Softwareentwicklung und **das Projekt**

Lars Wechsler

31. Mai 2023

Inhaltsverzeichnis

1 Projektorganisation

1.1 Grundlagen der Softwaretechnik

In diesem Kapitel soll es nicht mehr um Projekte gehen, die jeder Einzelne alleine angehen kann, sondern um das „Programmieren im Großen“ - sei es in einer (internationalen) Firma oder „privat“ in einem Verbund mit anderen Programmierys.
Gerade in großen Projekten ist eine qualitativ hochwertig Zusammenarbeit unerlässlich. Leider gibt es hier immer wieder große Reibungsverluste, neben den fachlichen Inhalten ist die Fähigkeit des „Teamwork“ - gerade heutzutage - ein immens wichtiger „soft skill“.
In der Informatik gibt es verschiedene Modelle, die die Projektorganisation strukturieren und z.T. veranschaulichen sollen (dazu gibt es ganze eigene Vorlesungen im Bereich der **Softwaretechnik**!).
Im Allgemeinen geht man grob von den folgenden Phasen im Leben eines Softwareprojekts aus:

1. **Anforderungen und Spezifikationen**
2. **Planung**
3. **Entwurf und Design**
4. **Implementierung und Integration**
5. **Betrieb und Wartung**
6. **Stilllegung**

Zur Beschreibung dieser Phasen, deren Interaktion und zeitlicher Abfolge gibt es verschiedenste Vorgehensodelle, von denen in folgenden Kapiteln einige vorgestellt werden sollen.
Zunächst aber noch Details zu den einzelnen Phasen

1.2 Der Lebenszyklus von Software

Anforderungen und Spezifikationen
Grundlegende Fragen, die eine Anforderungsanalyse zu Beginn eines Projektes beantworten sollte, sind z.B.:

- welches Problem soll konkret gelöst werden?
- welche Leistung soll das geplante Projekt erbringen?
- welche Personen müssen mit einbezogen werden?
- gibt es sich widersprechende Anforderungen verschiedener Personen/Gruppen?

Insbesondere die letzten beiden Punkte sollen verdeutlichen, dass gerade aus Kundensicht nicht immer Einigkeit über den Funktionsumfang oder die genauen Spezifikationen eines Programs herrscht. Eine genaue Spezifikation der zu erbringenden Leistung ist deshalb wichtig, um Unzufriedenheit diesbezüglich vorzubeugen.
Die Anforderungsanalyse beschränkt sich aber nicht nur auf konkrete Funktionalität, sondern bildet ein ganzes Spektrum an Faktoren ab:

- **Funktionale Anforderungen**: das was man zunächst erwarten würde - welche Funktion soll das System haben, wie soll es sich verhalten, etc.
- **Nichtfunktionale Anforderungen**: hier geht es eher um den eigentlichen Betrieb der Software, wie leistungsfähig soll sie sein - d.h. wie groß sind die Anforderungen an Speicher/Prozessor im laufenden Betrieb, ist die Software skalierbar, etc.
- **Designbedingungen**: gibt es bereits weitere technische Bedingungen, das könnte z.B. bereits existierende Software sein, zu der Schnittstellen vorhanden bzw. eine Kompatibilität hergestellt werden muss.
- **Prozessbedingungen**: dies sind eher interne Anforderungen der Entwickler - wie viele Personen sind notwendig, wie soll die Vorgehensweise bei der Entwicklung sein (zwar intern zu sehen, aber insbesondere auch der Kontakt mit dem Kunden - Ablieferung eines „fertigen“ Produkts oder Zwischenstände z.B.).

Übliche Verfahren, die zur Ermittlung der Anforderungen verwendet werden sind z.B. Fragebögen, Interviews, Simulationsmodelle, Workshops, etc.
Die Ergebnisse müssen natürlich verschriftlicht werden, der Auftraggeber fasst alles im sogenannten **Lastenheft** zusammen, das möglichst konkret die Gesamtheit aller Anforderungen beschreibt - kurz gesagt: **Was soll erstellt werden?**.
Das **Pflichtenheft** dagegen beschreibt - ebenfalls möglichst konkret - wie der Auftragnehmer die Anforderungen des Auftraggebers lösen möchte, kurz: **Wie und womit wird umgesetzt**. Erst nach Akzeptanz des Pflichtenhefts beginnt die eigentliche Umsetzung.
Hinweis: Diese Beschreibung entspricht in der Realität natürlich nicht immer den Gegebenheiten, sondern eher eine etwas akademische Sicht auf die Dinge. Gerade Software-Entwicklung ist ein sehr dynamischer Prozess und lässt sich selten gut in statische Kategorien verpacken. Die Schule kann hier kein sicheres, aktuelles Bild vermitteln, da Regeln und Abläufe ggf. in den Betrieben bereits überholt sind ⇒ Praktika deutlich nützlicher, um zu lernen „wie der Hase läuft".

Projektplanung
In jedem Fall muss es eine Form von **Projektmanagement** geben, dass das Projekt zunächst inital plant und die „Arbeit verteilt". Spätestens hier muss man sich für ein Vorgehensmodell (siehe nächste Kapitel) entscheiden. Grob könnte man dabei folgende Unterteilung treffen:

1. Sind sehr detaillierte Informationen zu den Anforderungen bekannt, bzw. gibt es keine Möglichkeit die Software „laufend anzupassen" (man denke z.B. an Flugzeugsoftware, die doch lieber gleich von Anfang an vollständig laufen sollte), so sind eher plan-getriebene, starre Verfahren wie das **Wasserfallmodell** oder **V-Modell** notwendig.

2. Sind die Informationen dagegen eher vage, der Auftrag ungenau oder eine hohe Flexibilität allgemein gefordert (da bereits wahrscheinlich ist, dass sich Anforderungen im Laufe der Entwicklung verändern), so bieten sich dynamische Modelle an, die sogenannten **agilen** Ansätze, z.B. **Scrum**.

Für Interessierte zum Nachlesen: Instrumente, die die Projektplanung beschreiben sind z.B. **Gantt-Diagramm**, **CPM-Netzplan**, **Petri-Netze**

Entwurf und Design
Hier geht es darum, dass die Entwickler die Rahmenbedingung für die konkrete Umsetzung des Produks festlegen, also z.B.:

- Welche Datenflüsse innerhalb (und außerhalb) der Software gibt es.
- Welche Algorithmen sollen verwendet oder entwickelt werden.
- Welche Schnittstellen sind notwendig.
- Welche Designprinzipien sollen verwendet werden, z.B. objektorientierte Prinzipien wie Abstraktion (Oberklassen), Kapselung, etc.

Zur Verschriftlichung können dabei die **UML-** Diagramme wie das Klassendiagram, das Sequenzdiagramm, oder viele weitere verwendet werden. (Für Interessierte und alle die Profis in diesem Bereich werden wollen: „UML 2 glasklar" gibt einen guten Überblick).
Gerade im Kontext des **Test-Driven-Development** bietet es sich auch an, bereits in dieser Phase Tests zu konzipieren, die die Integration des Systems in bestehende Strukturen testet!

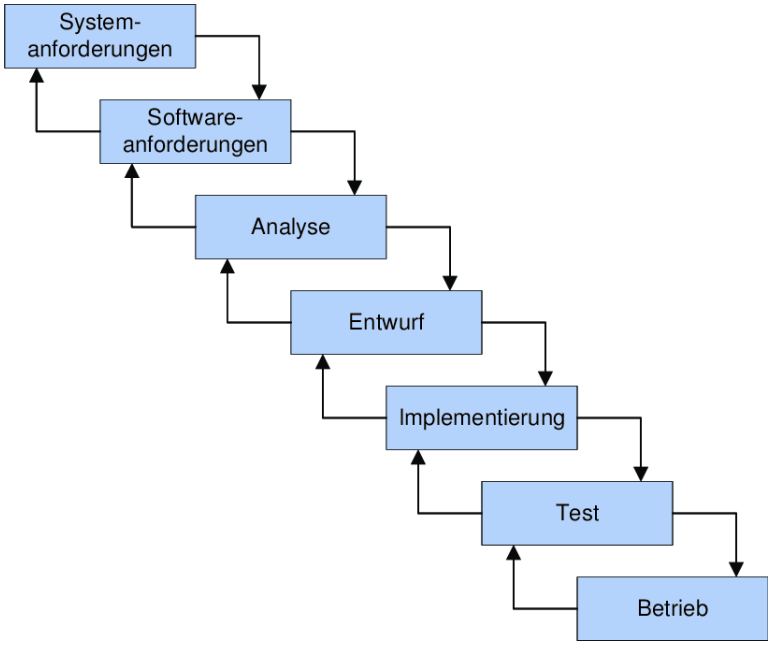
Implementierung und Integration
In dieser Phase geht es um die konkrete Umsetzung des zuvor beschriebenen, einzelne Komponenten der Software werden implementiert, getestet und danach zu größeren Einheiten zusammengebaut. In der Regel sollte dabei **gut dokumentiert werden (:))** und nach einer vorher festgelegten standardisierten Form gearbeitet werden.

Betrieb und Wartung sowie Stilllegung
Dieser Bereich des Lebenszyklus ist für uns aktuell eher uninteressant, da im Projekt (vermutlich?) keine kommerzielle Software entstehen wird. Grundsätzlich muss hier aber natürlich geregelt sein, ob der Auftragnehmer auch in

Zukunft die Einsatzfähigkeit garantieren muss (z.B. bei Einführung neuer Betriebssystem o.Ä.) oder generell: wann endet der Support für die Software?

1.3 Das Wasserfallmodell

Wie der Name schon suggeriert handelt es sich hier um ein lineares Modell, bei dem - zumindest in der ursprünglichen Variante - alle Phasen der Entwicklung streng sequentiell nacheinander ablaufen. Es gibt für jede Phasse festgelegte Start- und Endzeitpunkte. Die im vorangegangen Kapitel beschriebenen Phasen orientieren sich auch an den Phasen des ursprünglichen Wasserfallmodells bzw. umgekehrt. Man kann hier also von der „klassischen Vorgehensweise“ sprechen.
Eine modernere Variante, die wiederholende Elemente zulässt wird z.B. so dargestellt:

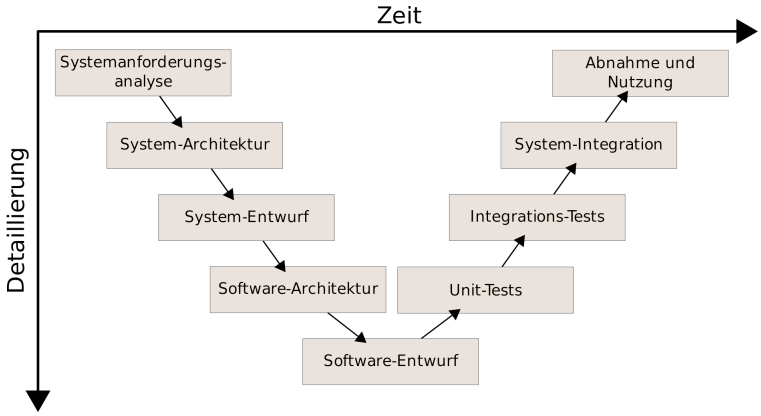


Quelle: Researchgate

Damit geht aber der klare Aufbau verloren, für iterative (sich wiederholende) bzw. dynamische Prozesse sind eher die agilen Methoden geeignet.

1.4 Das V-Modell

Das V-Modell stellt eine Erweiterung des Wasserfallmodells dar, es handelt sich im wesentlichen aber immer noch um ein lineares Modell:



Quelle: Wikipedia

Der wesentliche Unterschied zum Wasserfallmodell ist dabei, dass in den einzelnen Ebenen die entsprechenden Tests, die zu diesem Abschnitt gehören, auf der rechten Seite parallel dargestellt werden, d.h. die System-Architektur wird beispielsweise mit System-Integrationstests validiert.

1.5 Agile Methoden - Scrum

Die für uns relevanteste Vorgehensmodelle sind die **agilen Methoden** bzw. **agile Softwareentwicklung**. Im Gegensatz zu den bisherigen Modellen wird hier die Entwurfsphase möglichst kurz gehalten, d.h. die Entwicklung startet so früh wie möglich. Das bedeutet natürlich, dass die Anforderungen in der Regel nicht vollständig bekannt sind. Das vollständige Lasten- bzw. Pflichtenheft wird dann häufig durch sogenannte **User stories** ersetzt (siehe Scrum unten). Alle agilen Methoden basieren auf dem **agilen Manifest** (auch nicht mehr neu: formuliert 2001), eine Zusammenstellung einiger Aussagen, das folgendermaßen lautet:

Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbt tun und anderen dabei helfen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:

- **Individuen und Interaktionen** mehr als Prozesse und Werkzeuge
- **Funktionierende Software** mehr als umfassende Dokumentation
- **Zusammenarbeit mit dem Kunden** mehr als Vertragsverhandlung
- **Reagieren auf Veränderung** mehr als das Befolgen eines Plans.

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.
Quelle: agiles Manifest

Dieses Manifest wird um zwölf Prinzipien ergänzt, die hier zusammengefasst sind.
Wie bereits eingangs erwähnt sind agile Methoden dennoch nicht der heilige Gral der Software-Entwicklung, je nach Situation bzw. Anforderung kann es auch sinnvoll sein, die zuvor bereits erwähnten Modelle zu verwenden.

Scrum
In unseren Projekten soll eine abgespeckte Variante des Vorgehensmodells Scrum verwendet werden, deswegen folgt hier eine Skizze dieser Methode.

Grundsätzlich gibt es im Scrum-Framework drei **Rollen**, die Personen innerhalb der Entwicklerfirma einnehmen können:

1. **Scrum-Master**: der Scrum Master ist selbst kein Entwickler, sondern eher eine Art Manager, er ist dafür verantwortlich, dass die Regeln eingehalten werden und Kommunikation bzw. Zusammenarbeit reibungslos verläuft. Er ist außerdem für die Behebung von Konflikten zuständig und fungiert als Moderator bei Meetings. Der Scrum-Master ist insgesamt eher als Coach zu verstehen, sobald ein Team den Scrum-Ablauf verinnerlicht hat ist er im Wesentlichen nicht mehr notwendig. Er hat auch keine beurteilende Funktion (ist also insbesondere nicht der direkte Vorgesetzte der Entwickler).
   Bei uns: Unsere Projekte sind mit 3 beteiligten Personen so klein, dass die Rolle des Scrum-Masters nicht explizit besetzt werden muss.

2. **Product Owner**: Achtung: Der Product Owner ist nicht der Kunde, der den Auftrag für die Software gegeben hat, sondern ein Mitglied der Entwicklerfirma! Er ist aber innerhalb der Firma für das Produkt verantwortlich, d.h. er gestaltet z.B. die **User stories**, die gewünschtes Verhalten der Software beschreiben, in Zusammenarbeit mit den Kunden. Er erstellt daraus das **Product Backlog**, das Grundlage für die **Sprints** ist (siehe unten). Er entscheidet dabei (im Idealfall) auch, was in welcher Reihenfolge implementiert werden soll.
   Bei uns: auch diese Rolle ist in unserem Fall nicht explizit notwendig, die Entscheidungen über das „Produkt“ sollten gemeinschaftlich getroffen werden.

3. **Entwickler**: wie der Name schon suggeriert implementieren die dem Projekt zugeordneten Entwickler die vom Product Owner erstellten Anforderungen. Dabei sind sie (im Idealfall) völlig frei, wie die entsprechende Software umgesetzt werden soll, solange sie die vereinbarten Qualitätsstandards einhalten. Das Entwickler-Team

sollte für seine Arbeit keine äußere Hilfe mehr benötigen. Die interne Struktur des Teams ist ebenfalls nicht festgelegt. In der Regel besteht ein Scrum-Team aus höchstens 10 Personen, um den Koordinierungsaufwand überschaubar zu halten.
Bei uns: Das Team besteht bei uns immer aus drei Entwicklys, denen jeweils einer der drei Bereiche aus dem MVC-Modell zugeordnet sind (siehe Kapitel MVC und Das Projekt!).

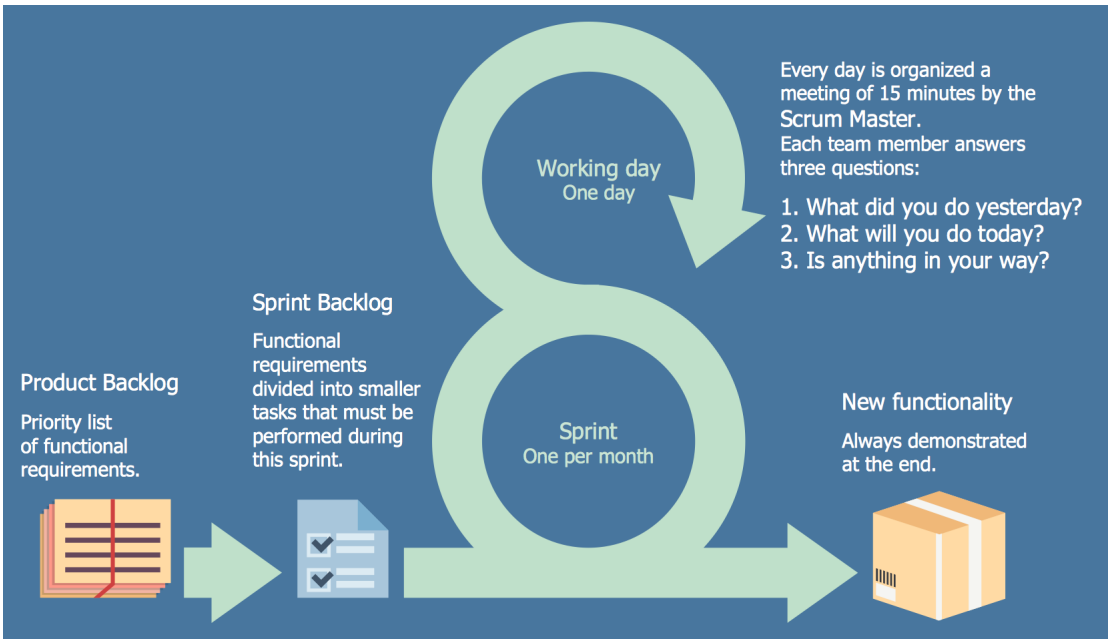
Natürlich wird Software auch von Personen außerhalb der Entwickler-Firma beeinflusst, diese Personen werden in Scrum **Stakeholder** genannt. Im Wesentlichen gibt es auch hier drei zentrale Gruppen:

1. **Kunden**: selbsterklärend, der Product Owner ist für den Kontakt zuständig.
   Bei uns: ihr seid eure eigenen Kunden! :)

2. **Anwender**: die Personen, die die Software tatsächlich verwenden, dieser Personenkreis kann, muss aber nicht mit den Kunden identisch sein. Sie sind eine Feedback-Quelle für das Entwickler-Team.
   Bei uns: z.B. Personen aus anderen Gruppen, die testen.

3. **Management**: in großen Firmen wichtig, für uns gänzlich uninteressant.

Da die Rollen innerhalb eines Teams nun geklärt sind folgt eine Übersicht des workflows:



Quelle: conceptdraw

Das vom Product Owner erstellte Product Backlog wird im **Sprint Backlog** in kleinere Aufgaben aufgeteilt, die während dieser Arbeitseinheit erledigt werden sollen. Eine solche Arbeitseinheit dauert in der Regel zwischen 2 und 4 Wochen und wird **Sprint** genannt. Die Entwickler einigen sich darauf, wie viel des Product Backlogs in diesem Sprint abgearbeitet werden kann
Wichtig dabei ist aber, dass nicht jeder der Entwickler die gesamte Zeit alleine vor sich hinwerkelt, sondern es gibt an jedem Arbeitstag ein Meeting (der **Daily Scrum**) an dem die drei oben abgebildeten Fragen beantwortet werden, frei übersetzt:

1. Was ist gestern passiert?

2. Was wird heute passieren?

3. Gibt es Probleme?

Diese Besprechungen sollen **kurz** und **effizient** sein - was sich in der Praxis häufig als schwierig heraustellt (hört man zumindest..). Am Ende eines Sprints gibt es - idealerweise - eine neue Version der Software bzw. ein neues Feature, das getestet werden kann im sogenannten **Sprint Review**. Hier sind häufig auch Kunden und insbesondere Anwender beteiligt. Der Sprint wird mit dann mit der **Sprint-Retrospektive** beendet - im Wesentlichen eine

Team-interne Besprechung, inwiefern in Zukunft effizienter und effektivergearbeitet werden kann.

Hinweis: Der ganze Prozess kann natürlich noch viel detaillierter dargestellt werden und es gibt auch noch deutlich mehr Details, für uns genügt aber dieser Überblick!

Bei uns: In der Schule benötigen wir die volle Pracht des Scrum-Workflows natürlich nicht, dennoch sollen Kernpunkte dieser Arbeitsweise in die Projektarbeit mit einfließen (Details und Beispiele dazu im Kapitel „Das Projekt"):

- Das Product Backlog (also die Beschreibung des zu entwickelnden Programs) wird in Form von **User Stories** zusammengestellt. Eine User Story ist dabei eine möglichst genaue Beschreibung in Alltagssprache, was das Programm können soll.
- Die Sprint-Dauer wird auf 1 bis 1,5 Wochen verkürzt, da das gesamte Projekt voraussichtlich nur eine Dauer von etwa 8 Wochen haben wird.
- Der Daily Scrum ist nicht möglich, es soll aber mindestens in jeder Unterrichtsstunde eine Besprechung des Teams geben - also nicht nur „basteln".
- Auch der Sprint Review soll möglichst während der Unterrichtszeit stattfinden (voraussichtlich Donnerstags, Ende der 3. Stunde) - um Tests auch mit nicht-Teammitgliedern zu ermöglichen.
- Vereinbarungen (also z.B. die User Stories) oder Implementierungsdetails sollen **schriftlich** festgehalten werden. Auch die Priorisierung - also was soll zuerst implementiert werden - soll deutlich gemacht werden.

2 Nebenläufige Prozesse

2.1 Parallele Prozesse allgemein

Neben den grundlegenden inhaltlichen Herausforderungen, die das Programmieren mit sich bringt, ist die gleichzeitige Verwendung von **Ressourcen** bei der Arbeit im Team noch eine zusätzliche Schwierigkeit. Das beginnt beispielsweise schon damit, dass in der Regel nicht zwei Entwickler an der gleichen Datei arbeiten können, wenn sie sich einfach irgendwo in einem Ordner befindet.
Die Beschäftigung mit solchen Problemen ist dabei so wichtig (und hat auch technische Auswirkungen), dass sich eine eigene Theorie darum aufgebaut hat. Für das Abtiur müssen die Grundlagen dieser Theorie ebenfalls verstanden werden. Für das konkrete Projekt finden sich Anmerkungen - z.B. zur Versionskontrolle und der effektiven Zusammenarbeit mit verschiedenen Tools - wie immer im Kapitel „Das Projekt“. Hier soll es um die theoretischen Hintergründe und Fachbegriffe gehen.
Als Ausgangslage stellen wir uns zwei Prozesse vor, die in irgendeiner Form in **Konkurrenz** zueinander stehen. Beispielsweise zwei Programme, die beide auf bestimmte Daten im Arbeitsspeicher zugreifen wollen.
Die Probleme liegen hier auf der Hand, wird der Bereich des Arbeitsspeichers von einem Program bearbeitet, während das andere ebenfalls bereits darauf zugegriffen hat, so entstehen Inkonsistenzen und Fehler.
Erinnerung: Das Problem ist bereits aus der Mittelstufe von Datenbanken bekannt, dort wurden Anomalien definiert. Im Folgenden ein Auszug aus einem Arbeitsblatt (der Text bezieht sich auf eine Tabelle Bestellung, in der Informationen zu Kunden und gekauften Artikeln vorhanden sind):
„... Es gibt jedoch noch weitere Probleme (auch Anomalien genannt), die durch Redundanzen entstehen, z.B. die UPDATE-Anomalie.
Wird einer oder mehrere Einträge einer Tabelle geändert, so spricht man von einem update (da der zugehörige SQL-Befehl ebenfalls UPDATE heißt!). Offensichtlich ist z.B. eine Namensänderung eines Kunden - beispielsweise durch Heirat - hier sehr problematisch, da jede einzelne seiner Bestellungen geändert werden muss, ansonsten ist die Datenbank inkonsistent. Das bedeutet, dass sie Widersprüche enthält.

Möchte man etwas aus einer Tabelle löschen benutzt man den SQL-Befehl DELETE. Möchte ein Kunde aus dem System gelöscht werden, so muss auch hier jede einzelne seiner Bestellungen gelöscht werden, um seine Daten vollständig zu entfernen. Dabei kann leicht etwas vergessen werden. Zusätzlich besteht die Gefahr auch die Informationen über einen Artikel vollständig zu löschen, wenn die Bestellung die einzige zu diesem Artikel war. Im zweiten Fall spricht man auch von der DELETE-Anomalie.

Angenommen, man möchte einen neuen Artikel zum Sortiment hinzufügen, so ist dies mit obigem Tabellenschema nicht ohne weiteres möglich. Zu jedem Artikel muss eine Bestellung existieren!

Auf den ersten Blick scheint es möglich, bei allen Werten einfach Null, also „nichts“ einzutragen, die bisher nicht bekannt sind. Dies wird jedoch durch den Primärschlüssel verhindert, der hier aus KdNr, ArtNr und BDatum besteht. Es muss also eine fiktive Bestellung zwangsweise angelegt werden, um den Artikel aufnehmen zu können, es kommt zu einer INSERT-Anomalie, also zu einer Anomalie bzw. Inkonsistenz beim Einfügen neuer Daten. ...“

Die Lösung dieser Probleme bestand darin, die Tabellen der Datenbank geschickt zu basteln, sodass diese Anomalien möglichst nicht mehr vorkommen. Dennoch bleiben Probleme, es könnten z.B. zwei Benutzer gleichzeitig versuchen einen Datenbankeintrag zu ändern, das ist die Stelle an der wir jetzt ansetzen - ein gutes Design reicht nicht mehr aus.
Statt miteinander zu konkurrieren, sollen die Programme miteinander **kooperieren**, d.h. ihre Aktionen bewust aufeinander abstimmen - in der Praxis kann dies natürlich nicht in der Interaktion mit jedem anderen Program passieren, deswegen übernimmt die Aufgabe der Koordinatione der Kooperation in der Regel eine übergeordnete Instanz, z.B. das Betriebssystem oder bei den erwähnten Datenbanken das Datenbankmanagementsystem (DBMS).
Zur Lösung des Problems wird das sogenannte **Semaphor-Prinzip** verwendet. Ein **Semaphor** war ursprünglich ein mechanisches Eisenbahnsignal, das anzeigt, ob ein Zug einen Gleisabschnitt belegt. Folgt man dieser Logik, so ist ein Semaphor im Wesentlichen eine Datenstruktur, die verwaltet, ob auf eine bestimmte Ressource gerade zugegriffen wird oder nicht.

Abbildung 1: Quelle: Wikipedia

Im einfachsten Fall besteht sie einfach aus einem Wahrheitswert oder einer Ganzzahl, die den Zustand der Ressource beschreibt, einer Methode „Reservieren()“, die effektiv den Zugriff für alle anderen sperrt und „Freigeben()“, die die entsprechende Ressource wieder entsperrt.
Der Semaphor kann auch erweitert werden, sodass er als Warteschlange fungiert, z.B. könnte man den obigen Zähler erweitern und speichern, wie viele Prozesse gerade „warten“. **Beispielbereiche für die Anwendung des Semaphor-Prinzips**

1. **DBMS**
2. **Druckerwarteschlangen**
3. **Dateizugriff**
4. **Verwaltung von Threads**: Auch bei Software an sich möchte man Parallelsierung nutzen, da es häufig Fälle gibt, bei denen ähnliche Dinge „gleichzeitig“ ablaufen sollen bzw. berechnet werden, dazu werden **Threads** verwendet.

Der letzte Punkt bringt uns direkt zum nächsten Thema:

2.2 Parallele Prozesse in Java

In den letzten zwei Jahrzehnten sind die Anforderungen an Prozessoren immens gewachsen, da immer größere Probleme mit Hilfe von Computern gelöst werden sollen, einige Beispiele:

1. Klimasimulationen
2. Strömungsmechanik
3. Modellierung allgemein
4. Animierte Filme
5. Computerspiele
6. Web-Suche (das Internet ist riesig! :)

Obwohl der Film schon etwas älter ist, sind die Zahlen bei Baymax beeindruckend:

Quelle: disney.fandom

Der Film wurde auf einem Cluster von 4.600 Rechnern mit insgesamt 55.000 Kernen erstellt mit einer kumulierten Rechenzeit von 200 Millionen Stunden. Das entspricht einer Rechenzeit von etwa 83 Stunden pro Frame!
Ohne Parallelisierung - d.h. ein einziger Kern rendert den kompletten Film - wäre solch ein Mammutprojekt nicht denkbar.
Hinweis: Tatsächlich wäre eine reine Parallelisierung immer noch nicht ausreichend, um die gigantischen Anforderungen des Rendering abzufangen, die größten Verbesserungen kommen durch die Verwendung intelligenter Algorithmen, die Parallelisierung schadet aber natürlich nicht.

An dieser Stelle könnte man einen langen Exkurs einfügen, wie wichtig Parallelität in den meisten Bereichen der Technik inzwischen ist, dafür haben wir hier aber keine Zeit.

Umsetzung in Java
Die Umsetzung in Java wird uns noch in der zwölften Klasse ausführlich beschäftigen, hier ein kleiner Vorgeschmack:

```
public class Threading{
    public static void main(String[] args) {
        // Wir erzeugen die Threads und geben ihnen eine Aufgabe
        Thread t1 = new Thread(new Task());
        Thread t2 = new Thread(new Task());

        // Wir starten die Threads.
        t1.start();
        t2.start();

        // Wir warten darauf, dass beide Threads fertig sind.
        try {
            t1.join();
            t2.joint();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Task finished");
    }
    }

    class Task implements Runnable {
    @Override
```

```
    public void run() {
        // Irgendeine Aufgabe wird ausgeführt.
        System.out.println("Task executed by " + Thread.currentThread().getName());
    }
}
```

Die Verwendung von Threads ist in Java grundsätzlich nicht schwer, da bereits eine vorgebaute Klasse **Thread** existiert, die dem entsprechenden Betriebssystem mitteilt, dass möglichst ein zweiter Kern genutzt werden soll. Natürlich ergibt sich hier aber ein analoges Problem zum vorigen Kapitel, wenn die Aufgabe, die jeder Thread ausführt auf gemeinsame Ressourcen zugreifen muss, so kann es zu den bereits besprochenen Problemen kommen. Wie Java diese Probleme löst ist Stoff der 12. Klasse.

3 Model View Controller

3.1 Entwurfsmuster

Quelle: lehrer-online

Wäre dies ein Skript zu einem anderen Fach, so wäre die erste Aufgabe die Analyse des obigen Comics. Glücklicherweise ist das in Informatik nicht notwendig - die Kernaussage ist klar. Zwar ist es durchaus nützlich für das Verständnis, Dinge selbst zu programmieren, aber Zeit ist eine begrenzte Ressource. Deswegen ist es unerlässlich, dass auf die Arbeit von von anderen Personen zurückgegriffen wird. Möchte man fremden Code nicht direkt kopieren - oder kann das nicht, weil er doch nicht zu 100 Prozent auf den eigenen Fall zutrifft - so kann man sich zumindest an der Art und Weise der Problemlösung orientieren.
Da es Problemklassen gibt, die häufig ein ähnliches Vorgehen erfordern, wurden bereits Lösungsschablonen entwickelt, die im Allgemeinen **Entwurfsmuster** heißen. Das bedeutet, dass ein grobes Klassendiagramm der Struktur bereits vorhanden ist, eine konkrete Ausgestaltung ist aber dem Programmiery überlassen.
Wir haben bereits mit dem **Kompositum** ein Entwurfsmuster kennengelernt, auch dort haben wir nicht direkt „**Das Kompositum**“ implementiert, sondern die grundlegende Idee, die dahinter steckt, verwendet.
Es gibt noch unzählige weitere Entwurfsmuster, hier eine kleine Auswahl, jeweils mit kurzer Erläuterung:

- **Singleton**: stellt sicher, dass von der betrachteten Klasse nur ein einziges Objekt erzeugt werden kann und bietet eine globale Zugriffsmöglichkeit auf dieses Objekt. Z.B. braucht ein Login-System nur eine Instanz (dieses soll alle Logins verwalten!).
- **Factory**: Abstrahiert die Erstellung von Objekten, indem sie eine gemeinsame Schnittstelle für die Erstellung von Objekten definiert und die spezifische Omplementierung der Objekterstellung in weitere Unterklassen auslagert. Dieses Muster kann z.B. bei der Erstellung von Spielfiguren o.Ä. nützlich sein.
- **Observer**: stellt eine Möglichkeit bereit, um auf Änderungen eines Objekts zu reagieren, indem andere Objekte (die Observer) benachrichtigt werden. Ein klassischer Anwendungsfall wäre ein Benachrichtigungsticker (siehe auch Benachrichtigungen am Handy!).
- **Decorator**: ermöglicht die dynamische Erweiterung von Objekten durch Hinzufügen von zusätzlichen Funktionen oder Eigenschaften (die Dekorationen). Ein Bild kann z.B. mit Filtern „dekoriert“ werden.
- **Adapter**: ermöglicht die Integration von inkompatiblen Klassen durch die Entwicklung einer gemeinsamen Schnittstelle (sehr allgemeines Konzept!).
- **Iterator**: es wird eine Möglichkeit bereitgestellt, sequentiell auf Elemente in einer Zusammenstellung zuzugreifen, z.B. in einer Liste.

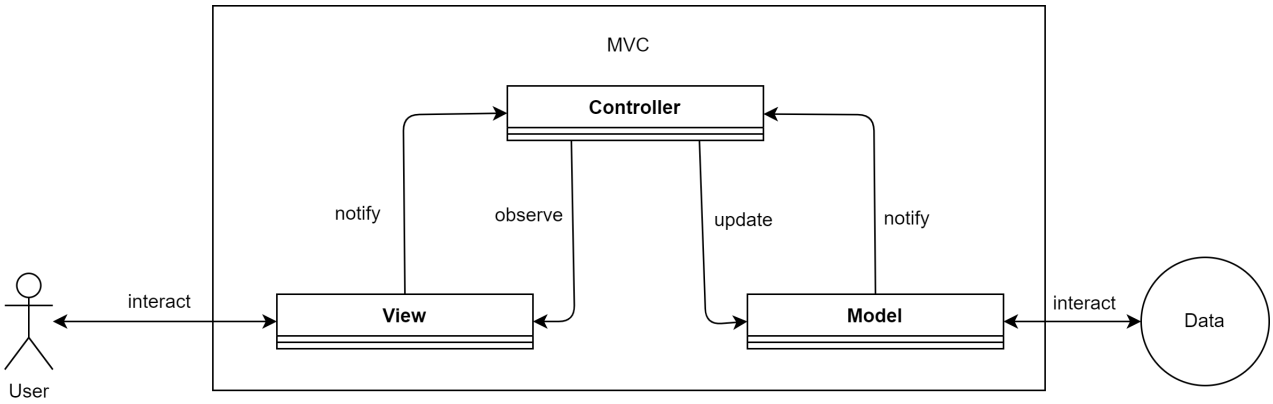
Das Muster, das für unser Projekt zentral sein soll, ist noch etwas allgemeiner als die Meisten der oben bereits erwähnten - **Model View Controller**. Es kann sich nicht nur auf einzelne Klassen beziehen (auch wenn das durchaus möglich ist), sondern beschreibt die grundlegende Implementierungsidee einer ganzen **Softwarearchitektur**.
Bevor wir uns Beispielen zuwenden ein kurzer Überblick über die drei Komponenten:

1. **Modell (Model)**: Das Modell beschreibt die konkreten Daten, die mit dieser Anwendung verknüpft sind. Die Daten können dabei in vielerlei Form vorliegen, von definierten Attributen in einfachen Klassen über Datenformate wie **JSON** bis hin zu ganzen Datenbanken. Hier kann gegebenenfalls auch noch Logik implementiert sein, wenn die Daten, z.B. bei Speicherung, noch verarbeitet werden müssen.

2. **Ansicht (View)**: Die Ansicht ist dazu da, die Daten des Modells in einer Form darzustellen, sodass der Anwender mit ihnen interagieren kann. Letztendlich läuft es im modernen Kontext in der Regel auf ein **Graphical User Interface (GUI)** hinaus. GUIs verwenden häufig das Entwurfsmuster Kompositum, da sie aus Komponenten mit ähnlichen Grundfunktionen (z.B. Menüeintrag, klickbares Objekt, Datendarstellung etc.) zusammengesetzt werden.

3. **Steuerung (Controller)**: Die Steuerung verwaltet die anderen beiden Komponenten. In der Regel wird das Entwurfsmuster „Observer“ verwendet, um auf Interaktion des Anwenders mit der Ansicht zu „**lauschen**“. Anschließend veranlasst die Steuerung alle notwendigen Änderungen, so werden beispielsweise neue Menüeinträge angezeigt, oder neue Daten werden von einem Eingabefeld in der

In einem sehr vereinfachten Schema mit jeweils einer Klasse dargestellt:



Ein mögliches Grundgerüst der Implementierung dieses Diagramms sieht so aus:

```
//Unser Model entspricht hier einfach nur einer Datenklasse, die Zugriff und
//Änderungen regelt.
public class Model {
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}

public class Controller {
    //Der Controller hat Referenzen auf ein Model und einen View
    private Model model;
    private View view;

    //Der Controller könnte Model und View im Konstruktur auch direkt
```

```
    //erzeugen - man könnte hier auch das Singleton-Pattern verwenden,
    //wenn man sichergehen möchte, dass in einer Instanz des Programms
    //nur ein Controller existieren kann!
    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
    }

    //Um Daten zu modifizieren (z.B. durch eine Eingabe im View getriggert)
    //mus der Controller auch auf die Methoden des Models zugreifen:
    public void setData(String data) {
        model.setData(data);
    }

    public String getData() {
        return model.getData();
    }

    //Der View wird mit dieser Methode aktualisiert, d.h. hier:
    //Die aktuellen Daten werden auf der Konsole ausgegeben.
    public void updateView() {
        view.printData(model.getData());
    }
}

//Der View ist hier am einfachsten:
//Er gibt einfach nur Daten auf der Konsole aus.
public class View {
    public void printData(String data) {
        System.out.println("Data: " + data);
    }
}
```

Diese Implementierung ist noch recht abstrakt, geben wir den Klassen ein klein wenig mehr Leben:

```
//Das Model hat sich nicht wesentlich verändert, es soll jetzt aber
//einen Spielstand modellieren statt einfach nur eines Textes
public class Model {
    private int score;

    public int getScore() {
        return score;
    }

    public void increaseScore() {
        score += 10;
    }
}

import java.util.Random;

public class Controller {
    private Model model;
    private View view;
    //Um Zufallszahlen zu erzeugen
    private Random random;
```

```
    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
        random = new Random();
    }

    public void performAction() {
        //In diesem kleinen Beispiel wird eine Zufallszahl erzeugt, ist
        //diese gerade, so haben wir "gewonnen" und der Score wird erzeugt
        int randomValue = random.nextInt(10) + 1;
        if (randomValue % 2 == 0) {
            model.increaseScore();
            //außerdem wird angezeigt, dass wir Erfolg hatten!
            view.printSuccessMessage();
        } else {
            //andernfalls wird die Versagens-Meldung angezeigt - Schade :)
            view.printFailureMessage();
        }
        //In jedem Fall wird der derzeitige Spielstand ausgegeben.
        view.printScore(model.getScore());
    }
}

//Der View interagiert weiterhin nur mit der Konsole:
public class View {
    public void printSuccessMessage() {
        System.out.println("Great job! You earned 10 points.");
    }

    public void printFailureMessage() {
        System.out.println("Sorry, try again.");
    }

    public void printScore(int score) {
        System.out.println("Your current score is: " + score);
    }
}
```

Ausgehend von diesen sehr grundlegenden Beispielen können jetzt die einzelnen Klassen sukzessive ausgebaut werden - es muss auch nicht zwingend bei diesen drei Klassen bleiben! Z.B. könnten die Daten in mehreren Klassen organisiert sein. Wollen wir beispielsweise verschiedene Karten modellieren, könnte eine Oberklasse Karte verwendet werden und für die einzelnen speziellen Kartenarten jeweils eine Unterkarte, also z.B. wie bei Uno:

1. Zieh Vier Farbwahl
2. Aussetzen
3. Richtungswechsel
4. Zieh 2
5. Zahlenkarte

Alle erben von Karte, da sie z.B. alle ausgepielt werden können, die Eigenschaften unterscheiden sich aber und sie können spezielle Effekte haben.
Anmerkung: Weitere Hinweise zur weiteren Ausdifferenzierung bzw. Ausgestaltung der einzelnen Komponenten finden sich im nächsten Kapitel.

4 Das Projekt

4.1 Rahmenbedingungen

Und jetzt zur eigentlichen Aufgabe, in einem Satz:

> **Aufgabe - das Projekt**: Entwickeln Sie in Dreierteams ausgehend von einer eigenen Problemstellung mit Hilfe agiler Methoden ein Programm, das sich am Architekturmuster MVC orientiert.

Wichtige Punkte vorweg:

- Voraussichtliches Startdatum: 2.5.23
- Voraussichtliches Enddatum: 27.6.23
- ersetzt kleinen praktischen Leistungsnachweis
- Dreiergruppen (keine Ausnahmen)
- Abschlusspräsentation/Vorführung des Programs bzw. des aktuellen Standes
- Jeder in der Gruppe muss beitragen ⇒ Jeder ist verantwortlich für einen der drei Bereiche Model, View oder Controller. Das bedeutet **nicht**, dass zwangsläufig alles in diesem Bereich von derselben Person programmiert werden muss.
  Aber: Nachfragen im konkreten Bereich müssen vom Verantwortlichen in der Abschlusspräsentation beantwortet werden.
- Der Code des Projekts muss **ausführlich kommentiert** werden. Im Zuge der Bewertung ist diese Vorgabe notwendig, auch wenn sie dem agilen Manifest widerspricht. Desweiteren müssen grob die von den einzelnen Teammitgliedern bearbeitenden Aufgaben klar werden (Details bei **User Stories**, **Git** oder Punkt 4 des Fahrplans). Als Ausgleich muss **keine weitere** Dokumentation der **Arbeitsweise** in der Gruppe (z.B. als Portfolio) erstellt werden.
- **Für die Experten**: Die Programmiersprache muss nicht zwingend Java sein, es muss aber in der Gruppe Konsens bezüglich der Sprache herrschen. Bei geplanter Abweichung von Java zunächst auch Rücksprache mit der Lehrkraft notwendig. Der Fokus liegt auf **Objektorientierung**.
- Es sollen **mindestens** für dei wichtigsten Funktionen, in jedem Fall aber für die gesamte Anwendung Tests vorhanden sein.
- Die Zeit während des Unterrichts reicht (in den meisten Fällen) nicht aus für ein sehr gutes Ergebnis!

Fahrplan

1. **Ideensammlung**: Das Team überlegt sich gemeinsam welche Art von Program erstellt werden soll und **einigt sich** auf ein grobes Ziel.
2. **Erste Erstellung von User Stories**: Zuerst sehr grob: „Was soll das Programm können?“. Für Details siehe entsprechendes Unterkapitel.
3. **Besprechung und Entwurf**: Besprechung mit Lehrkraft ⇒ Absegnung des Projekts, parallel: Beginn der groben Planung, z.B.: wird eine Datenbank benötigt, welches Framework für die GUI, etc., grobes Klassendiagramm der benötigten Komponenten (siehe auch **Model**, **View**, **Controller**)
4. **Arbeitsaufteilung - Organisation Sprint**: Konkretisierung der User Stories, gegebenenfalls Aufdröselung in die drei Teilgebiete, Erstellung von Aufgaben (**Issues** oder **Tasks**), die im nächsten Sprint abgearbeitet werden sollen, sowie deren Priorisierung.
   Die Aufgaben können entweder handschriftlich fixiert werden oder entsprechende Tools verwendet werden, z.B. **Trello**,**Notions** direkt **Github**, oder andere tools.
5. **Sprint**: Eigentliche Implementierungsarbeit, dabei immer wieder Austausch am Beginn der Unterrichtszeit. (**Daily Scrum** - **Standup-Meeting**)

6. **Sprint-Retrospektive und Neudefinition**: Etwa alle 1,5 Wochen: Rückblick - was ist geschafft? Neue Aufgaben verteilen.
7. **Arbeit!** Wiederholung von 4 bis 6.
8. **Abschlusspräsentation**: Vorstellen des eigenen Programms, Rückfragen zum Code beantworten.

4.2 Bewertungskriterien

Bewertet werden folgende Kategorien, jeweils mit Beispielen:

1. **Generelle Umsetzung der Vorgaben (Doppelte Gewichtung)**: wurde tatsächlich MVC als Grundlage verwendet, wurde im Team gearbeitet oder sind es drei Einzelprojekte, wurde der Fahrplan grundlegend umgesetzt, ...
2. **Komplexität und Umfang des Programs**: wurde TicTacToe oder doch eher Frogger implementiert, besteht nur eine einfache Funktionalität oder mehrere/eine komplexe, ...
3. **Umsetzung und Design**: wurde „BruteForce“ gecodet oder Konzepte verwendet, wurden die Ziele umgesetzt oder fehlen Aspekte, ...
4. **Dokumentation**: selbsterklärend :)
5. **Arbeit in der Gruppe und im Team**: wurde über Struktur und Implementierung diskutiert, wurde kollaborativ gearbeitet, ...
6. **Abschlusspräsentation**: wurde das Programm vollständig vorgestellt, wurden alle Nachfragen korrekt beantwortet, war der Vortragsstil ansprechend, ...

4.3 Git und Github

Um Probleme wie Überspeicherung, Datenverlust, etc. vermeiden, so bietet es sich an ein **Versionskontrollprogramm** zu verwenden, d.h. ein Programm, dass sich den Stand des Codes zu definierten Zeitpunkten „merkt“. Die populärste Variante ist heutzutage **Git**.
Wer nach der Schule professionell programmieren möchte, muss sich zwangsläufig mit Git auseinandersetzen. Im Rahmen dieses Skripts werden nur absolute Grundlagen dargestellt, da alles andere den Rahmen sprengen würde. Ausführliche Tutorials gibt es aber überall im Internet zu finden, ein paar Anlaufstellen:

- w3Schools Git Tutorial: textbasiertes Tutorial
- offizielles Dokumentationstutorial
- Videoreihe von Morpheus - Deutsch
- Allgemein: Google/ChatGPT/Youtube: ihr wisst wie das geht :)

4.3.1 Git

Die zentrale Verwaltungseinheit bei Git ist ein sogenanntes **Repository**. Hier werden alle Daten, die mit dem Projekt zu tun haben abgespeichert.
Hinweis: Git ist ein eigenes Programm und muss vor Verwendung installiert werden (**Windows-Installer**.
Git kann rein lokal verwendet werden, d.h man erzeugt z.B. ein neues Repository mit (Hinweis: die Dollarzeichen sind nur ein command prompt symbol und sollen **nicht** eingegeben werden!):

```
$ git init
Initialized empty Git repository in /path/to/repository/.git/
```

Es wird dadurch ein (versteckter) Ordner erzeugt, in dem git die Versionierung verwaltet (ihr könnt bei Interesse natürlich hineinsehen, ansonsten wird dieser Ordner aber nicht manuell bearbeitet).
Nehmen wir an, es wurde eine Datei *test.txt* im selben Ordner erzeugt, dann müssen wir git mitteilen, dass wir diese Datei „tracken“ wollen, d.h. ihre Änderungen sollen durch Git überwacht werden:

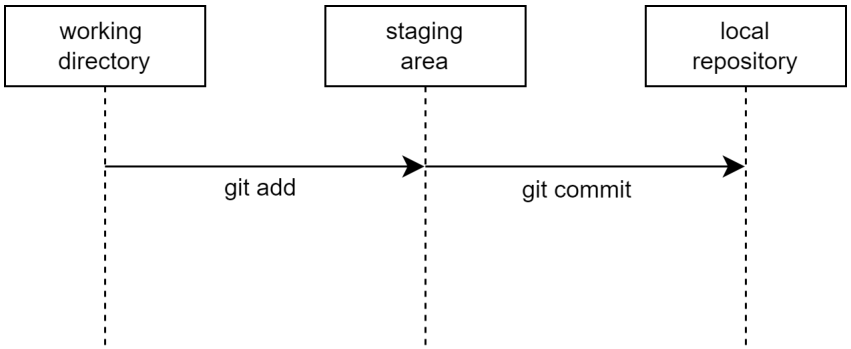
```
$ git add test.txt
```

Es wird jetzt noch eine weitere Datei „test2.txt“ erzeugt. Wollen wir Informationen über den aktuellen Status unseres Repositories einholen, können wir *gitstatus* verwenden:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test2.txt
```

Hier werden jetzt einige Informationen angezeigt, der Reihe nach:

- **No commits yet**: Ein Commit ist in Git eine Aktion, mit der Änderungen am Repository festgehalten bzw. gespeichert werden. Ein Commit enthält also alle Änderungen, die seit dem letzten Commit vorgenommen wurde. Wird etwas „commitet“, wird in der Regel eine Nachricht beigefügt, die beschreibt, um was es in diesem Commit ging.
- **Changes to be committed**: hier stehen alle Dateien, die seit dem letzten Commit geändert wurden **und** mit git add dem Tracking hinzugefügt wurde. Sie liegen jetzt in der sogenannten **staging area**:



  In der Staging Area sind alle Dateien zusammengefasst, die getrackt werden, diese können mit einem Commit dem Repository hinzugefügt werden. Das zeigt insbesondere eines: Nur weil eine Datei im selben Ordner liegt, ist sie noch nicht dem Repository hinzugefügt!

- **Untracked Files**: Dies teilt uns auch git status mit, test2.txt liegt zwar in unserem Ordner, wird aber noch nicht getrackt!
- **On branch master**: Dies zeigt uns an, auf welchem „**Zweig**“ wir uns gerade befinden, dazu später mehr.

Sind wir zufrieden mit unseren Änderungen, so können wir sie per **git commit** dem Repository hinzufügen, möchte man noch eine Commit Nachricht hinzufügen verwendet man:

```
$ git commit -m "Das ist unser erster Commit!"
[master (root-commit) 56f4a69] Das ist unser erster Commit!
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
```

Git teilt uns jetzt mit, dass wir auf dem Branch master(siehe unten) mit der Nachricht ”Das ist unser erster Commitëine Datei hinzugefügt haben. (Es wird also auch verändern - insertions und deletions - unterschieden von „neu hinzugefügt“).
Sehen wir jetzt wieder unseren Status an:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test2.txt
```

In der **Staging area** ist jetzt nichts mehr, deswegen ist die entsprechende Nachricht verschwunden.
Für die folgenden Überlegungen brauchen wir mehr als einen Commit, d.h. wir fügen jetzt in einem nächsten Schritt die zweite Datei auch noch hinzu und machen einen commit:

```
$ git add test2.txt
$ git commit -m "test2.txt hinzugefügt"
```

Würden wir wieder git status aufrufen würden wir sehen, dass es keine Änderungen gibt.
Soweit so gut, aber wie kommt die eigentliche **Versionskontrolle** ins Spiel? Um das besser zu verstehen brauchen wir einen weiteren Befehl:

```
$ git log
commit 999ae8f676f7ad55ae4011c2af17b6e079eafe76 (HEAD -> master)
Author: Wechsler <61123667+Wechsler@users.noreply.github.com>
Date:   Fri Apr 21 14:27:59 2023 +0200

    test2.txt hinzugefügt

commit 56f4a694ef905db95b3464371eaef65fcf12e0b2
Author: Wechsler <61123667+Wechsler@users.noreply.github.com>
Date:   Fri Apr 21 14:21:09 2023 +0200

    Das ist unser erster Commit!
```

Wir haben wieder viele Informationen hier verpackt:

- **Author und Date**: zu jedem Commit wird der Benutzer, der den Commit erstellt hat gespeichert, sowie Datum und Uhrzeit (in diesem Fall ist der Autor auch noch mit Github verknüpft, dazu später mehr).
- **commit <Hash>**: Jeder Commit besitzt einen eindeutigen Hash-Wert, der es ermöglicht dorthin zurückzuspringen. Dazu würde der Befehl git checkout <Hash> verwendet werden. Das „Springen“ durch einzelne Commits ist aber eine gefährliche Sache, da es - wenn man nicht aufpasst - doch zu Datenverlust oder Ähnlichem kommen kann. Es wird deswegen - insbesondere für Anfänger - nicht empfohlen.

Damit stellt sich die Frage, wie wir Git stattdessen sinnvoll für die Entwicklung verwenden können. Die Antwort darauf sind sogenannte **branches**, also Zweige.
Ein grundlegender Ablauf könnte dabei so aussehen:
(*Hinweis:* wer nicht so viel lesen möchte, der Ablauf ist auch in diesem **Video** dargestellt - zum Nachlesen ist das untenstehende Skript aber sicher nützlicher)

1. Das Repository ist in einem bestimmten - **funktioniereden** Zustand (in unserem Fall sind die Dateien test.txt und test2.txt vorhanden).
2. Es soll ein neues feature programmiert werden. Wir erzeugen einen neuen **branch** namens „development“ via:

   ```
   $ git branch development
   $ git status
   On branch master
   nothing to commit, working tree clean
   ```

   git status teilt uns mit, dass „master“ immer noch der aktuelle branch ist, d.h. wir haben zwar einen neuen branch erzeugt, aber sind noch nicht dorthin gewechselt, das wird ebenfalls mit dem checkout-Befehl realisiert:

```
$ git checkout development
Switched to branch 'development'
```

3. Jetzt können wir Ändeungen durchführen, z.B. eine weitere Datei hinzufügen und einen commit ausführen:

```
$ git add dev.txt
$ git commit -m "Commit auf branch development!"
[development b5f5db0] Commit auf branch development
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dev.txt
```

4. Unser Feature ist noch nicht fertig programmiert, aber wir müssen auf unserem master dringend einen Bugfix durchführen, d.h. wir kehren auf den master-branch zurück:

```
$ git checkout master
Switched to branch 'master'
```

Wirft man jetzt einen Blick in den entsprechenden Ordner, so ist die Datei „dev.txt“ verschwunden, da sie auf diesem branch noch nicht existiert! Wir führen unseren Bugfix aus - in diesem Fall wird nur eine Datei bugfix.txt erzeugt und hinzugefügt, dann ein commit erstellt:

```
$ git add bugfix.txt
$ git commit -m "wichtiger bugfix"
[master 6d569da] wichtiger bugfix
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bugfix.txt
```

5. Zufrieden kehren wir zu unserem branch development zurück:

```
$ git checkout development
```

Die Datei „dev.txt“ ist jetzt wieder vorhanden, nicht aber der Bugfix, wir wollen diesen aber auch auf dem development branch haben, da er sonst vielleicht unser feature beeinflusst. Der aktuelle Stand von Master soll also in den development-branch gebracht werden, das kann mit einem **merge** geschehen:

```
$ git merge master
Merge made by the 'recursive' strategy.
bugfix.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bugfix.txt
```

Achtung: Dies ist eine heikle Stelle, hier klappt alles und der branch wird einfach nur mit der „rekursiven Strategie“ (was das genau bedeutet soll uns an dieser Stelle egal sein) gemerged. In diesem Fall ist das nur das Hinzufügen einer weiteren Datei.
ABER: gibt es auf dem Master-branch Änderungen, die Änderungen auf dem development-branch in irgendeiner Form **widersprechen** - so gibt es einen **merge-conflict**, der händisch behoben werden muss, d.h. es muss der Quellcode der beiden Versionen verglichen werden und dann entsprechend geändert werden. Das ist unser **worst case** - merge-Konflikte sollten soweit wie möglich vermieden werden, indem auf unterschiedlichen branches nur an unterschiedlichen Stellen im Code gearbeitet wird - bzw. an unterschiedlichen features.

6. Wir schließen jetzt den development-Prozess unseres features ab, indem wir eine weitere Datei dev2.txt hinzufügen und commiten:

```
$ git add dev2.txt
$ git commit -m "feature fertig!"
[development 6047718] feature fertig!
 1 file changed, 0 insertions(+), 0 deletions(-)
```

```
 create mode 100644 dev2.txt
```

(Natürlich wurde das feature ausführlich getestet bevor so ein commit gemacht wird :)

7. Wir kehren auf den master-branch zurück:

```
$ git checkout master
Switched to branch 'master'
```

Erwartungsgemäß verschwinden dev.txt und dev2.txt wieder aus unserem Ordner. Wir können jetzt aber den Development-branch in unseren Master-mergen, wie wir es oben mit dem master-branch gemacht haben:

```
$ git merge development
```

Es finden sich jetzt alle 5 Dateien wieder in unserem Ordner, der Branch development existiert weiter, das können wir z.B. so verifizieren:

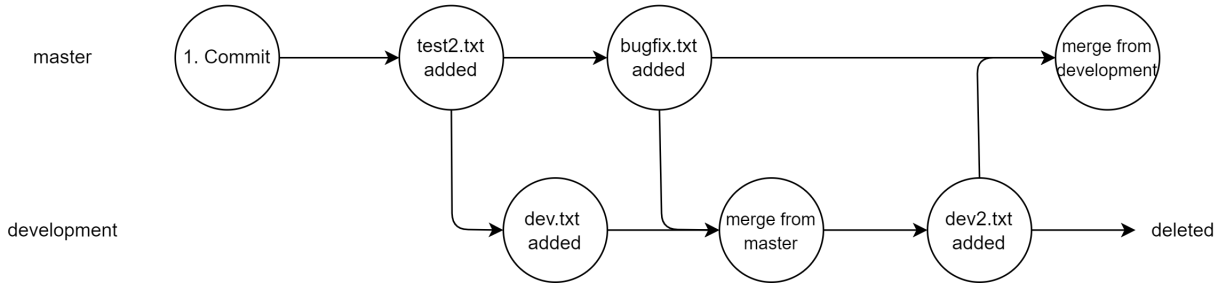
```
$ git branch
development
* master
```

Da die Entwicklung des features abgeschlossen ist brauchen wir den branch auch nicht mehr, gelöscht wird so:

```
$ git branch -d development
Deleted branch development (was 6047718).
```

Alles ist jetzt wieder aufgeräumt und bis zu unserem nächsten Feature kann alles so bleiben.

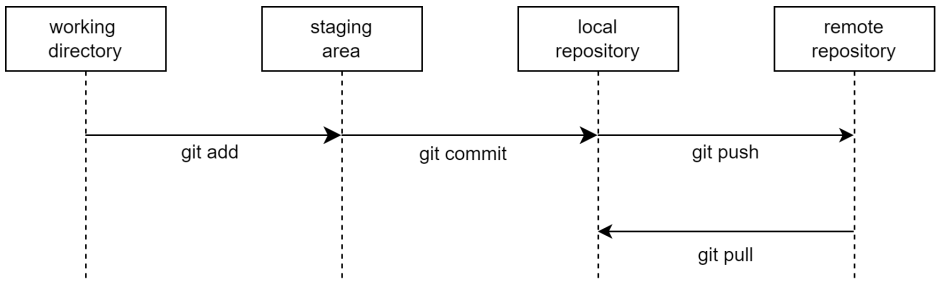
Der Prozess, der oben beschriebenwurde kann auch in einem Bild dargestellt und veranschaulicht werden, z.B. so:



Die gute Nachricht: Alles was zuvor besprochen wurde muss nicht zwingend händisch ausgeführt werden (man versteht so aber besser was eigentlich passiert!) - VsCode nimmt uns viel der Arbeit ab und bietet eine graphische Oberfläche mit der wir ein Repository verwalten können, Details dazu im Unterricht bzw. im oben schon erwähnten Video,

4.3.2 Github

Bisher ist die Versionsverwaltung zwar nützlich für uns, aber noch nicht wirklich für die Verwendung in der Gruppe geeignet, da alles nur lokal auf den eigenen Rechnern passiert. Dieses **Video** zeigt die Grundlagen von Github und wie damit kollaborativ gearbeitet werden kann.



Im Wesentlichen wird eine weitere Ebene hinzugefügt, das remote repository, das mit dem lokalen Repository via push und pull synchronisiert wird.

4.4 User Stories und Aufgaben

Nehmen wir an, das Projekt, das umgesetzt werden soll ist eine Form von Spiel, in diesem Fall Schach. Eine Beschreibung eines späteren Anwenders könnte dazu z.B. so ausfallen (im schlimmsten Fall wäre natürlich die Antwort: „Ich möchte Schach spielen können“, wir gehen aber von etwas detaillierteren Anforderungen aus):

1. Man soll in einem Menü eine Schwierigkeit auswählen können und dann das Spiel starten.
2. Die Schachfiguren sollen über das Brett bewegt werden können.
3. Ich möchte sehen können, welche Züge ich mit welcher Figur machen kann.
4. Ein ungültiger Zug soll eindeutig erkennbar sein, wenn ich ihn versuche auszuführen.
5. Nach dem Spiel sollen die Züge des Gegners sichtbar sein, damit seine Strategie nachvollziehbar wird.
6. Eine Hilfefunktion wäre gut, die mir Grundlagen erklärt oder an bestimmten Stellen unter die Arme greift, wenn ich Hilfe brauche.
7. Ich möchte Spiele speichern und laden können.

Diese User-Stories, also das umgangssprachliche definieren der Anforderungen sollte zunächst einmal festgehalten werden, handschriftlich oder - wenn gewünscht in einem Tool wie **Trello** oder **Notion**.
Aus den User-Stories müssen jetzt konkrete Aufgaben für Entwicklys ausgearbeitet und priosiert werden. Nehmen wir an, es sind drei Teammitglieder beteiligt:

- Monika: zuständig für das Model
- Viktor: zuständig für den View
- Claudia: zuständig für den Controller

Analysieren wir zunächst die erste User Story:

> Man soll in einem Menü eine Schwierigkeit auswählen können und dann das Spiel starten.

Ohne weitere Priorisierung könnte man daraus z.B.: folgende Aufgaben ableiten:

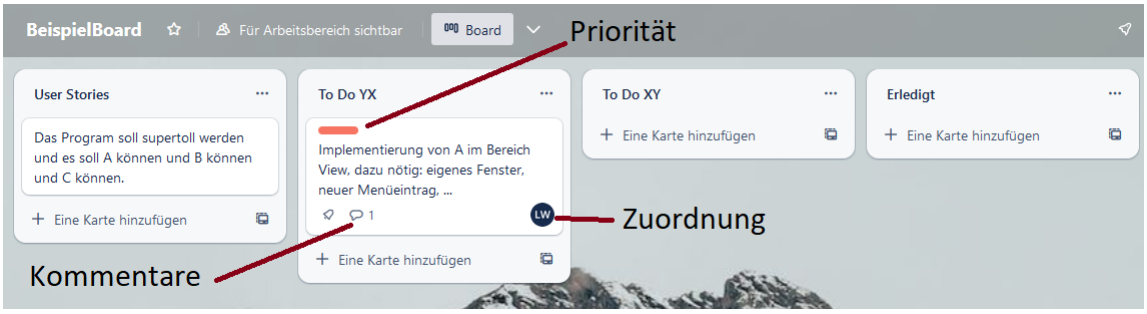
- Viktor:
  - erzeuge ein Spielfenster
  - erzeuge ein Menü mit Spielstart und Schwierigkeitswahl
  - erzeuge das Spielfeld
  - erzeuge die Figuren
- Monika:
  - speichere Daten der Figuren
  - speichere Daten des Spielfeldes bei Start
  - speichere Daten über den Schwierigkeitsgrad
- Claudia:
  - reagiere auf Schwierigkeitswahl
  - reagiere auf Spielstart

Man kann unschwer erkennen, dass die meisten dieser Aufgaben weitere Fragen nach sich ziehen, die zunächst im Team besprochen werden müssen, z.B.:

- Wie sollen die Daten gespeichert werden?
- Welche Schwierigkeitsstufen soll es geben?
- Wie sollen sich diese unterscheiden?

- In welcher Auflösung soll das Spielfeld und das Menü dargestellt werden?
- ...

Es lassen sich dutzende weitere solcher Fragen stellen, deswegen ist eine **Priorisierung** unerlässlich. In obigem Beispiel: Die Auswahl und Verwendung unterschiedlicher Schwierigkeitsgrade ist zwar erstrebenswert, gehört aber zunächst nicht zur Grundfunktionalität eines Schachspiels, d.h. die Priorität sollte hier sicherlich für Viktor darauf liegen, Figuren und das Schachbrett vernünftig erzeugen und darstellen zu können. Monika muss sich darüber klar werden, wie die Figuren in den Daten repräsentiert werden. Claudia kann ohne die beiden Implementierungen noch nicht produktiv arbeiten, da der Controller auf bestehende Strukturen zugreifen muss, sie kann also entweder die anderen beiden unterstützen oder bereits ein Grundgerüst bauen, in das nur noch konkrete Methodennamen der beiden anderen Bereiche eingebaut werden müssen.
Eine übersichtlichere Möglichkeit, die Gedanken zu ordnen und kollaborativ daran zu arbeiten sind Concept Boards wie Trello:



Hier können die User Stories zuerst gesammelt werden und dann konkrete Aufgaben für Person YX oder XY daraus abgeleitet werden. Jeder Eintrag kann (hilfreich :) kommentiert werden oder es können weitere Zuordnungen der Aufgaben erfolgen, wenn sich ergibt, dass ein Bereich zur Lösung nicht ausreicht. Es können auch Priorisierungen vorgenommen werden (hier z.B. über ein Farbsystem geregelt).
Alternativ können auch Aufgabenlisten nach anderen Kriterien erstellt werden und es wird rein über die Zuordnung gearbeitet, je nachdem, wie die Arbeit für euch am effizientesten ablauft.
Wichtig: in irgendeiner Form **müssen** die Aufgaben verteilt und dokumentiert werden!

4.5 Hinweise Model

Wenn es an die Daten des Modells geht gibt es im Wesentlichen die drei bereits erwähnten Methoden, die auch im Schwierigkeitsgrad ansteigen. Grundsätzlich gibt es natürlich auch noch weitere Arten der Datenverwaltung, um den Rahmen jedoch nicht zu sprengen beschränken wir uns auf die **Datenklassen**, **JSON** und **SQL-Datenbanken**.

4.5.1 Datenklassen

Jedwede Modellierung einer „natürlichen“ Entität führte bereits bisher zu einer Art Datenklasse - z.B. die Klasse „Human“, die die Menschen in unserer Warteschlange repräsentiert haben.
Datenklassen zeichnen sich dadurch aus, dass sie keine weitere Logik enthalten außer die für diese Daten relevante, d.h. wir hatten beispspielsweise den Menschen:

```
public class Human {
    String name;
    int age;

    public Human(String name, int age) {
        this.name=name;
        this.age =age;
    }

    //getter, setter, equals...

}
```

Die einzige Aufgabe dieser Klasse ist es, die Daten, die einen Menschen charakterisieren zu verwalten.
Mit Java 14 gibt es ein neues Konstrukt, dass die Erstellung von Datenklassen etwas einfacher macht, die sogenannten **records**. Aus der offiziellen Hilfe bzw. Dokumentation entnommen:

```
record Rectangle(float length, float width){ }
```

ist gleichbedeutend mit:

```
final class Rectangle implements Shape {
    final double length;
    final double width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double length() { return length; }
    double width() { return width; }
}
```

Die Verwendung dieses Konstrukts ist natürlich nicht immer sinnvoll und auch nicht notwendig, die Erwähnung hier soll nur als Hinweis verstanden werden.
Möchte man nicht nur Daten verwalten, wie ein Objekt, also z.B. der Mensch aussieht, sondern auch konkrete Menschen in Klassen verwalten, könnte man das auch mit weiteren Klassen regeln (dies ist aber eher unüblich).
Beispiel: es soll ein ganzer Chor modelliert werden, neben Name und Alter wird in der Mensch-Klasse noch die Tonlage hinterlegt, das könnte z.B. so aussehen:

```
public class Choir {
    /*final damit der Choir nicht anderswo verändert werden kann, nur hier
    static, damit kein Objekt der Klasse Choir erzeugt werden muss, um auf
    den Chor zuzugreifen */
    final static Human[] choir = {
        new Human("Horst", 55, "Bass");
        new Human("Angela", 60, "Alt");
        //...
    };
}
```

4.5.2 JSON

JSON (JavaScript Object Notation) ist ein textbasiertes Datenformat. Die Daten werden hier in einem bestimmten Format in einer eigenen Datein (.json) abgespeichert. Die Syntax ist dabei relativ einfach zu lesen und zu verstehen, ein JSON-Datenobjekt wird immer durch geschweifte Klammern gekennzeichnet, ein Beispiel:

```
{
    "name": "Max Mustermann",
    "age": 30,
    "email": "max.mustermann@example.com",
    "address": {
        "street": "Musterstraße 123",
        "city": "Musterstadt",
        "country": "Deutschland"
    },
    "phoneNumbers": [
        "+49 123 456789",
        "+49 987 654321"
    ]
}
```

In JSON werden Daten in Form von Schlüssel-Wert-Paaren dargestellt, der Schlüssel steht dabei in Anführungszeichen, gefolgt von einem Doppelpunkt und den entsprechenden Werten (wenn es sich um Text handelt auch in Anführungszeichen!), einzelne Datenpunkte werden durch Kommata voneinander abgetrennt.
In obigem Beispiel sieht man auch noch zwei weitere mögliche Notationen:

1. **Die Adresse**: hinter dem Doppelpunkt der Adresse findet sich eine weitere geschweifte Klammer, das heißt die hinterlegten Daten können wieder ein **JSON-Objekt** sein. (man kann sie also beliebig tief „verschachteln“).

2. **Die Telefonnummern**: hier befindet sich hinter dem Doppelpunkt eine eckige Klammer, das bedeutet, dass sich hier eine Liste an Werten findet. Der Unterschied wird insbesondere im Zugriff auf die Daten sichtbar, es gibt aber insbesondere keinen weiteren Schlüssel, der die Werte in der Liste charakterisiert, in der Adresse oben dagegen wird die Straße noch einmal explizit als **Schlüssel** hinterlegt.

Möchte man in Java mit JSON-Objekten umgeen, gibt es zum Beispiel die externe Bibliothek **org.json**, ein Anwendungsbeispiel:

```java
import org.json.*;

public class Beispiel {
   public static void main(String[] args) {
      //JSON wird in einem Text definiert
      String jsonText = "{ \"name\": \"Max Mustermann\", \"age\": 30,
      \"email\": \"max.mustermann@example.com\",
      \"address\": { \"street\": \"Musterstraße 123\",
      \"city\": \"Musterstadt\", \"country\": \"Deutschland\" },
      \"phoneNumbers\": [ \"+49 123 456789\", \"+49 987 654321\" ] }";

      //Es wird ein JSON-Objekt aus dem definierten Text gebaut.
      JSONObject jsonObj = new JSONObject(jsonText);

      //Die Daten werden mit den passenden gettern ausgelesen...
      String name = jsonObj.getString("name");
      int age = jsonObj.getInt("age");
      String email = jsonObj.getString("email");
      String street = jsonObj.getJSONObject("address").getString("street");
      String city = jsonObj.getJSONObject("address").getString("city");
      String country = jsonObj.getJSONObject("address").getString("country");
      JSONArray phoneNumbers = jsonObj.getJSONArray("phoneNumbers");

      //... und ausgegeben
      System.out.println("Name: " + name);
      System.out.println("Alter: " + age);
      System.out.println("E-Mail: " + email);
      System.out.println("Straße: " + street);
      System.out.println("Stadt: " + city);
      System.out.println("Land: " + country);
      for (int i = 0; i < phoneNumbers.length(); i++) {
         String phoneNumber = phoneNumbers.getString(i);
         System.out.println("Telefonnummer " + (i+1) + ": " + phoneNumber);
      }
   }
}
```

In obigem Beispiel ist der JSON-Text noch im Code direkt angelegt, es ist aber natürlich noch möglich diesen in eine eigene Datei, z.B. data.json auszulagern:

```
    try {
         // Dateiinhalt in einen String lesen
         String jsonText = new String(Files.readAllBytes(Paths.get("data.json")));

         // ... wie oben
      } catch (Exception e) {
         System.err.println("Fehler beim Lesen der JSON-Datei: " + e.getMessage());
      }
```

Achtung: der Pfad ist relativ zum root-Verzeichnis des Projekts zu verstehen, liegen die Daten also in einem Unterordner, so muss dieser in den PFad eingebettet werden, z.B.: „ordner/data.json“.
Allgemein lässt sich sagen, dass die Verwendung von JSON mit Java häufig etwas umständlich wirkt, aber dennoch vernünftig funktioniert.

4.5.3 SQL-Datenbanken - nur für Experten

Es ist unwahrscheinlich, dass bei einem der Projekte die Verwendung einer Datenbank notwendig wird, aber der Vollständigkeit halber sei diese Möglichkeit erwähnt. Man kann aus Java direkt auf eine existierende SQL-Datenbank zugreifen, der Prozess ist aber häufig trickreich und fehleranfällig. Eine vernünftige Übersicht findet sich z.B. hier. Ich empfehle die Verwendung von JDBC, also der ersten Option, wenn gewünscht.
Anbei ein Anschauungsbeispiel, wie mit JDBC ein Login-System realisiert wurde:

```
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class DatabaseConnect {

    private static final String DB_PASSWORD = "password";
    private static final String DB_USERNAME = "client";


    public static void setPassword(String username, String password) {
        String sqlSetPassword = "INSERT INTO login VALUES (?, ?)";
        String hashedPassword = buildPasswordString(password);
        Connection con = connect();
        try {
            PreparedStatement ps = con.prepareStatement(sqlSetPassword);
            ps.setString(1, username);
            ps.setString(2, hashedPassword);
            ps.executeUpdate();
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void updatePassword(String username, String password) {
        String sqlUpdatePassword = "UPDATE login SET `Password` = ?  WHERE `Username` = ?";
        String hashedPassword = buildPasswordString(password);
        Connection con = connect();
        try{
            PreparedStatement ps = con.prepareStatement(sqlUpdatePassword);
            ps.setString(1, hashedPassword);
            ps.setString(2, username);
            ps.executeUpdate();
            con.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }

    public static boolean login(String username, String password) {
        String sqlFetchPassword = "SELECT `password` FROM login WHERE `Username` = ?";
        String hashedLoginPassword = buildPasswordString(password);
        Connection con = connect();
        try {
            PreparedStatement ps = con.prepareStatement(sqlFetchPassword);
            ps.setString(1, username);
            ResultSet results = ps.executeQuery();
            results.next();
            String databasePassword = results.getString(1);
            con.close();
            if(databasePassword.equals(hashedLoginPassword)) return true;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    private static String buildPasswordString(String password) {
        BigInteger bigInteger = null;
        try {
            bigInteger = new BigInteger(1, getSHA(password));
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        StringBuilder hexString = new StringBuilder(bigInteger.toString(16));
        while(hexString.length() < 64) {
            hexString.insert(0, '0');
        }
        return hexString.toString();
    }

    private static byte[] getSHA(String input) throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        return md.digest(input.getBytes(StandardCharsets.UTF_8));
    }

    private static Connection connect(){
        String CONNECTION_URL = "jdbc:mysql://localhost:3306/database";
        Connection con = null;
```

```
        try{
            con = DriverManager.getConnection(CONNECTION_URL, DB_USERNAME, DB_PASSWORD);
        } catch(SQLException e) {
            e.printStackTrace();
        }
        return con;
    }

}
```

4.6 Hinweise View

Ähnlich wie beim Model gibt es verschiedene Wege, die man einschlagen kann, um dem User eine Interaktion mit dem Program zu ermöglichen. Wie oben steigt dabei der Schwierigkeitsgrad der vorgestellten methoden tendenziell an.

4.6.1 Konsole

Beginnen wir mit der einfachsten Methode, der Asugabe und Interaktion über die Konsole. Diese Möglichkeit haben wir durch die Verwendung von System.out.println() schon sehr häufig verwendet. Grundsätzlich spricht natürlich nichts dagegen den Anwender so mit dem Program kommunizieren lassen, jedoch haben sich - gerade im Windows-Bereich - die graphischen Oberflächen für die Mehrheit der Bevölkerung durchgesetzt. Wollt ihr trotzdem diesen Weg gehen, dann sei an die Scanner-Klasse erinnert, damit auf Eingaben des Benutzers reagiert werden kann, z.B.:

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        // Erstelle einen neuen Scanner, der auf System.in (Standard-Eingabe) liest
        Scanner scanner = new Scanner(System.in);

        // Lese die nächste Zeile der Eingabe
        System.out.println("Bitte geben Sie Ihren Namen ein:");
        String name = scanner.nextLine();

        // Lese eine eingegebene
        System.out.println("Bitte geben Sie Ihr Alter ein:");
        int age = scanner.nextInt();

        // Gib den Namen und das Alter aus
        System.out.println("Sie sind " + name + " und sind" + age + " Jahre alt.");

        // Schließe den Scanner
        scanner.close();
    }
}
```

4.6.2 Swing

Eine der klassischen Methoden Graphical User Interfaces (GUIs) in Java zu schreiben ist die Swing-Bibliothek. Sie kommt bereits mit der Standard-Bibliothek und kann direkt verwendet werden, die grundlegende Syntax ist nicht schwer, der folgende Code erzeugt ein Fenster mit einem Label:

```java
import javax.swing.*;

public class SwingDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Meine SWING-Applikation");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hello World!");
        frame.add(label);
        frame.setVisible(true);
    }
}
```

Im Wesentlichen ist das Hauptfenster ein Frame, der als Breite den absoluten Wert 400px und als Höhe den Wert 300px bekommt. Wenn wir auf das bekannte X klicken wird beendet und zu unserem Frame wird ein Label hinzugefügt. Am Ende der Methode wird der Frame sichtbar „geschalten“.
Leider bleibt es nicht ganz so einfach, insbesondere die Interaktion mit dem Benutzer ist in den Implementierungen in Java zuerst etwas unübersichtlich, fügen wir einen Button nach unserem Label hinzu, der das Label ein- bzw. ausblenden soll.

```java
JButton button = new JButton("Hide/Show Label");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (label.isVisible()) {
            label.setVisible(false);
        } else {
            label.setVisible(true);
        }
    }
});
frame.add(button, BorderLayout.SOUTH);
```

Betrachten wir zuerst die letzte Zeile, der Button wird wieder dem Frame hinzugefügt, jetzt aber im „Süden“, also am unteren Ende unseres Rahmens. Allgemein werden die Positionen in sogenannten Layouts verwaltet, die als Container für die einzelnen graphischen Objekte dienen. Je nachdem wie man die Anordnung haben möchte gibt es verschiedene Klassen, z.B. das obige BorderLayout, dass die Komponenten in fünf Bereichen anordnet: Norden, Süden, Osten, Westen und Mitte.
Jetzt zum komplizierteren Teil, zunächst erzeugen wir einen neuen Button mit seinem Text. Würde man den Rest des Codes weglassen wäre dieser Button aber funktionslos, der Konstruktor erzeugt nur das graphische Element, das auch angezeigt wird, stattet es aber noch **nicht** mit Funktion aus.
Das passiert erst in der nächsten Zeile, wir wollen einen sogenannten **ActionListener** hinzufügen, d.h. wenn etwas mit dem Button passiert, möchten wir eine Aktion auslösen.

Hinweis: Wollen wir beispielsweise nur auf Maus-Eingaben reagieren, so könnte alternativ ein MouseListener hinzugefügt werden.

Wir möchten natürlich nicht, dass einfach irgendetwas passiert, sondern unseren eigenen ActionListener definieren. Hier wird es etwas trickreicher: ActionListener ist eigentlich ein Interface, das fordert, dass eine Methode *actionPerformed()* implementiert wird. Wir möchten aber nicht für jeden unserer Listener eine eigene Klasse irgendwo in unserem Code haben, sondern möglichst an genau dieser Stelle den Listener implementieren. Dazu wird eine sogenannte **anonyme innere Klasse** verwendet. Wir sehen oben, dass nach Aufruf des Konstruktors eine geschweifte Klammer aufgeht, wir definieren also eine weitere - nicht benannte (deswegen auch „anonym“) - Klasse direkt in unserem Code (deswegen „innere“), diese anonyme innere Klasse implementiert das ActionListener-Interface und wir überschreiben die **actionPerformed()** - Methode so wie wir sie wollen:

ist das Label sichtbar, wird es versteckt, andernfalls wieder sichtbar gemacht.

Hinweis: Wir kümmern uns nicht darum, was genau das zugehörige ActionEvent war, deswegen greifen wir innerhalb der *actionPerformed()*-Methode auch nicht auf den Eingabeparameter *e* zu, der das Event, also das Ereignis repräsentiert!

Dieses Beispiel stellt natürlich nur einen sehr, sehr kleinen Einblick in die Welt von Swing dar, sollte aber als Ausgangspunkt genügen, um von hier aus weiter zu arbeiten. Es gibt unzählige Tutorials im Internet, eine Auswahl (nicht streng auf Tauglichkeit geprüft):

1. **Javatpoint**: textbasiertes Tutorial, geht nacheinander auf verschiedene Objekte ein, gut als Nachschlagewerk mit Beispielen geeignet.
2. **geeksforgeeks**: textbasiert, weniger übersichtlich strukturiert, aber viele Artikel.
3. **Offizielle Doumentation**: wenn Details genau nachgelesen werden müssen.
4. **Morpheus**: Youtube-Reihe. Sehr ausführlich, aber schon etwas älter.
5. **Java Code Junkie**: Youtube-Reihe, ebenfalls ausführlich, etwas aktueller.
6. Allgemein: Google/ChatGPT/Youtube: ihr wisst wie das geht :)

4.6.3 JavaFX - für Experten

Etwas moderner, aber noch etwas umfangreicher in der Gestaltung als Swing ist JavaFX. Entscheidet man sich für diesen Ansatz, so sollte direkt das Ganze Projekt als JavaFX Projekt angelegt werden, da sich Abhängigkeiten ergeben, die nicht von der Standardbibliothek abgedeckt werden. Wird mit VSCode gearbeitet finden sich als Einstiegspunkt hier Hinweise, sodass die entsprechenden Abhängigkeiten automatisch eingebaut sind.
Auch für JavaFX gibt es zahlreiche Tutorials (ebenfalls nicht vollständig geprüft):

1. **Offizielles Oracle Tutorial**: referenziert die NetBeans IDE, natürlich auch mit anderen IDEs möglich.
2. **Javatpoint**: textbasiertes Tutorial, geht nacheinander auf verschiedene Objekte ein, gut als Nachschlagewerk mit Beispielen geeignet.
3. **geeksforgeeks**: textbasiert, weniger übersichtlich strukturiert, aber viele Artikel.
4. **Offizielle Doumentation**: wenn Details genau nachgelesen werden müssen.
5. **ITCademy**: Youtube-Reihe, relativ ausführlich und aktuell.
6. Allgemein: Google/ChatGPT/Youtube: ihr wisst wie das geht :)

4.7 Hinweise Controller

Im Gegensatz zu Model und View lassen sich für den Controller keine generellen Empfehlungen für den Anfang geben, da es stark auf die Implementierungen der anderen beiden Bereiche ankommt, wie Methoden geschrieben werden.
Es empfiehlt sich, dass in der Anfangsphase des Projekts das Teammitglied, das als Hauptbereich den Controller hat die anderen Teammitglieder unterstützt, je nachdem wo die größeren Probleme vorliegen.

4.8 Ideensammlung für Projekte

Hinweis: Idealerweise sollte das Projekt natürlich im zeitlichen Rahmen beendet werden, sollte das aufgrund des Umfangs nicht möglich sein muss das nicht zwangsläufig zu Punktabzug führen. Es muss aber eine Grundfunktionalität hergestellt werden, damit erkennbar ist, dass das Projekt grundsätzlich erfolgreich verläuft und nur noch nicht „vollständig“ ist.

1. **Spiele** in verschiedenster Form: text based adventure, Quizspiel, Kartenspiel, etc. Der Phantasie sind keine Grenzen gesetzt,...
2. **Verwaltungssoftware**: für Noten, Rezepte, Inventar, Bibliothek,...
3. **Physikalische Simulationen**: Simulation verschiedern physikalischer Gesetze, grafische Veranschaulichung,...
4. **Ploten**: Das ganz eigene Geogebra, oder ein kleiner Mathehelfer,...
5. **Lernprogram**: Ein interaktives Program zum Vokabeln lernen, oder vielleicht für Chemie-Formeln,...