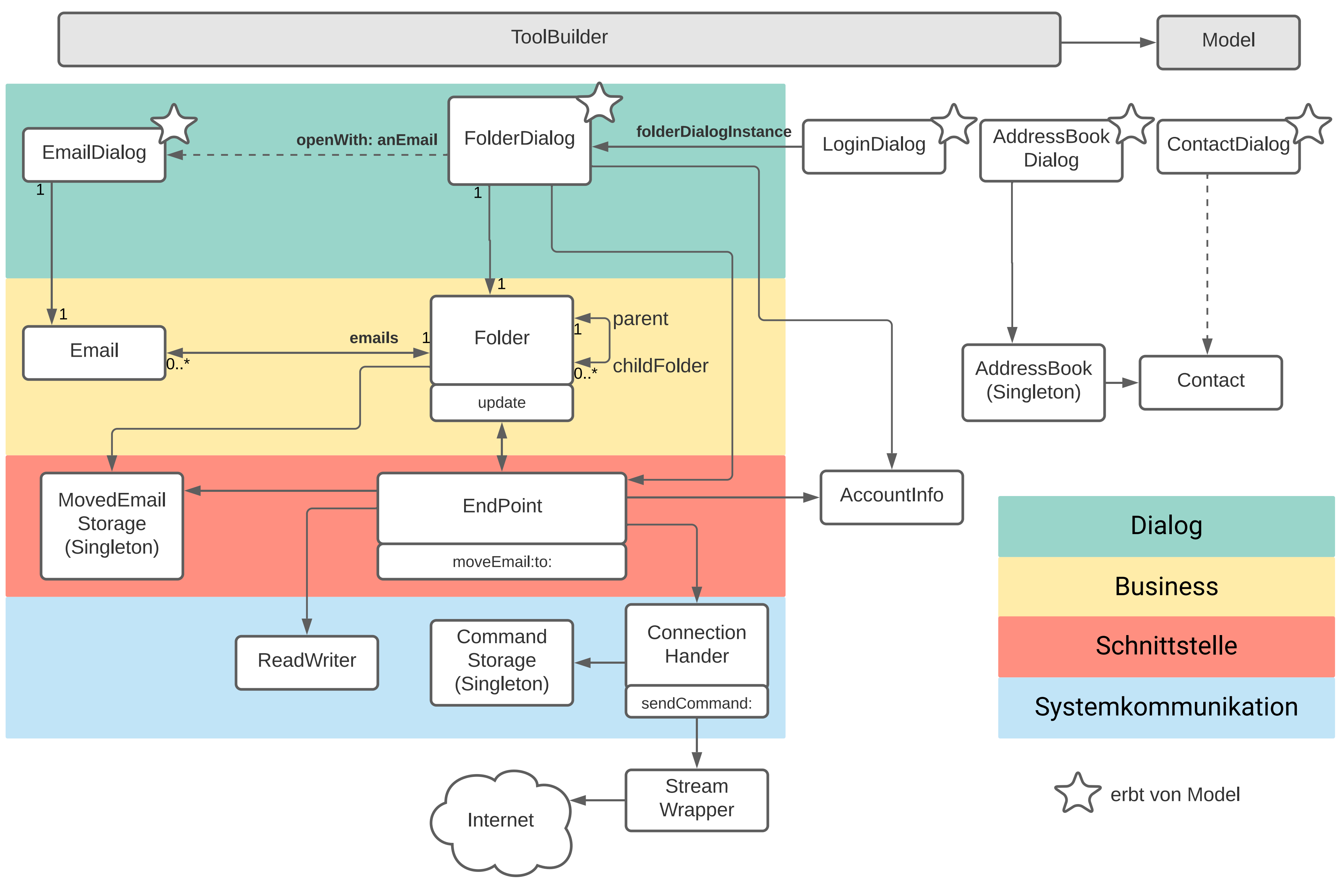
- Unterschied auf verschiedenen Betriebssystemen: kein betriebssystemspezifischer Code für Fehlerbehebung

**Standards**

- Neue Features werden in separatem Branch entwickelt
- Reviews und grüne Tests



ToolBuilder
Model
IMAPUI
EmailDialog
openWith: anEmail
FolderDialog
folderDialogInstance
LoginDialog
Address BookDialog
PasswordDialog
1
1
1
1
IMAPCore
Email
folder
emails
1
0..*
Folder
parent
childFolder
1
0..*
AddressBook (Singleton)
IMAPProtocol
ICCommand
ICParser
ICProtocolFolder
Schnittstelle
EndPoint
AccountInfo
ReadWriter
Stream Wrapper
Internet
erbt von Model


ToolBuilder
Model
EmailDialog
openWith: anEmail
FolderDialog
folderDialogInstance
LoginDialog
AddressBook
Dialog
ContactDialog
1
1
1
1
Email
emails
1
Folder
parent
1
childFolder
0..*
0..*
update
AddressBook
(Singleton)
Contact
MovedEmail
Storage
(Singleton)
EndPoint
moveEmail:to:
AccountInfo
Dialog
Business
Schnittstelle
Systemkommunikation
ReadWriter
Command
Storage
(Singleton)
Connection
Hander
sendCommand:
Internet
Stream
Wrapper
erbt von Model

# Verhaltenstechnische Zusammenhänge - CommandStorage (User Story tracing)

User-Stories zum Offline Mode:

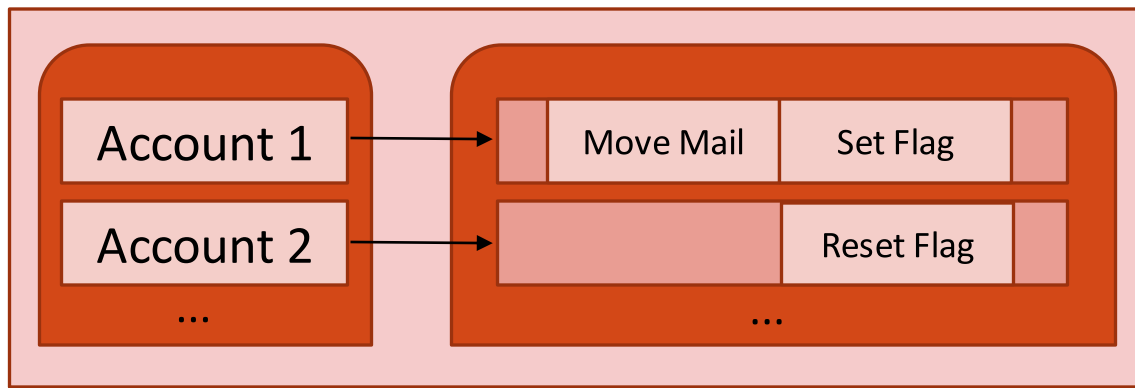
Ich als Nutzer möchte **offline Flags setzen** können, damit ich auch unterwegs ohne Internet arbeiten kann. Dabei sollen die Änderungen wieder **auf den Server synchronisiert** werden, sobald eine Internetverbindung besteht.

# Verhaltenstechnische Zusammenhänge - CommandStorage

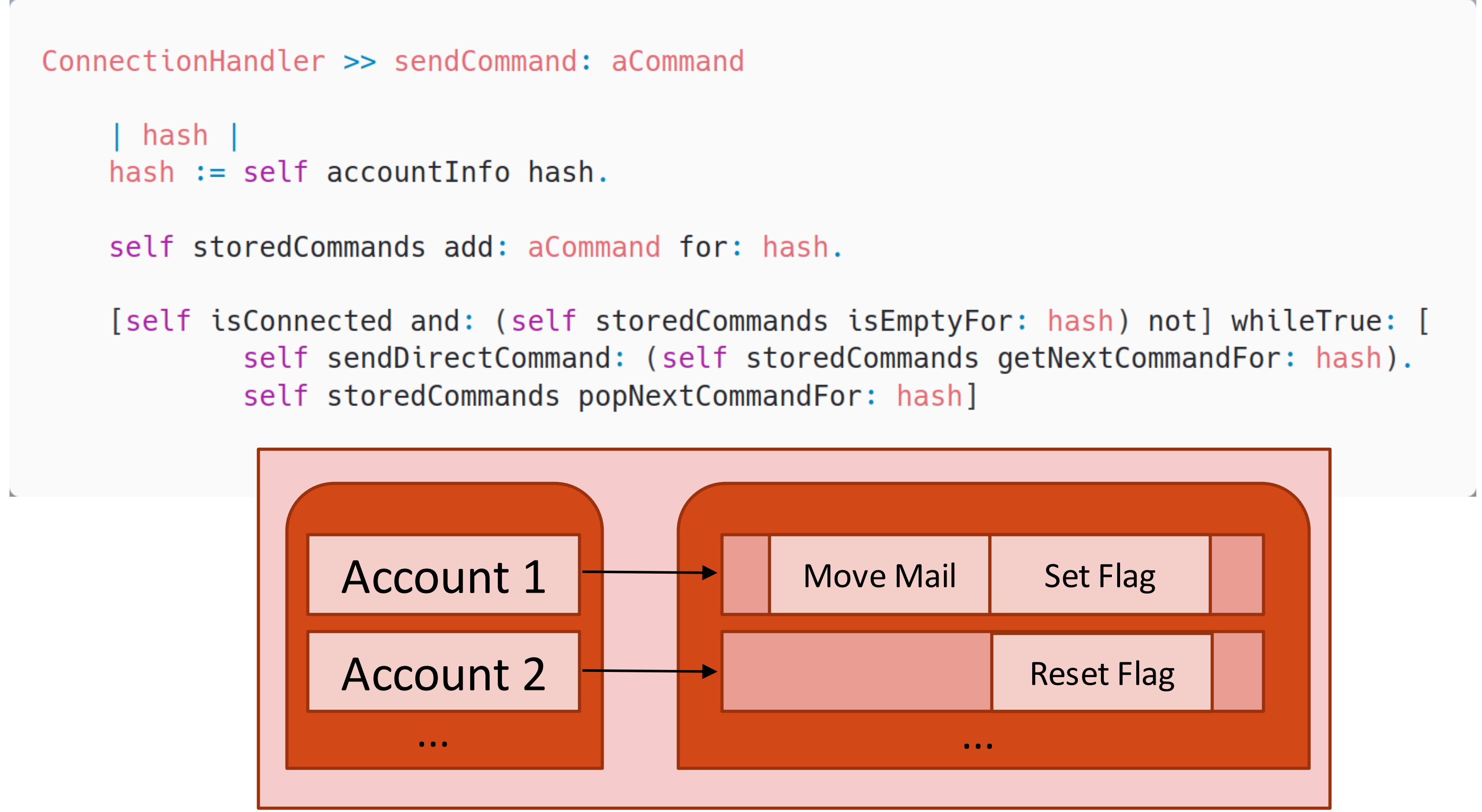
**Aufgabe**: Änderungen, die offline gemacht werden, müssen auf den Server synchronisiert werden.

**Lösung**: neue Klasse CommandStorage

- Dictionary mit mehreren Queues für die verschiedenen Accounts



# Verhaltenstechnische Zusammenhänge - CommandStorage

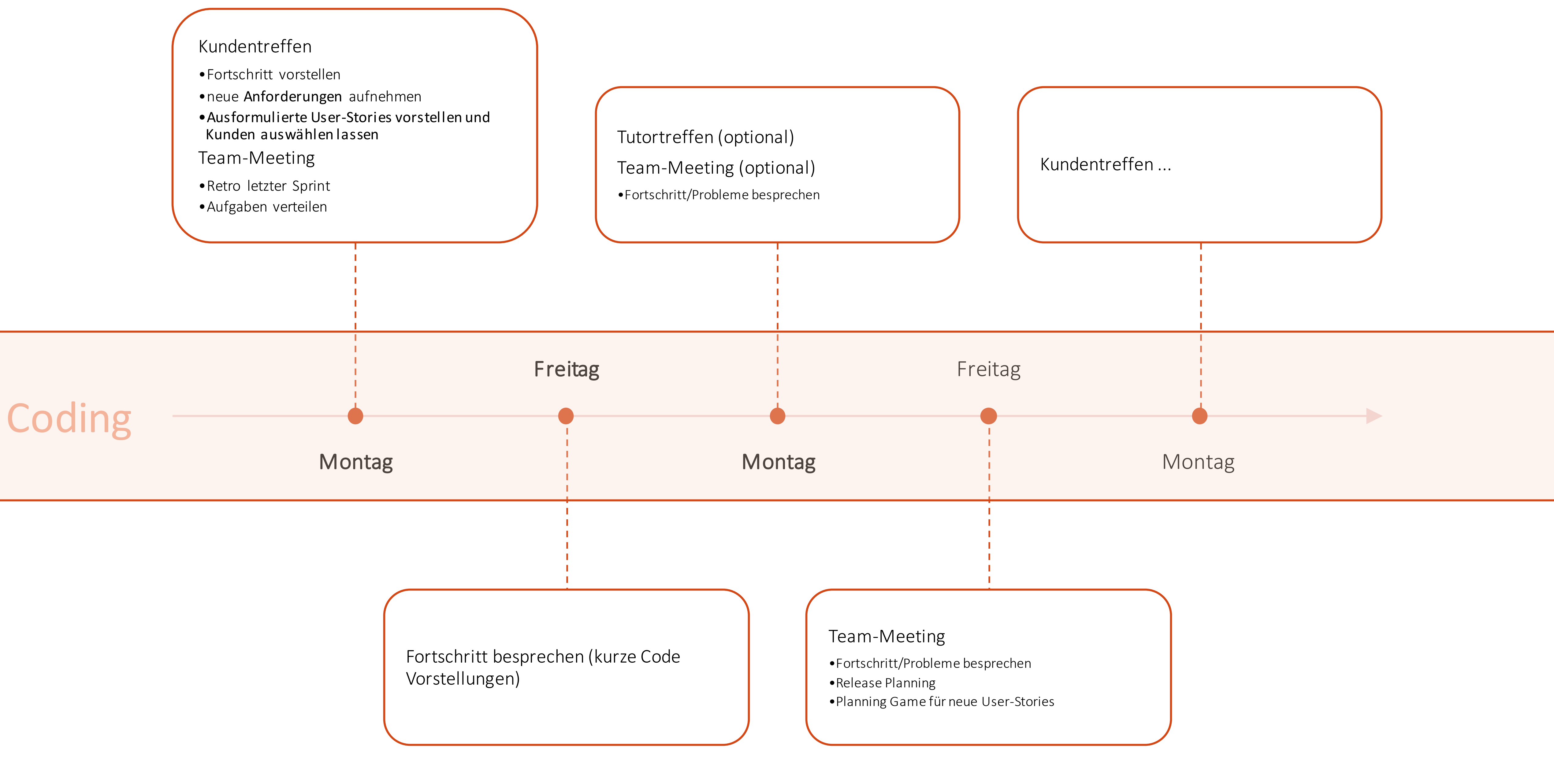


```
ConnectionHandler >> sendCommand: aCommand

    | hash |
    hash := self accountInfo hash.

    self storedCommands add: aCommand for: hash.

    [self isConnected and: (self storedCommands isEmptyFor: hash) not] whileTrue: [
            self sendDirectCommand: (self storedCommands getNextCommandFor: hash).
            self storedCommands popNextCommandFor: hash]
```

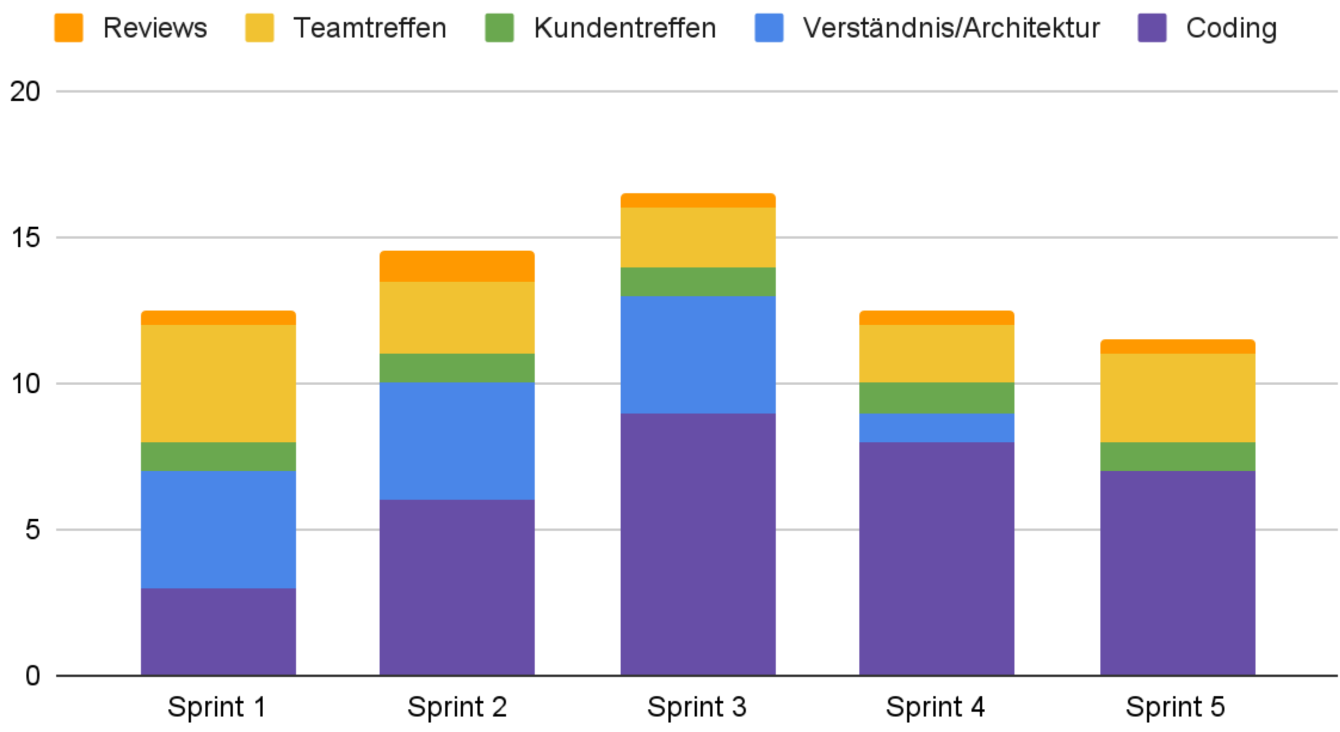
# Entwicklungsprozess

## Ausgangsbedingungen

- Bereits in SWA ein Team
- Beschränkte Erfahrung mit Anwendungsdomäne: nutzen Emails, noch nicht an Email-Clients gearbeitet, IMAP Protokoll aus WWW grob bekannt
- Squeak-Wissen aus SWA, neu: ToolBuilder
- Besondere Ausgangsbedingungen: viel Legacy Code - 5 Iterationen, 4157 Zeilen Code (mit Tests)


Kundentreffen
•Fortschritt vorstellen
•neue Anforderungen aufnehmen
•Ausformulierte User-Stories vorstellen und Kunden auswählen lassen
Team-Meeting
•Retro letzter Sprint
•Aufgaben verteilen
Tutortreffen (optional)
Team-Meeting (optional)
•Fortschritt/Probleme besprechen
Kundentreffen ...
Coding
Freitag
Freitag
Montag
Montag
Montag
Fortschritt besprechen (kurze Code Vorstellungen)
Team-Meeting
•Fortschritt/Probleme besprechen
•Release Planning
•Planning Game für neue User-Stories

# Arbeitszeit



# Entwicklungsprozess

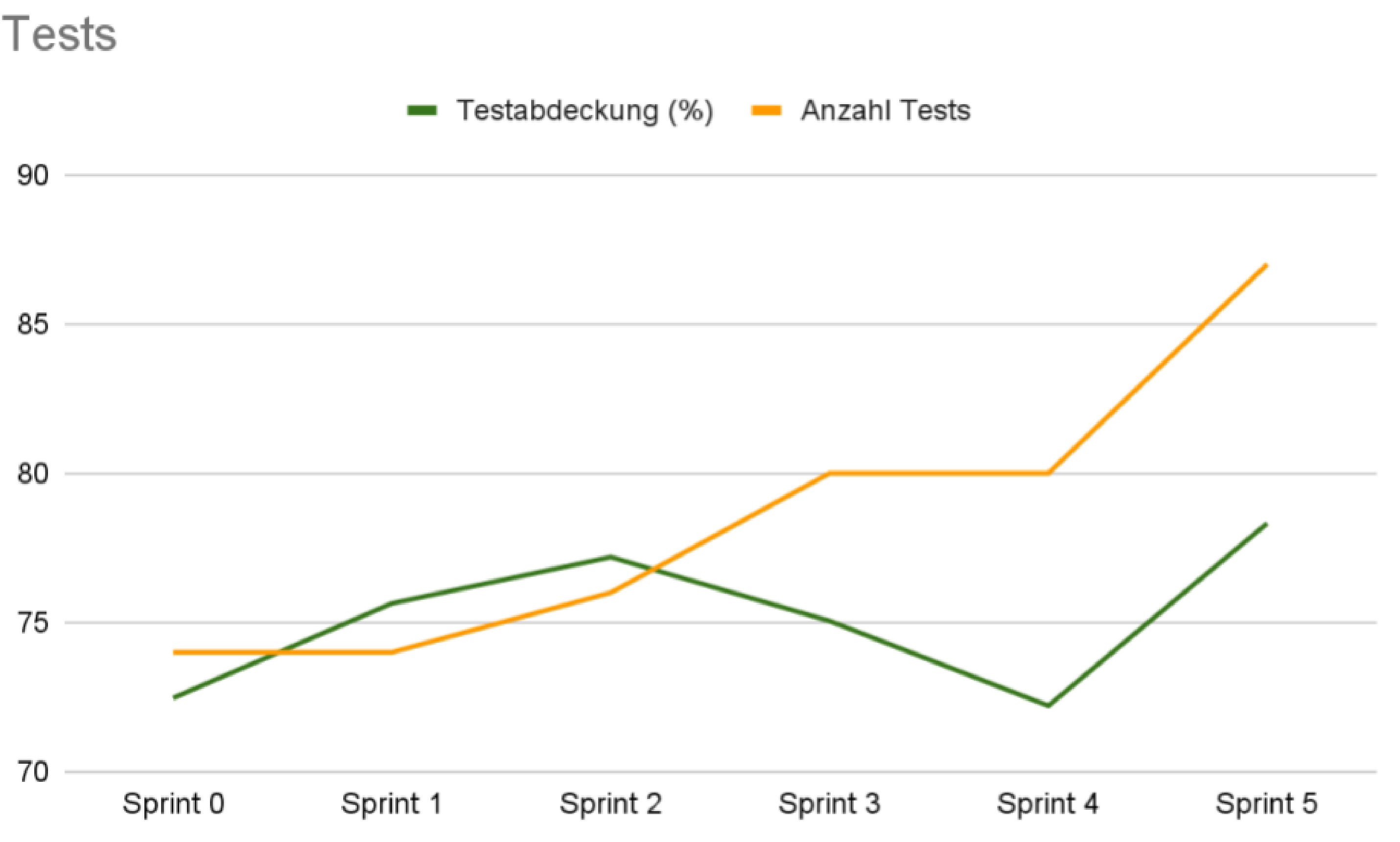
## Probleme

- Großer Zeitaufwand
  - **Lösung:**
    - Meetingzeiten reduzieren – vorher festlegen, was besprochen werden muss, dann Timer setzen
    - User-Stories kleiner teilen
    - Nicht vergessen, Zeit zum Verstehen des Legacy-Codes einzuberechnen

# Entwicklungsprozess

- Unorganisierter Anfang:
  - Wer schreibt im Kundengespräch mit? Wer formuliert User-Stories aus?
  - **Lösung:** Für jeden Sprint wird ein Sprint-Beauftragter festgelegt. Dieser trifft Vorbereitungen für das Kundentreffen, schreibt im Kundengespräch mit und hat alle Aufgaben im Überblick (erinnert ggf. andere Teammitglieder)

# Metriken - Tests




- vergleichsweise geringe Testabdeckung
  - **Grund**: viel ungetesteter Legacy-Code
  - Besonders Server-Kommunikation -> Aufwand groß, Fokus stattdessen auf neue Funktionalität
- **Beobachtung**:
  - Schreiben von Tests für Legacy-Code kann helfen, diesen zu verstehen
  - Sicherheit bei Veränderungen
- **Entscheidung**: restliche Zeit nutzen um Testabdeckung zu erhöhen

# Simple Design

Grundsatz YAGNI - “You Aren’t Gonna Need It”

- Aktuelle Anforderungen umsetzen, statt für die Zukunft zu planen

Erwartete Vorteile

- Kein "Overdesign" oder verschwendete Zeit um nicht Benötigtes zu implementieren
- Verständlicher Code mit dem gut weitergearbeitet werden kann

Risiken

- Späterer Refactoring – Aufwand

# Simple Design

1. Tests sind grün
2. Code drückt Intention aus
3. Keine Code-Duplikation
4. Wenige Elemente (Klassen, Methoden...)

**Refactoring notwendig!**

# Simple Design ⇆ Refactoring

**Refactoring**

Ziel: Codestruktur verbessern aber Funktionalität beibehalten

- Danach neue Features simpler umsetzbar

Umsetzung:

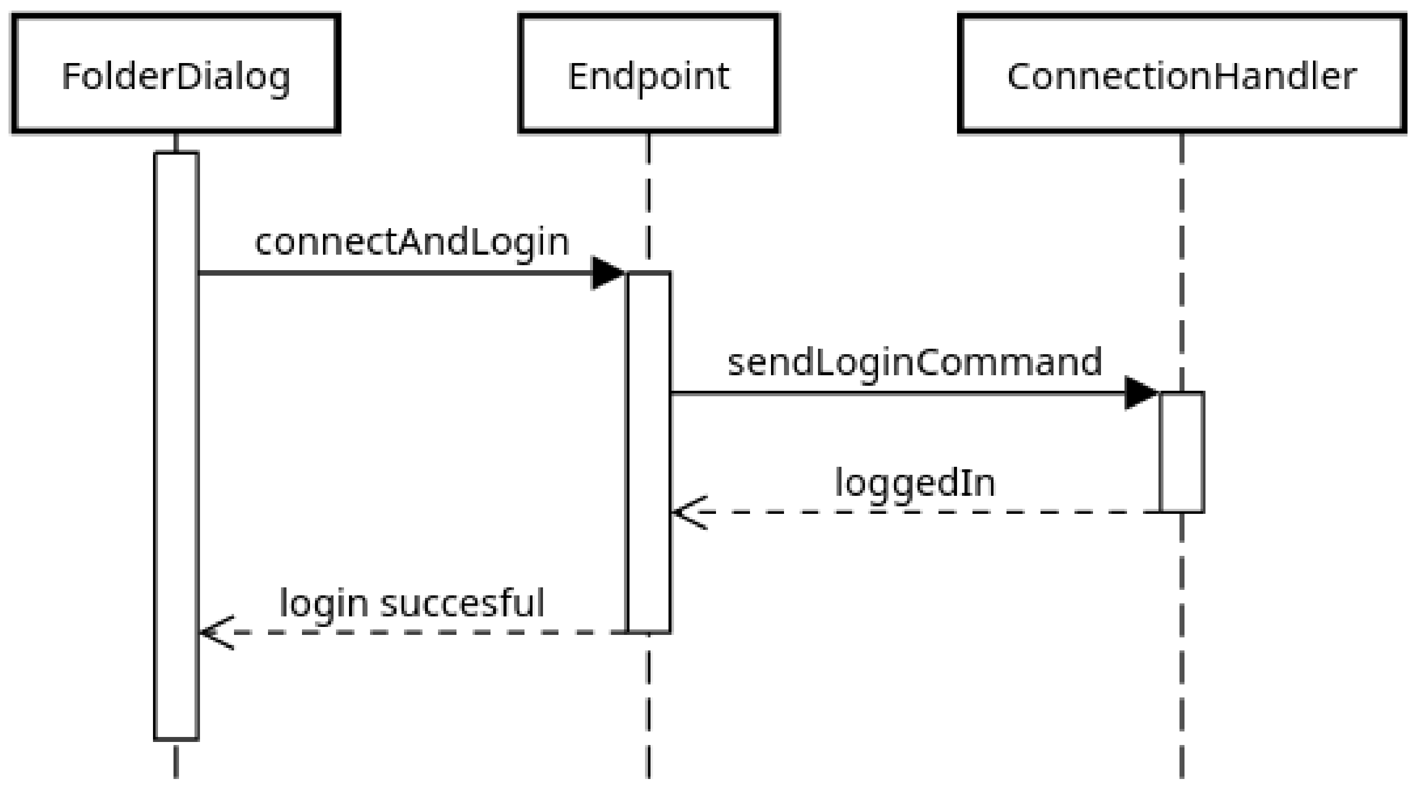
- Bei Umsetzen der User Stories: Ist Refactoring für das Feature sinnvoll?
- Große Designentscheidungen (und mögliche Nachteile) in Teammeetings diskutieren

# Refactoring Login User Stories

Ich als Kunde möchte **ohne eingeloggt** zu sein auf den **Offline-Modus** zugreifen, um zuvor abgerufene Mails zu lesen und zu bearbeiten.

Ich als Kunde möchte die Möglichkeit haben, mich in mein Konto **einzuloggen**, um auf die **Online-Funktionalität** zuzugreifen, **ohne den IMAP-Client erneut zu öffnen**

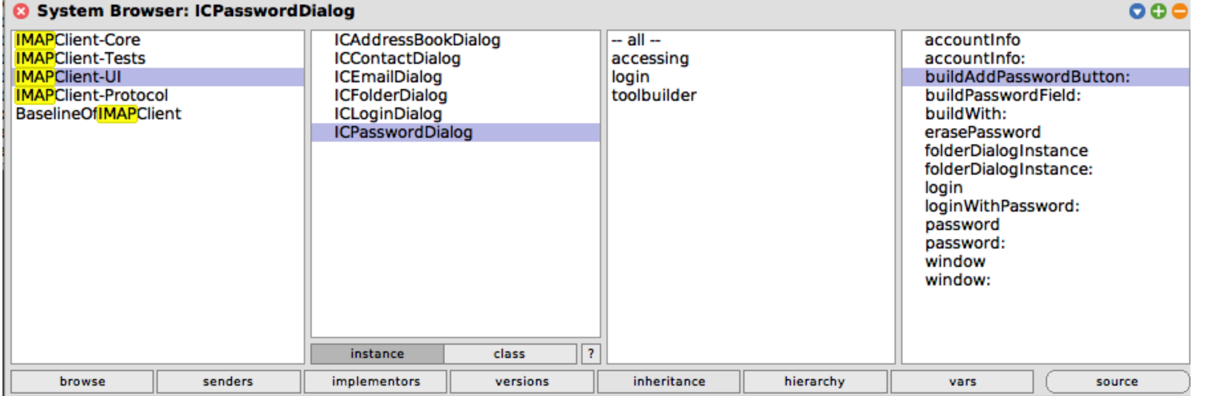
**Nach Änderungen:**



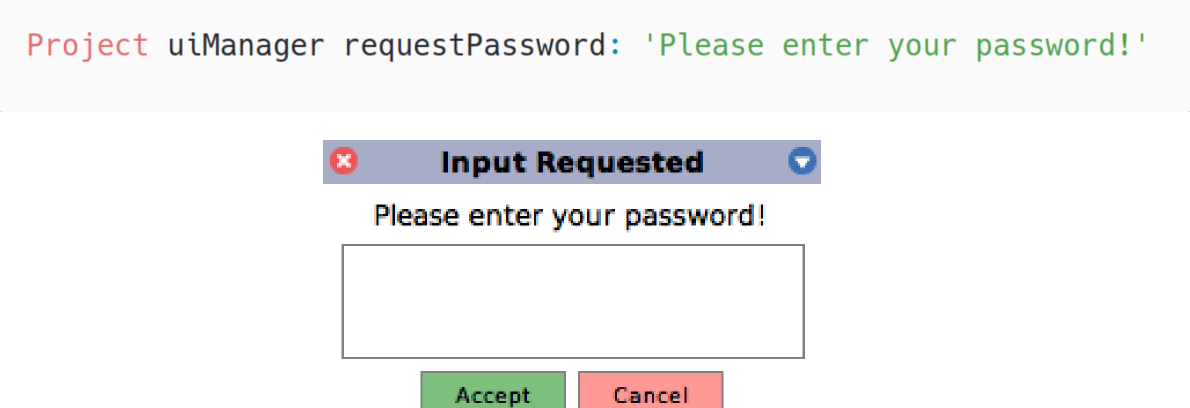

- Rückgaben: war Aktion erfolgreich
- connectAndLogin nur einmal aufrufen

# Refactoring - Login

- **Vorher:** PasswordDialog




**Nachher:** uiManager




- Gleiche Funktionalität 150 Zeilen weniger!
- Refactoring und umgesetzte User Stories:
  - Removed lines –249, added lines: +94

# Simple Design - Reflexion

| Das lief gut | Das lief schlecht | Das haben wir mitgenommen |
|---|---|---|
| • Mit Command Storage einfache und verständliche Lösung für Synchronisation gefunden<br>• Notwendiges Refactoring, um neue Funktionalität in Login-Prozess einzubauen<br>• Fehlerbehebung einfacher | • Umsetzung schwierig bei Legacy-Code mit komplexer Architektur | • Kleine Refactorings, die im Zusammenhang mit der Umsetzung von User Stories stehen, sind notwendig<br>• Bestehende Klassen (z.B. UI Manager) nutzen, statt neue hinzuzufügen<br>• Simple Design muss von Anfang an gedacht werden |

# Collective Code Ownership

*"I'm not afraid to change my own code. And it's all my own code."*

**Ziel**: gemeinsame Verantwortung über den Code

Was ist mein Code?

- Nur der, den ich selbst geschrieben habe?
- Unsere Definition: Code, dessen Hintergrund ich kenne, bei dem ich verstehe, welches konkrete Problem er löst und den ich selbst so *oder sehr ähnlich* geschrieben hätte

***Problem***: bislang unbekannter Legacy-Code

- ist nicht mein Code, weil ich ihn (noch) nicht verstehe
- **Lösung**: Code gemeinsam erarbeiten und Verständnis dokumentieren
  - Bei kritischen Stellen: Eine Person nutzt Debugger, andere schreibt Sequenzdiagramm mit
  - Vorjahresarchitektur analysieren (Wiki)

# Collective Code Ownership

***Problem***: manche Teammitglieder arbeiten mehr an bestimmtem Code und kennen sich mit diesem besser aus

- Gefahr: Entstehung von ungeteiltem Expertenwissen
- **Lösung**: Rotieren der Pair-Programming-Teams
  - zu Beginn Staffelstabübergabe: eine Person bleibt im Problembereich erhalten
  - Anforderungen berühren 2 Bereiche: Offline Mode (Funktionalität), Adressbuch (UI)
  - überraschend kurze Einarbeitungszeit
  - Aufbau von gemeinsamem Verantwortungsbewusstsein für den Code in Coding-Session (z.B. Achten auf Codequalität)
- zeitgleiches Arbeiten (Switchen in Breakout-Rooms möglich, um Fachwissen hinzuholen zu können)

# Collective Code Ownership

Schlechtes Fazit bei Code-Reviews und Code-Besprechungsmeetings:

- hoher Zeitaufwand/geringe Effizienz
- bei Code-Review auf Github: aufwändige selbstständige Erarbeitung des fremden Codes (schwieriger als nötig)
- Code-Meetings werden sehr lang
- Antwort auf Zeitdruck darf nicht sein, die Änderungen einfach nur zu bestätigen
- Review auf zu niedrigem Level: Probleme auf Code-Ebene werden leicht gefunden, verbesserungsbedürftige Herangehensweisen aber nur schwer

Lösung: In Meetings über Lösungsideen sprechen, Codeverständnis dann nur noch kleiner Schritt

⇒ erfordert Kommunikationsfähigkeiten: Ideen und Lösungen präzise und kurz erklären

**Gemeinsame Coding-Standards müssen eingehalten werden!**

# Collective Code Ownership

| Das lief gut | Das lief schlecht | Das haben wir mitgenommen |
| --- | --- | --- |
| • Gemeinsam den Code erarbeiten<br>• Rotation der Pair-Programming Teams<br>• Zeitgleiches Arbeiten der Teams mit geringen Kommunikationshürden (gleiche Zoom-Session, unterschiedliche Breakout-Rooms)<br>• Qualitätsprobleme konnten schnell von jedem behoben werden<br>• Gutes allgemeines Systemverständnis ⇒ half auch beim Schätzen | • Code-Reviews auf Github<br>• Meetings mit Vorstellung des Codes<br>*Zu hoher Zeitaufwand, nur auf Codelevel und am Ziel (CCO) vorbei* | • Präzise und kurze Kommunikation über Lösungsideen muss vorbereitet werden und hilft enorm!<br>• Alle müssen die Lösungsideen und Herausforderungen kennen, Verständnis des einzelnen Codes folgt schnell!<br>• **Gemeinsame Coding-Standards müssen eingehalten werden!** |

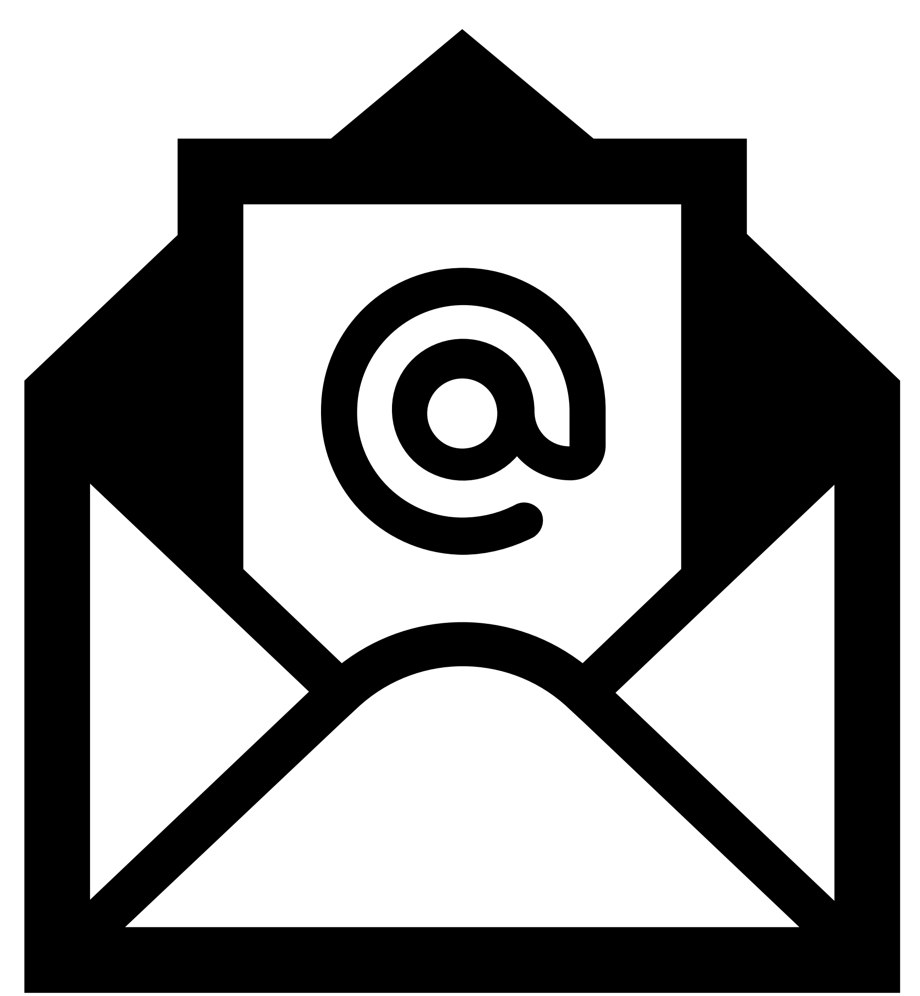
# Reflexion/Ausblick

Was haben wir gelernt?

Prozess
- Gleich zu Beginn Gedanken machen über organisierten Prozess: warum?
  - Umsetzung von XP-Praktiken fördert Fortschritt und Qualität des Projekts (siehe Test-Coverage-Verlauf, Arbeitszeit, Bugs)
- Verständnis für Legacy Code braucht Zeit
- Teamkommunikation: durch Collective Code Ownership gezielter
- Retros sind wichtig! - nicht vergessen regelmäßig über den Prozess zu reflektieren und immer wieder Anpassungen zu treffen

Teamarbeit
- Erwartungen am Anfang ehrlich besprechen

Zusammenarbeit mit Kunden
- Vorher im Team besprechen, um klare Kommunikation mit Kunden zu ermöglichen
- Ehrlichkeit und Transparenz sind zwingend für sinnvolle Zusammenarbeit: Probleme dem Kunden nicht verschweigen



# THE END