Nachdenkzettel – Gruppe Pick'n Drink

Nachdenkzettel Git

Was sind Gründe ein Version Control System (VCS) wie Git zu verwenden?

Es gibt einige wichtige Gründe, ein Version Control System (VCS) wie Git zu verwenden, so wie der Name schon verrät, ist einer der wichtigsten Gründe die Versionsverwaltung, das bedeutet, es können mehrere Versionen eines Projekts verwaltet werden. Git ermöglicht es, den Verlauf von Änderungen an Dateien oder Ordnern genau zu verfolgen, einschließlich aller früheren Versionen. Dadurch können frühere Zustände wiederhergestellt werden. Durch die Versionsverwaltung fällt es mehreren Entwicklern leichter, an einem Projekt zu arbeiten, was ebenfalls ein Grund für die Nutzung eines VCS darstellt. Weitere Gründe für die Nutzung eines VCS sind die gleichzeitige Zusammenarbeit an verschiedenen Teilen des Codes, ohne sich gegenseitig zu stören, oder das Erstellen von mehreren Verzweigungen, sog. Branches. Durch das regelmäßige Committen und Pushen von Änderungen in ein Remote-Repository, wird das Projekt vor Datenverlusten geschützt. Zudem ist Git Plattformunabhängig und bietet durch die OpenSource Anwendung einen guten Austausch mit der Community.

Welche Dateitypen gehören in ein Repository, welche nicht? Bitte begründen Sie Ihre Entscheidung.

Dateien die in eine Repository gehören:
java → muss definitiv in ein Repository aufgenommen werden, da es sich hier um den Quellcode handelt um den sog. Kern des Projektes.
Folgende Dateien können in ein Repository aufgenommen werden, wenn sie für das Projekt eine Bedeutung haben, müssen aber nicht unbedingt: **xml, json, Bilddateien** (für die Grafikoberfläche oder Grafiken des Projektes), **Musikdateien** (z.B. bei einer Musik-App), **UML Modelle** (wenn sie die Entwicklung durch eine Dokumentation und Planung unterstützen), **Notizen und Dokumentationen** (sind wichtig, um das Projekt zu verstehen und zu warten), **Log-Dateien** (um Probleme zu diagnostizieren und können je nach Bedarf in einer Repository gespeichert werden) und die **Konfigurationsdateien** (die für die Bereitstellung und Ausführung des Projekts erforderlich sind).

Dateien die nicht in ein Repository gehören:
Vertrauliche Daten (z.B. Passwörter) → Vertrauliche Informationen sollten niemals in einem Repository gespeichert werden. Vertrauliche Daten müssen immer sicher und getrennt von dem Quellcode und den Konfigurationsdateien aufbewahrt werden.
Abschlussarbeiten → ein Repository beinhaltet immer nur den Projektcode und die zugehörigen Ressourcen, alles andere ist für das Projekt nicht relevant und ist daher nicht in einem Repository zu speichern.

Beschreiben Sie kurz den Ablauf zur Arbeit mit einem Git-Repository. Gehen Sie auf folgende Aspekte ein: Hinzufügen zur Staging-Area, Commit, push, pull.

Die Staging-Area ist ein sogenannter Cache bzw. Zwischenspeicher, der Änderungen von Dateien organisiert, bevor diese über einen Commit in das Repository übernommen werden. Die Staging-Area ermöglicht es uns, gezielt auszuwählen, welche Änderungen in das Commit aufgenommen werden sollen und welche nicht. Wenn ich als Entwickler nun meine neue Änderung auf mein lokales Git-Repository laden möchte, beginne ich zuerst damit meine Änderungen der Staging-Area hinzuzufügen, diese können über eine Checkbox angeklickt werden. Nachdem alle notwenigen Änderungen ausgewählt wurden, wird eine Commit erstellt, um diese Änderungen in unser lokales Git-Repository aufzunehmen. Hierzu kann ich dann noch eine Commit-Message schreiben, die alle wichtigen Änderungen und Informationen beschreibt und dokumentiert. Nun sind die Änderungen in meinem lokalen Git-Repository gespeichert und können über den Befehl Push in unser Remote-Repository hochgeladen werden. Alle anderen Teammitglieder bzw. Entwickler des Projektes können nun die vorliegenden Änderungen pullen und diese werden dann in deren jeweiliges lokales Repository übernommen.

Wie beheben Sie einen Merge Conflict?

Bei einem Merge Conflict handelt es sich um einen Konflikt zwischen zwei verschiedenen Versionen, bei denen derselbe Code oder dieselbe Datei in verschiedenen Branches geändert wurde. Die Entwicklungsumgebung Intelij IDEA weißt beim commit and push den Entwickler darauf hin, dass ein Merge Conflict entstanden ist. Unser Team hat dann die Möglichkeit zu entscheiden, ob der Merge Conflict aufgelöst werden soll oder nicht. Intelij IDEA bietet eine Gegenüberstellung der betroffenen Codeabschnitte aus den verschiedenen Branches, nun kann Schritt-für-Schritt entschieden werden, welche Änderungen aus welcher Branche übernommen werden sollen. Am Ende werden die Änderungen noch übernommen und können in die Repo gespeichert werden.

Wie ist Ihr Git-Vorgehen im Team? Wann werden Änderungen gepusht? Wie oft veröffentlichen Sie Ihre Änderungen? Wie beschreiben Sie Ihre Änderungen in den commit-Messages?

In unserem Team versuchen wir immer dann zu pushen, wenn es zu neuen und sinnvollen Änderungen im Projekt kommt. Zudem pushen wir immer dann, wenn wir unsere Änderungen mit dem Team teilen wollen und damit weiterarbeiten wollen. Daher veröffentlichen wir unsere Änderungen regelmäßig. Wir achten jedoch immer darauf, nur Commits zu

pushen, die auch stabil und funktionsfähig sind, damit sie auch sinnvoll wieder von anderen Projektmitgliedern genutzt werden können. Unsere Commit-Messages enthalten immer eine kleine Stichwortartige Zusammenfassung aller Änderungen: klar, präzise und informativ. Unwichtige Informationen, bzw. nicht für das Projekt relevante Commit-Messages sind nicht erlaubt.

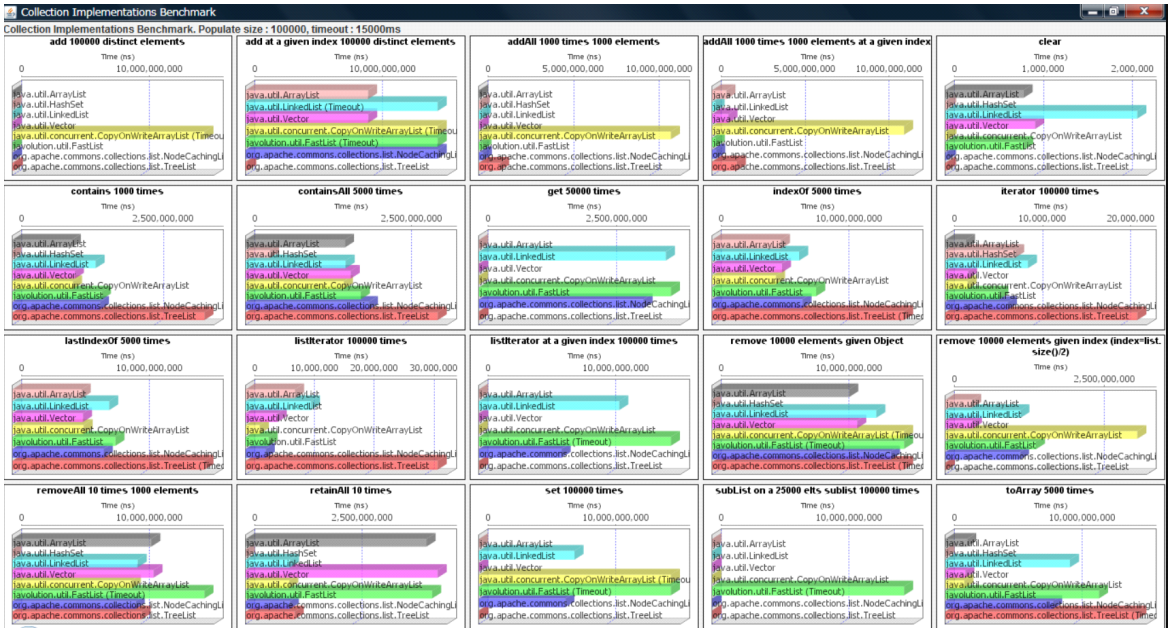
Nachdenkzettel Collections

ArrayList oder LinkedList - Wann nehmen wir was?

ArrayList: Elemente werden in einem internen Array speichert und dynamisch vergrößert, wenn Elemente hinzugefügt werden.
+ Zugriff auf Elemente ist schnell
- Einfügen und Löschen kann langsamer sein, da bei Veränderungen möglicherweise Elemente verschoben werden müssen

Linkedlist: wird auch verkettete Liste genannt. Alle Listenelemente stehen in Verbindung zum jeweiligen Vorgänger bzw. Nachfolger. Existiert kein Nachfolger, so verweist das letzte Element auf die null-Referenz.
+ Elemente können schneller hinzugefügt und gelöscht werden, da nur die Verweise zum Nachbarn geändert werden müssen.
- wenn ein Element an einer bestimmten Position ausgelesen werden soll, ist der Zugriff langsamer, da die Verbindungen bis zu dem passenden Element durchlaufen werden müssen
→ Die Entscheidung für einen bestimmten Listen-Typ sollte man von der Art und Anzahl der Zugriffe abhängig machen

Interpretieren Sie das folgende Benchmarkdiagram (Quelle: DZone.com - Java Collection Performance). Was fällt dabei auf? Wann nutzen Sie welche Collection?



Die Benchmark zeigt verschiedene Collection-Typen, die auf ihre Leistungsfähigkeit überprüft wurden. In diesem Fall werden die Zeiten gemessen, die verschiedene Java-Collection-Implementierungen benötigen, um bestimmte Aufgaben auszuführen.
Die verschiedenen Collection-Typen wurden zum Beispiel auf folgende Operationen getestet: add, contains, get, index of, iterator, remove, set und toArray.
Für jede dieser Operationen werden die Leistungszeiten für verschiedene Mengen von Operationen dargestellt, wie das Hinzufügen von 10.000 oder 100.000 Elementen, oder das Suchen, Abrufen und Löschen von Elementen. Die Zeiten werden in Nanosekunden (ns) gemessen und auf der vertikalen Achse jedes Diagramms dargestellt. Die Leistung kann stark variieren, je nachdem, ob es um das Hinzufügen am Ende der Liste, das Suchen von Elementen, das Abrufen von Elementen an bestimmten Positionen oder das Iterieren über alle Elemente geht.
→ wichtig ist zu beobachten, dass nicht jeder Collection-Typ für jeden Anwendungsfall geeignet ist, jede Collection hat Vor- und Nachteile.

ArrayList: schnellen zufälligen Zugriff auf Elemente, Elemente am Ende der Liste hinzufügen, seltene Änderung der Listenstruktur, kleine oder mittelgroße Listen
HashSet: Einzigartigkeit der Elemente, schnelle Suchoperationen, nicht sortierte Sammlung, Effizienz bei ‚add' und ‚remove'
LinkedList: Häufiges Einfügen und Löschen von Elementen, Liste als Warteschlange oder Stapel, große Listen mit nicht zufälligem Zugriff, Speicheroptimierung bei großen Elementen

Vector: Thread-Sicherheit
CopyOnWriteArrayList: ist eine thread-sichere Variante der ArrayList, die effizient ist, wenn es viele Leseoperationen und nur wenige Änderungen an der Liste gibt
FastList : Echtzeitanforderungen, Hohe Leistung bei Iteration und Modifikation, Anpassbarkeit und Speichereffizienz
NodeCachingLinkedList: Effiziente Einfügung und Löschung
TreeList: Schneller zufälliger Zugriff und Einfügeoperationen, große Listen

Warum ist die CopyOnWriteArrayList scheinbar langsam?

Bei jedem Schreibvorgang wie dem Hinzufügen oder Entfernen von Elementen, kopiert sie die ganze Liste, was bei großen Listen viel Zeit und Speicher in Anspruch nimmt. Daher kann sie langsamer wirken im Vergleich zu der ArrayList.

Schauen Sie sich folgenden Code an. Was passiert? Erklären Sie, warum dies passiert. Wie kann der Effekt umgangen werden?

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    int i = itr.next();
    if (i > 5) { // filter all ints bigger than 5
        list.remove();
    }
}
```

Der Codeausschnitt löscht alle Elemente einer Liste, die größer als 5 sind. Es wird zu Beginn des Codes ein Iterator für die Liste „list“ geschrieben. Solange der Iterator ein nächstes Element besitzt wird die while-Schleife durchlaufen. Für jedes Element wird überprüft, ob es über 5 ist. Wenn die Bedingung eintritt wird die Methode list.remove() ausgeführt. Letzteres Element führt dazu, dass nicht das aktuelle Element des Iterators entfernt wird, daher sollte die Methode itr.remove() angewendet werden.

Nachdenkzettel Interfaces

Class B implements X. Jetzt fügen Sie eine neue Methode in Interface X ein. Was passiert? Was bedeutet dies für die „Robustheit“ der Interface-Definition?

Wenn eine Klasse ein Interface implementiert, verpflichtet sie sich dazu, deren Methoden zu implementieren. Das bedeutet, die Klasse muss bestimmte Vertragsbedingungen einhalten.
Daher müsste die neue Methode in der Klasse B auch implementiert werden. Wenn nicht, dann würde die Klasse B nicht mehr den „Vertrag“ des Interfaces erfüllen, was zu einem Kompilierungsfehler führen würde.

Die Robustheit einer Interface-Definition bezieht sich darauf, wie widerstandsfähig und flexibel das Interface gegenüber Änderungen ist.
Das Einfügen einer neuen Methode würde für die Robustheit auf der einen Seite bedeuten, dass man durch die Erweiterung neue Funktionen hinzufügen kann, ohne bestehenden Code zu ändern. Die Klassen, die das Interface implementieren, können dann die neuen Methoden integrieren.
Durch die Integration der neuen Methoden in eine bestehende Klasse kommt es auf der anderen Seite zu Änderungen im bestehenden Code, was zusätzlichen Arbeitsaufwand mit sich bringt.

Ihr Code enthält folgendes Statement:

X myX = new X();
myX.doSomething();

Was ist daran problematisch, wenn Sie eine Applikation für verschiedene Branchen/Kunden/Fälle bauen? Wie schaffen Sie Abhilfe? Welche Rolle spielen dabei Factories?

Es ist problematisch, da X eine starre Instanziierung direkt im Code darstellt. Dadurch sind der Code und die konkrete Implementierung stark miteinander gekoppelt. Es könnte zu einer mangelnden Flexibilität kommen, weil der Code stark an die Klasse X gebunden ist. Möchte man etwas ändern oder ersetzen, müsste man den Code anpassen. Der Code wird schwer wartbar, man müsste bei Änderungen jeden Ort finden, an dem X instanziiert wird. Außerdem wird es schwierig, wenn verschiedene Kunden unterschiedliche Implementierungen von X benötigen.
Durch Factories würde die Flexibilität, Wartbarkeit und Konfigurierbarkeit verbessert werden. In einer Factory kann man jede, nach Bedarf verschiedene, Implementierung von X erstellen, was vor allem für den Aspekt der verschiedenen Branchen oder Kunden sehr hilfreich wäre.

Was gehört zum „Interface“ einer Java-klasse? Heißt: Was ist für außenstehende Objekte ersichtlich?

Das Interface einer Java-Klasse definiert, welche Aspekte der Klasse für externe Klassen und Objekte sichtbar und zugänglich sind. Man kann es sich vorstellen wie eine Art Vertrag, der beschreibt, wie andere Teile des Codes mit der Klasse interagieren können, ohne die internen Details der Implementierung zu kennen.
Das Interface bezieht sich auf die public Elemente und Verhaltensweisen der Klasse, die von außenstehenden Objekten verwendet werden können.

Public Methoden: Alle public Methoden, die von außenstehenden Objekten aufgerufen werden können. Sie repräsentieren die Schnittstelle (das Interface) der Klasse und definieren wie externe Objekte mit der Klasse interagieren können.

Konstruktoren: public Konstruktoren, die von außenstehenden Objekten verwendet werden können um neue Instanzen der Klasse zu erstellen.

Implementierte Interfaces: Eine Klasse kann mehrere Interfaces implementieren. Das Interface, das von der Klasse implementiert wird, definiert ebenfalls einen Teil des Interfaces der Klasse.

Static Elemente: static Methoden oder Konstruktoren, die direkt auf die Klasse selbst bezogen sind und nicht an eine bestimmte Instanz gebunden sind.

Public Felder: Teil der Interfaces, sollten aber wegen Prinzipien wie Datenkapselung und Encapsulation vermieden werden. Klassen Felder sollten in der Regel privat gehalten werden und den Zugriff über Getter- und Setter-Methoden ermöglichen.

Nachdenkzettel Logging

Kennzeichnen Sie in der nachfolgenden Konfiguration

Was geloggt wird

Es werden Nachrichten in die Konsole ausgegeben, die das angegebene Pattern enthalten: %d %-5p [%t] %C{2} (%F:%L) - %m%n.
Außerdem werden Nachrichten in die Datei A1.log geloggt, mit folgendem Pattern: %t %-5p %c{2} - %m%n.

Wieviel geloggt wird

Wie viel geloggt wird, hängt immer vom Log-Level ab. Das Log-Level wird angegeben. Es wird dann alles geloggt, was diesem Level, oder einem höheren Level, entspricht. Umso höher die Priorität bei dem Log-Level ist, umso weniger Informationen werden geloggt, es wird also immer spezifischer. In dieser Konfiguration ist das Log-Level auf ‘debug’ gesetzt, was bedeutet, dass Nachrichten mit dem level ‘debug’ oder höher (zB ‘info’, ‘warn', ‘error'...) geloggt werden.
Außerdem gibt es in dieser Konfiguration auch einen eigenen Logger für die Klass se2examples.core.businessLogic.VehicleManager, welcher auch auf ‘debug’ gesetzt ist.

Wo geloggt wird

Die Log-Nachrichten werden in die Datei A1.log geschrieben. Das angegebene append=„false" sorgt dafür, dass die Datei A1.log überschrieben wird, wenn sie bereits existiert. Ausgegeben werden die Nachrichten in die Konsole STDOUT.

Wie geloggt wird

Die Konfiguration enthält zwei verschiedenen Appender. Einen Console-Appender und einen File-Appender. Durch den Console-Appender wird die Information in die Konsole ausgegeben, was es ermöglicht, dass die Lognachricht direkt im Entwicklungsprozess überprüft werden kann. Geloggt wird nach dem angegebenen Pattern. In der Konsole enthält das Pattern einen Zeitstempel (%d), das Log-Level (%-5p), den Thread-Name ([%t]), den Klassenname (%C{2}), den Dateiname und eine Zeilennummer ((%F:%L)), sowie die eigentliche Lognachricht (%m).
Der File-Appender ist für das langfristige protokollieren nützlich. Hier werden die Lognachrichten in eine Datei geschrieben, ebenfalls nach dem angegebenen Pattern. Das Pattern für die Datei enthält den Thread-Name (%t), das Log-Level (%-5p), die Kurzform des Klassennamens (%c{2}) und die Lognachricht (%m).

```
<Configuration>
    <Appenders>
        <File name="A1" fileName="A1.log" append="false">
            <PatternLayout pattern="%t %-5p %c{2} - %m%n"/>
        </File>
        <Console name="STDOUT" target="SYSTEM_OUT">
            <PatternLayout pattern="%d %-5p [%t] %C{2} (%F:%L) -
%m%n"/>
        </Console>
    </Appenders>

    <Loggers>
        <Logger name="se2examples.core.businessLogic.VehicleManager"
level="debug">
            <AppenderRef ref="A1"/>
        </Logger>
        <Root level="debug">
            <AppenderRef ref="STDOUT"/>
        </Root>
    </Loggers>
</Configuration>
```

Könnte man auch alle Klassen eines Packages in eine eigene Datei loggen lassen? Wie?

Ja, das ist möglich. In der hier gegebenen Konfiguration wird schon eine einzelne Klasse in eine Datei geloggt. Ein ganzes Package in eine eigene Datei zu loggen funktioniert letztendlich genau gleich.
Man würde hierfür einen weiteren File-Appender mit einer neuen Datei erstellen, in die das Package geloggt wird. Dann braucht man einen Logger ähnlich zu dem hier gegebenen, der eine einzelne Klasse loggt. Der Unterschied ist, dass man keine Klasse, sondern nur das Package angibt:

```
<Logger name="se2examples.core.businessLogic" level="debug">
    <AppenderRef ref="EigeneDatei"/>
</Logger>
```

Dann würde im hier gegeben Beispiel das Package buisnessLogic in die „Eigene Datei“ geloggt werden, alle anderen Klassen, die nicht in diesem Package sind, würden weiterhin in die Konsole ausgegeben werden.

Geben Sie ein Beispiel, wann Sie die folgende Log Level verwenden würden

error -> für kritische Fehler, die die Funktionalität der Anwendung beeinträchtigen. Beispiel: Exceptions, die den Programmlauf unterbrechen. Oft im catch-Block verwendet.
info -> für allgemeine Infos über den Programmablauf. Beispiel: erfolgreiche Anmeldung eines Users, Beginn/Ende eines Prozesses.
debug -> zur Protokollierung von Entwicklungs- oder Debugging Informationen. Beispiel: Werte von Variablen, die zur Fehlersuche nützlich sind.

Macht ein Logging-Framework Ihre Anwendung langsamer? Wie sollte sich ein Logging-Framework idealerweise verhalten?

Ein Logging-Framework kann die Anwendung langsamer machen, da es zusätzlicher Code ist, der ausgeführt werden muss. Außerdem kann das Schreiben der Log-Nachrichten in die Konsole oder in die Datei Zeit brauchen, wenn es synchron passiert. Man sollte also Log-Level immer überprüfen, um so unnötige Berechnungen und Formatierungen zu vermeiden.

Idealerweise ist ein Logging-Framework flexibel konfiguriert, also dass man als Entwickler entscheiden kann, welche Log-Level benötigt werden und aktiviert sein sollen. So kann man, je nach Umgebung, nur die aktuell wichtigen Informationen loggen. Außerdem ist es gut, wenn das Framework effizient implementiert ist, zum Beispiel durch asynchrone Protokollierung oder Puffern, womit die Leistung verbessert werden kann. Zusätzlich sollte der Entwickler immer darauf achten, dass nur sinnvoll geloggt und nicht unnötig protokolliert wird. Die Nachrichten sollten klar, präzise und verständlich sein.

Nachdenkzettel Vererbung

Class B extends X. Jetzt fügen Sie eine neue Methode in X ein. Müssen Sie B anpassen?

Nicht unbedingt. Solange die neue Methode in X keine abstrakte Methode ist, muss B nicht angepasst werden. Unterklassen erben nicht-abstrakte Methoden automatisch, sie müssen nicht implementiert werden.
Abstrakte Methoden hingegen müssen von den abgeleiteten Klassen immer implementiert werden.

Gegeben ist folgendes Code-Snipet

```
class B extends X {
   public void newMethodinB() { ... }
}
```

Jetzt fügen Sie eine neue public Methode in B, die abgeleitete Klasse, ein. Sie möchten diese neue Methode im Code verwenden. Prüfen Sie die folgenden Codezeilen:

```
X x = new B();
x.newMethodinB();
```

Was stellen Sie fest? Welche Lösung präferieren Sie? Warum?

Hier wird versucht, die Methode newMethodInB auf einem Objekt vom Typ ‘X’ aufzurufen. Dieses Objekt ist aber nur ein Objekt der abgeleiteten Klasse B, nicht der übergeordneten Klasse X. Bei diesem Code würde der Compiler ein Problem melden, weil die Methode in der Klasse X nicht definiert ist. Um die newMethodInB aufrufen zu können, bräuchte man eine Referenz vom Typ B. Man könnte also folgendes machen:

```
B b = new B();
b.newMethodinB();
```

Eine weitere Lösung wäre das Casten. Man könnte X zu B casten, was den Methodenaufruf möglich machen würde. Allerdings kann das zu Laufzeitfehlern führen, weshalb die andere Lösung die Bessere wäre.

Nehmen Sie an, String wäre in Java nicht final. Die Klasse FileName „extends“ die Klasse String. Ist das korrekt? Was kann passieren? Wie heißt das Prinzip dahinter? Tipp: Denken Sie daran, was mit Dateinamensregeln des Betriebssystems passiert. Beispielszenario:

```
String s = new FileName();
//...
s.setValue(":datei?name.txt"); // Methodenname hypothetisch
```

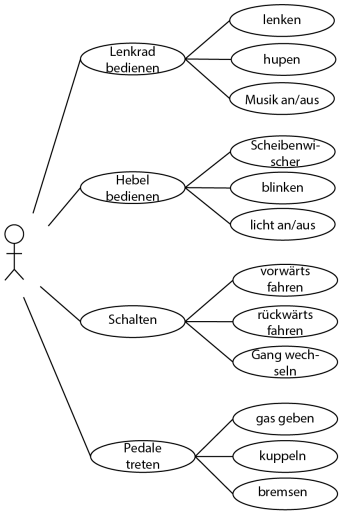
Es wäre theoretisch möglich, von der Klasse String zu erben, wenn die Klasse nicht als final deklariert wäre. Theoretisch könnte man auch eine Instanz von FileName mit einem String Objekt initialisieren. Es könnten dabei aber Probleme auftreten.
Strings sind in Java unveränderlich. Wenn also FileName nicht korrekt implementiert wird, könnten Methoden die Daten verändern, was gegen die Immutabilität von String verstoßen würde. Außerdem könnte es zu Problemen mit den Dateinamensregeln des Betriebssystems kommen. Dateinamen können bestimmte Beschränkungen haben, wenn FileName diese nicht berücksichtigt, könnten Fehler auftreten.

Das Prinzip der finalen Klasseninvariante
Die Klasseninvariante sorgt dafür, dass die interne Darstellung der Klasse geschützt ist. Selbst wenn die Klasse nicht als final deklariert ist, wird die interne Darstellung durch die Klasseninvariante geschützt und kann somit nicht durch Unterklassen geändert werden. So wird die Klasse von unerwünschten Veränderungen oder Sicherheitsproblemen geschützt.

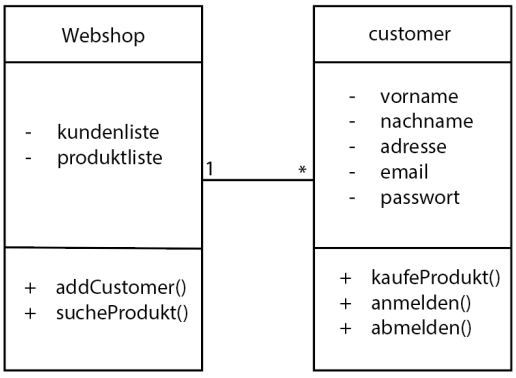
Im Beispielszenario könnte setValue die Integrität der internen Darstellung von String-Objekten gefährden. Dadurch kann es zu Problemen kommen, vor allem wenn die Betriebssystemspezifischen Dateinnamensregeln nicht beachtet werden. Dies sollte in einem sicheren Design von String nicht möglich sein, da die Klasse den Prinzipien der finalen Klasseninvariante folgt. Man sollte also eine solche Vererbung eher vermeiden und stattdessen lieber die Funktionalität erweitern.

Nachdenkzettel: UML

Erstellen sie ein Use Case Diagramm für ein Auto



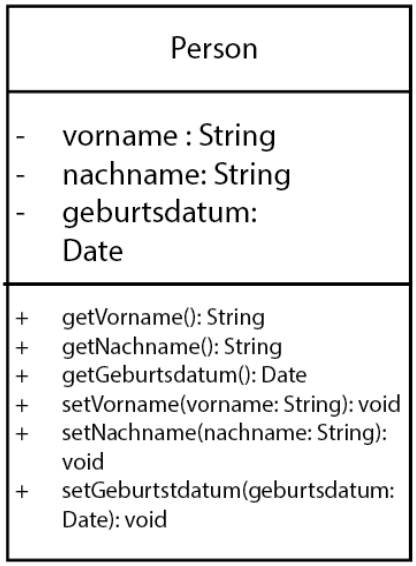
Ein Webshop hat viele Kunden. Zeichnen Sie das Klassendiagramm für die Beziehung. Wie kann man diese Beziehung in der Implementierung in Java darstellen?



```java
public class Webshop {

    2 usages
    private List<Kunde> kundenliste;
    2 usages
    private List<Produkt> produktliste;

    no usages
    public Webshop() {
        kundenliste = new ArrayList<>();
        produktliste = new ArrayList<>();
    }

    no usages
    public void addKunde(Kunde kunde) {        //ermöglicht das Hinzufügen eines Kunden zum Webshop
        kundenliste.add(kunde);
    }

    no usages
    public Produkt sucheProdukt(String produktName) {      //sucht nach einem Produkt mit dem angegeben
        for (Produkt produkt : produktliste) {              //Namen in der Produktliste
            if (produkt.getName().equals(produktName)) {
                return produkt;
            }
        }
        return null; // Produkt nicht gefunden
    }
}
```

```java
public class Kunde {

    no usages
    private String vorname;
    no usages
    private String nachname;
    no usages
    private String adresse;
    no usages
    private String eMail;
    no usages
    private String passwort;

    // Konstruktor und weitere Methoden für den Kunden

    no usages
    public void kaufeProdukt(Produkt produkt) {
        // Implementierung des Kaufprozesses
    }

    no usages
    public void anmelden() {
        // Implementierung der Anmeldung
    }

    no usages
    public void abmelden() {
        // Implementierung der Abmeldung
    }
}
```

Definieren Sie eine Klasse ‘Person’ mit ihren Attributen und Methoden als UML-Klassendiagramm.



Nachdenkzettel Clean Code

Gegeben sind folgende Klassendefinitionen:

```
class Formularfeld;
class Textfeld extends Formularfeld;
class Zahlfeld extends Formularfeld;
class TextUndZahlFeld extends Formularfeld;
class TextfeldOCR extends Textfeld;
class ZahlfeldOCR extends Zahlfeld;
class TextUndZahlFeldOCR extends TextUndZahlFeld;
class TextfeldSonderZ extends TextUndZahlFeld;
class TextfeldOCRSonderZ extends TextUndZahlFeldOCR;
//...
```

Jede weitere Eigenschaft oder Spezialisierung führt zu vielen neuen Klassen durch Kombination. Die Folge ist explosives Anwachsen der Zahl der Klassen mit identischem Code. Wie können Sie dies umgehen?

Das explosive Anwachsen an Klassen kann man umgehen, indem man Interfaces benutzt. Darin befinden sich abstrakte Methoden, die von Klassen implementiert werden müssen. Man sucht demnach ähnliche Funktionen und Eigenschaften und hebt diese in Oberklassen und Interfaces hervor. Hier kann man z.B IText und INummer als interfaces implementieren

und die jeweiligen Methoden, also setText() in IText und setNumber() in INummer in den Klassen Textfeld, Zahlfeld und TextUndZahlfeld überschreiben.

(siehe Präsentation)

Korrekte Initialisierung und Updates von Objekten
Betrchten Sie die Klasse Address:

```
public class Address {
    private String city;
    private String zipCode;
    private String streetName;
    private String houseNumber;

    public void setCity (String c) {
        this.city = c;
    }
    public void setZipcode (String z) {
        this.zipCode = z;
    }

    // weitere Setter für alle Attribute
}
```

Was kann bei der Initialisierung von Adress-Objekten schiefgehen? Wie korrigieren Sie dies? Wie ist Ihre Meinung bzgl. Der Parameterbenennung der aufgeführten set-Methoden?

Im vorliegenden Code sind alle Parameter lediglich deklariert, was bedeutet, dass die Variablen erstellt, aber nicht mit Werten initialisiert wurden. Falls ihnen später keine Werte zugewiesen werden, erhalten sie standardmäßig den Wert "null", was zu einer möglichen Nullpointerexception führen kann. Daher empfiehlt es sich, einen Konstruktor zu erstellen. Ein Konstruktor stellt sicher, dass keine Werte vergessen werden können, da alle notwendigen Informationen beim Erstellen eines Objekts bereitgestellt werden müssen. Dies hilft dabei, potenzielle Fehler im Programmablauf zu vermeiden.

```
public Address(String city, String zipCode, String streetName, String houseNumber ) {
        this.city = city;
        this.zipCode = zipCode;
        this.streetName = streetName;
        this.houseNumber = houseNumber;
    }
```

Zudem sollte den Parametern der set-Methoden immer klare Namen zugewiesen werden, dies hat folgende Gründe: Verständlichkeit des Codes, für die Dokumentation, Fehlervermeidung und leichtere Wartbarkeit.

```
public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
//getter und setter für alle weiteren Attribute
```

Nachdenkzettel Exceptions

Was ist falsch mit dem folgenden Code? Benennen Sie die Fehler und beheben Sie diese.

```
public void doSomething() {
    try {
        lock(); // lock some resource
        // open some resource
        // try to change some things
        // fool around a bit
    } catch (Exception e) {

    } catch (ConcurrentModificationException e) {
        System.out.println("bad stuff going on today!") }
    finally {
        return;
    }
}
```

Fehler:

- Die catch-Blöcke sind in der falschen Reihenfolge, da Exception die Oberklasse aller Exceptions ist, werden alle nachfolgenden catch-Blöcke nicht behandelt
- Der erste catch-Block ist leer, dadurch wird die Exception ignoriert und kann im Falle eines schwerwiegenden Fehlers unentdeckt bleiben, dazu wird Debuggen erschwert
- durch das return im finally-Block wird die Methode vorzeitig verlassen, unabhängig von einer Ausnahme

```
public void doSomething() {
  try {
    lock(); // lock some resource
    // open some resource
    // try to change some things
    // fool around a bit
  } catch (ConcurrentModificationException e) {
    System.out.println("bad stuff going on today!");
  } catch (Exception e) {
    System.out.println("Something happened!");
  } finally {
   /* Wenn man nichts aufräumen muss,
    dann kann der Block leer sein */
  }
}
```

Was ist die Ausgabe des folgenden Programms?

```
public class TestException1 {

    public static void main(String[] args) {
        try {
            badMethod();
            System.out.print("A"); }
        catch (Exception ex) {
            System.out.print("B"); }
        finally {
            System.out.print("C");
        }

        System.out.print("D");

        }

        public static void badMethod() {
            throw new Error();
        }
}
```

Im try-Block wird als erstes die badMethod() ausgeführt, dadurch wird ein Error geworfen. Da Error eine Unterklasse von Throwable ist, wird der catch-Block ignoriert. Finally wird immer ausgeführt, unabhängig ob es eine Exception gab oder nicht, deshalb wird C in der Konsole ausgegeben. Nach dem try...catch…finally soll ein D ausgegeben werden. Dies passiert aber nicht, da das Programm aufgrund des Errors vorzeitig beendet wird.

Wann sollten Sie einen re-throw einer Exception implementieren?

Ein re-throw einer Exception ist dann sinnvoll, wenn man die Exception besser auf höheren Ebenen behandeln kann. Z.B. fängt eine Methode eine Ausnahme ab, aber kann diese nicht vollständig beheben, dann wird diese re-thrown, damit eine höhere Ebene des Codes sie besser behandeln kann. Auch ist es sinnvoll, eine Exception erneut zu werfen, wenn man mehr Informationen braucht, z.B. durch Logging oder auch zum Umwandeln in eine andere Exception durch “throw new Exception()”, um diese besser beheben zu können.

In einem Web-Shop wird zur Laufzeit festgestellt, dass eine Kunden-ID nicht in der Datenbank vorhanden ist. Ist dies eine System-Exception, eine Custom-Exception oder keine Exception? Warum?

Wenn in einem Web-Shop zur Laufzeit festgestellt wird, dass eine Kunden-ID nicht in der Datenbank vorhanden ist, dann ist das eine Custom-Exception, da das Problem keine Ausnahme ist, die von Java automatisch generiert wird. So eine Exception wird normalerweise vom Programmierer selbst erstellt. Z.B Eine ‘CustomerNotFoundException’ um zu zeigen, dass der Nutzer in der Datenbank nicht gefunden wurde.

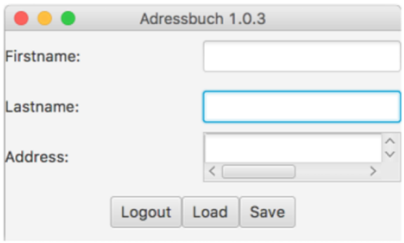
Was ist der Vorteil von Exceptions gegenüber dem Auswerten von Fehlerwerten im Return?

Anstatt ständig return codes abzufragen, sparen wir Zeit, indem wir eine Ausnahme abfangen und direkt behandeln. Durch das Auffangen von Exeptions ersparen wir uns den Programmabbruch und haben eine klare Info über den Fehler durch

Meldungen, was uns bei der Fehlerbehebung hilft. Letztendlich unterstützen diese Ausnahmen den Programmablauf und verhindern Überraschungen und Fehler im Programm.

Nachdenkzettel 9: JavaFX

Welche Layout-Manager würden Sie verwenden, um das folgende Fenster zu realisieren? Tipp: Folgende Layout-Manager wurden in der Vorlesung besprochen: BorderPane, HBox, VBox, StackPane, GridPane, FlowPane, TilePane.




```
<BorderPane >
<bottom>
      <HBox alignment="CENTER" >

            <Button alignment="CENTER" text="Logout" />
            <Button alignment="CENTER" text="Load" />
            <Button alignment="CENTER" text="Save" />

      </HBox>
   </bottom>
   <right>
      <VBox spacing="30" alignment="TOP_RIGHT" >

            <TextField />
            <TextField  />
            <ScrollPane />

      </VBox>
   </right>
   <left>
      <VBox spacing="30" alignment="TOP_LEFT" >
         <children>
            <Label text="Firstname:" />
            <Label text="Lastname:" />
            <Label text="Address:" />
         </children>
      </VBox>
   </left>
</BorderPane>
```

Geben Sie ein Beispiel für die zwei Formen von JavaFX Eventhandlern (alt und seit Java8).
Es gibt einmal die ältere Methode mit der anonymen inneren Klasse:

```
Button btn1 = new Button("1"); //Button-Objekt wird erstellt
btn1.setOnAction(new EventHandler<ActionEvent>() { //eventHandler wird dem Button zugeordnet
  @Override
  public void handle(ActionEvent event) { /*wenn Button geklickt wird,
wird durch die handle Methode der Text des Buttons auf clicked geändert*/
    btn1.setText("Clicked");
  }
});

//Der Button 1 ändert sich auf clicked, wenn man auf ihn drückt
```

Und die Lambda Methode:

```
Button btn2 = new Button("2"); // Button wird erstellt
btn2.setOnAction(event -> btn2.setText("Clicked")); /*EventHandler wird dem Button zugewiesen
und durch einen Lambda-Ausdruck dargestellt*/

//"event -> btn2.setText("Clicked")" stellt EventHandler dar
//Wenn es zu einem ActionEvent kommt, wird die handle-Methode aufgerufen
//Button 2 ändert sich auf clicked, wenn gedrückt
```

Interpretieren Sie den folgenden Code

```
public class LayoutExample extends Application
{
    private TextField inputArea = new TextField();
    private TextArea outputArea = new TextArea();

    public static void main(String[] args){
        Application.launch(args);
    }

    @Override
    public void start(Stage stage){
        Label headerLbl = new Label("Please insert  Message in
TextArea!");
        Label inputLbl = new Label("Input: ");
        Label outputLbl = new Label("Output: ");
        Button okBtn = new Button("OK");
        HBox output = new HBox();
        output.getChildren().addAll(outputLbl, outputArea);

        okBtn.addEventHandler(MouseEvent.MOUSE_CLICKED,
                event ->  outputArea.appendText("You: " +
inputArea.getText() + "\n"));

        BorderPane root = new BorderPane();
        root.setTop(headerLbl);
        root.setRight(okBtn);
        root.setBottom(output);
        root.setLeft(inputLbl);
        root.setCenter(inputArea);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("SE2 Nachdenkzettel GUI");
        stage.show();
    }
}
```

Zeichnen Sie das Fenster mit Inhalt, das durch den JavaFX Code beschrieben wird. Was macht der Eventhandler?

Ein Textfeld “inputArea” und eine TextArea “outputArea” wird erstellt.
Im GUI-Fenster wird ein Label erstellt, dass die Überschrift darstellen soll: “Please insert Message in TextArea!”
Darunter ein Label für die inputArea : “Input: ” & ein Label für die outputArea: “Output :“
Dann wird ein “OK” Butten erstellt und eine HBox.
Der HBox werden dann die Kinder outputLbl und outputArea zugewiesen.
Mit dem Eventhandler wird festegelegt, dass wenn der Button gedrückt wird, ein Text in der outputArea ausgegeben wird, mit “You: ‘eigener Text’” und jeweils einen Umbruch in eine neue Zeile.
Dann wird eine BorderPane root erstellt mit der Anordnung der jeweiligen Eigenschaften.
Eine Scene mit root als Wurzelelement wird erstellt und das Fenster wird “SE2Nachdenkzettel GUI” als Titelbalken haben.
Dann wird die stage mit der Methode show aufgerufen.

Wie gehen Sie mit langlaufenden Operationen um, die durch einen Eventhandleer gestartet werden (z.B. langlaufende Datenbank-Abfrage)? Was müssen Sie dabei beachten? Warum?

Mit Threads wird das gleichzeitige Ausführen von Aufgaben ermöglicht, dadurch wird die Benutzeroberfläche nicht blockiert, während etwas anderes, wie eine langlaufende Operation, im Hintergrund läuft. Wenn langlaufende Operationen gelaufen sind, muss man die UI aktualisieren. Dies kann man mit 'Platform.runLater' machen. Auch Streams kann man benutzen, da diese unendlich sein können indem man zustandslose Operationen benutzt wie map() und filter(), hier muss man jedoch aufpassen, dass diese nicht gleichzeitig auf dasselbe zugreifen. Fehler müssen zudem behandelt werden, durch logging und Exceptions.