PM2, Aufgabenblatt 1

Programmieren 2 – Sommersemester 2017

Eclipse, Testen, Debugging, Vertragsmodell, Typhierarchien, Subtyp-Polymorphie

Ausgabedatum: ..... 27. März 2017

Kernbegriffe

Eine *integrierte Entwicklungsumgebung* (engl.: integrated development environment, kurz IDE) ist beim Programmieren neben der Programmiersprache ein wichtiges Werkzeug. Sie stellt u.a. Editoren für verschiedene Quelltexte zur Verfügung, lässt Quelltexte durch den Compiler übersetzen und bietet Unterstützung beim Debuggen von Programmen. Sie integriert viele der zentralen Tätigkeiten bei der Softwareentwicklung unter einer gemeinsamen Oberfläche.

Das *Vertragsmodell* (engl.: design by contract) von Bertrand Meyer formalisiert das Aufrufen einer Operation an einem Objekt. Das aufrufende Objekt wird von Meyer als *Klient* bezeichnet, das aufgerufene als *Dienstleister*. Die Beziehung zwischen den beiden wird als ein *Vertragsverhältnis* aufgefasst: Der Klient fordert eine Dienstleistung beim Dienstleister an, indem er eine Operation des Dienstleisters aufruft. Der Dienstleister muss die Dienstleistung nur erbringen, wenn der Klient bestimmte *Vorbedingungen* erfüllt, die der Dienstleister für die Operation formuliert; hält der Klient sich nicht an seinen Teil des Vertrags, muss der Dienstleister es auch nicht tun. Wenn der Klient jedoch die Vorbedingungen erfüllt, dann ist der Dienstleister verpflichtet, die Dienstleistung zu erbringen. In den *Nachbedingungen* wird festgelegt, wie sich die Dienstleistung auswirkt. Vor- und Nachbedingungen werden auch unter dem Begriff *Zusicherungen* zusammengefasst.

Konvention für Testklassen in PM2: Es soll *nicht* getestet werden, ob ein Dienstleister sicherstellt, dass seine Vorbedingungen vom Klienten eingehalten werden (keine Negativtests für Vorbedingungen).

Ab jetzt gilt immer: Bei neuem Code das Vertragsmodell einsetzen (wenn sinnvoll), intensiv testen und gründlich kommentieren (javadoc), auch wenn das nicht explizit gefordert ist.

In objektorientierten Programmiersprachen können Typen hierarchisch angeordnet werden, es entstehen *Typhierarchien*. Jede Klasse und jedes Interface in Java definiert einen Typ. Subtypen werden gebildet, indem ein Interface ein anderes erweitert oder indem eine Klasse eine andere Klasse erweitert bzw. ein Interface implementiert. *Subtypen* (speziellere Typen) bieten immer mindestens die Schnittstelle ihrer *Supertypen* (allgemeinere Typen), darüber hinaus können sie diese Schnittstelle um eigene Operationen erweitern.

An eine Referenzvariable eines bestimmten Typs können Exemplare eines spezielleren Typs gebunden werden; dies bezeichnet man als *Subtyp-Polymorphie*. Der deklarierte Typ einer Variablen ist dabei ihr *statischer Typ*. Ist eine Variable zur Laufzeit an ein Objekt gebunden (und hält damit eine gültige Referenz auf ein Objekt), dann bestimmt die Klassenzugehörigkeit dieses Objekts den *dynamischen Typ* der Variablen. Ruft man an einer Referenzvariablen eine Operation auf, hängt es von ihrem dynamischen Typ ab, welche Implementation tatsächlich aufgerufen wird (aus welcher Klassendefinition die gerufene Methode stammt); dies nennt man *dynamisches Binden*.

Will man ein Exemplar eines spezielleren Typs, das an eine Variable mit einem allgemeineren statischen Typ gebunden ist, wieder unter seinem speziellen Typ ansprechen (um auch Operationen aufrufen zu können, die nur der speziellere Typ besitzt), ist eine *Typzusicherung* (auch: *Downcast*) notwendig. In Java geschieht dies durch das Schreiben des spezielleren Typs in runden Klammern vor einem Ausdruck. *Vorsicht*: Eine Typzusicherung ohne vorherigen *Typtest* (in Java mit `instanceof`) kann zur Laufzeit zu einer `ClassCastException` führen, wenn das Exemplar nicht dem spezielleren Typ entspricht.

Um in Java eine Klasse zu einem Subtyp eines Interfaces zu machen, muss im Klassenkopf mit dem Schlüsselwort `implements` erklärt werden, dass die Klasse das Interface implementiert. Dabei handelt es sich um reine *Typ-Vererbung*, d.h. es werden lediglich Operationsdeklarationen, aber keine implementierenden Methoden vererbt. Wenn eine Klasse, von der Exemplare erzeugt werden können, ein Interface implementiert, müssen für alle im Interface definierten Operationen in dieser Klasse Methoden implementiert werden.

Eine Klasse kann in Java auch mehrere Interfaces implementieren und auch ein Interface kann von mehreren Interfaces erben (mit dem Schlüsselwort `extends`). Dies nennt man *Mehrfach(typ)vererbung* oder *multiples Subtyping*.

Aufgabe 1.1: Wechsel der Entwicklungsumgebung

BlueJ ist gut für den Einstieg in die objektorientierte Programmierung geeignet, für die Arbeit an größeren Projekten jedoch nicht. Deshalb steigen wir an dieser Stelle auf die Entwicklungsumgebung *Eclipse* um. Eclipse ist frei verfügbar und wird auch im professionellen Kontext häufig eingesetzt. Auf den Rechnern der Labore ist Eclipse bereits installiert. (Für die Installation auf eigenen Rechnern könnt ihr Eclipse von der Website des Eclipse-Projekts herunterladen, http://www.eclipse.org/. Eclipse ist in verschiedenen Varianten verfügbar, für PM2 ist die „Eclipse IDE for Java Developers“ sinnvoll.)

In dieser Aufgabe geht es darum, Eclipse kennen zu lernen. Es macht also nichts, wenn ihr den Quelltext nicht komplett versteht. Für den ersten Umgang mit Eclipse werden wir das Projekt *Mediathek* nutzen.

1.1.1 Auf den Laborrechnern ist Eclipse in der neuesten Version (2017: Neon) installiert. Beim ersten Starten erhaltet ihr einen Dialog, in dem ihr den so genannten Workspace angebt. Dies ist ein Ordner im Dateisystem, in dem Eclipse Projekte ablegt und verwaltet. Wählt hier einen Unterordner in eurem Home-Verzeichnis (in der Hochschule als Netzlaufwerk „Z:“ eingebunden, wählt also beispielsweise „Z:\PM2\eclipse_workspace“). Falls dieser Dialog nicht erscheint, kann der Workspace über *File>Switch Workspace>Other…* geändert werden.

1.1.2 Für einen reibungslosen Austausch von Daten über Plattformgrenzen hinweg sollte die Codierung für Textdateien auf UTF-8 gestellt werden. Geht dazu auf *Window>Preferences>General>Workspace* und stellt dort das *Text file encoding* auf *Other: UTF-8* ein.

Ladet anschließend die Datei MediathekBlatt01.zip aus dem pub-Bereich auf euren Rechner. In diesem Archiv befindet sich ein Eclipse-Projekt, das ihr importieren müsst. Wählt dazu in Eclipse *File>Import...* und dann *Existing Projects into Workspace* (unter *General*) und gebt im folgenden Schritt für *Select archive file* die Datei MediathekBlatt01.zip an.

Wechselt mit *Window>Open Perspective>Java* in die so genannte *Java-Perspektive*. Wenn auf der linken Seite kein *Package Explorer* geöffnet sein sollte, öffnet ihn mit *Window>Show View>Package Explorer*. Klappt das Mediathek-Projekt auf und schaut euch an, wie Eclipse Klassen und deren Quelltexte darstellt (per Doppelklick auf eine Klasse wird ein Editor geöffnet).

1.1.3 Macht euch klar, was die verschiedenen Fenster der Eclipse Java-Perspektive anzeigen. Findet heraus, was der *Outline-View ist*, und erklärt es bei der Abnahme.

1.1.4 Eclipse kann euch beim Einhalten der Quelltextkonventionen unterstützen. Mit *Strg+Shift+F* startet ihr das automatische *Code-Formatting*. Unter *Preferences>Java>Code Style>Formatter* müsst ihr es noch an unsere Konventionen anpassen. Für PM2 gibt es neue, erweiterte Konventionen. Formatiert eure Quelltexte entsprechend. Exportiert euren Formatter als Datei, damit ihr die Konventionen auch auf anderen Rechnern importieren könnt.

Aufgabe 1.2: Mediathek kennen lernen

Mit dieser Aufgabe lernt ihr die Mediathek genauer kennen. Es handelt sich um ein Entwicklungsprojekt, das uns über Teile des Praktikums begleiten wird. Die Mediathek ist eine Software für Mediathekare. Ein Mediathekar bedient die Programmoberfläche, leiht Medien an Kunden aus und nimmt diese wieder zurück. Die euch vorliegende Version der Mediathek enthält noch nicht die gesamte benötigte Funktionalität. In den nächsten Wochen werdet ihr die Software ausbauen. Die grafische Benutzungsoberfläche wird uns gestellt. Wir sind für die Umsetzung der benötigten Funktionalität verantwortlich.

1.2.1 In BlueJ lassen sich Exemplare interaktiv erzeugen und beliebige Operationen an diesen Exemplaren aufrufen. Das ist eine besondere Eigenschaft von BlueJ, auf die wir nun verzichten müssen. Üblicherweise werden Java-Programme über die Methode `public static void main(String[] args)` gestartet. Wählt im Package Explorer die Klasse `StartUpMediathek_Blatt01` aus und startet die Methode `main`, indem ihr *Run As>Java Application* aus dem Kontextmenü der Klasse auswählt. Spielt mit der Oberfläche herum. Welche Funktionalität ist bereits implementiert? Was wird noch nicht unterstützt, sollte aber in keiner guten Mediathek fehlen? **Schriftlich.**

1.2.2 Nun schauen wir uns den Quelltext näher an. Im src-Ordner findet ihr mehrere Klassen und Interfaces. Zu Beginn sind nur einige Klassen und Interfaces für uns interessant. Diese Aufgabe dient dazu, sich erst einmal zurechtzufinden. **Erarbeitet euch schriftlich** die Antworten zu den folgenden Fragen, so dass ihr sie bei der Abnahme erklären könnt:

- ☐ Wie viele Test-Klassen gibt es?

- ☐ Die Klassen, die die grafische Benutzungsoberfläche gestalten, sind für uns vorläufig nicht wichtig. Woran kann man sie erkennen?
- ☐ Die Benutzungsoberfläche arbeitet mit einem `VerleihService`. Schaut euch das Interface an. Welche Dienstleistungen bietet es durch seine Operationen an?

1.2.3 **Zeichnet ein UML-Klassendiagramm**, ausgehend vom Interface `VerleihService`. Das Klassendiagramm soll den `Verleihservice` zeigen sowie alle Klassen und Interfaces, die der `VerleihService` direkt oder indirekt benutzt. Operationen und Variablen müssen nicht eingezeichnet werden. Das Diagramm werdet ihr auf den nächsten Aufgabenblättern brauchen.

Ihr könnt das Diagramm händisch auf einem Blatt Papier oder auch digital zeichnen: mit *Visio* oder einem der auf den Labor-Rechnern installierten UML-Editoren oder dem sehr einfachen UML-Editor *Violet*, zu finden unter http://violet.sourceforge.net/.

Aufgabe 1.3: Aufgabenteilung über Interfaces kennen lernen

Das GUI-Team arbeitet parallel zum PM2-Team. Damit die beiden Teams möglichst unabhängig voneinander arbeiten können, haben sie sich auf das Interface `VerleihService` geeinigt, über das die notwendigen Informationen ausgetauscht werden.

1.3.1 Das GUI-Team wollte nicht auf die Implementation des PM2-Teams warten und hat deshalb schon einmal einen so genannten *Dummy* geschrieben, der das Interface `VerleihService` implementiert. Untersucht die Klasse `DummyVerleihService`. Welchem Zweck dient sie? Implementiert sie das Interface so, wie die Autoren von `VerleihService` es sich gedacht haben? Welche Grenzen hat die Implementation? **Schriftlich.**

1.3.2 Zum Glück gibt es mittlerweile eine Implementation des PM2-Teams, die besser funktioniert: `VerleihServiceImpl`. Wir wollen nun den `DummyVerleihService` ersetzen. Hierzu müssen wir erstmal herausfinden, an welcher Stelle ein Objekt dieser Klasse erzeugt wird. Klickt im Editor der Klasse `DummyVerleihService` mit rechts auf den Konstruktornamen und wählt *References->Project*. Es öffnet sich eine View, in der angezeigt wird, wo der Konstruktor überall aufgerufen wird. Dies ist in unserem Programm nur eine Stelle. Mit einem Doppelklick könnt ihr nun dorthin springen und stattdessen den Konstruktor des `VerleihServiceImpl` einsetzen. **Schreibt auf**, warum ihr so einfach ein Objekt einer anderen Klasse an diese Stelle verwenden könnt.

1.3.3 Startet das Programm und testet die Oberfläche. Welcher Fehler tritt jetzt noch beim Verleih eines Mediums auf? Mit diesem Problem beschäftigen wir uns in einer späteren Aufgabe.

1.3.4 In den kommenden Wochen werden wir an demselben Projekt weiterarbeiten. Sorgt ab jetzt nach *jedem* Praktikumstermin dafür, dass beide Programmierpartner die Mediathek auf ihren Netzlaufwerken zur Verfügung haben. Ihr könnt das Projekt aus Eclipse exportieren mit dem Menüpunkt *File->Export*. Dort wählt ihr *General->Archive File*. **Führt bei der Abnahme vor, dass ihr das Projekt mit beiden Usern öffnen könnt.**

Aufgabe 1.4: Debugger verwenden

Ihr habt gesehen, dass die Mediathek noch fehlerhaft ist: die Ausleihe funktioniert nicht. Wir verwenden nun den Debugger, um herauszufinden, in welcher Klasse der Fehler liegt. Dabei beginnen wir mit unserer Suche in der Klasse, die die Interaktion an der Benutzungsschnittstelle steuert.

1.4.1 Öffnet die Klasse `AusleihWerkzeug`. Setzt in die erste Zeile der Methode `leiheAusgewaehlteMedienAus` einen *Haltepunkt* (engl. *breakpoint*). Findet heraus, wie das geht, z.B. über die in Eclipse integrierte Hilfe.

1.4.2 Um eine Anwendung zu debuggen, muss sie im Debug-Modus gestartet werden. Startet erneut die Anwendung, dieses Mal jedoch mit *Debug As->Java Application*. Öffnet dann die *Debug-Perspective* auf die gleiche Weise, wie ihr zuvor die Java-Perspektive geöffnet habt.

Selektiert nun in der Mediathek einen Kunden und mehrere Medien. Drückt dann den Button *ausleihen*. Die Anwendung bleibt an eurem Haltepunkt stehen. Die Fenster der Debug-Perspektive zeigen den aktuellen Zustand des Programms. Oben rechts könnt Ihr die Werte der Variablen betrachten. Oben links findet ihr den *Aufruf-Stack* (im *Debug-View*). Die aktuelle Position im Quelltext wird im mittleren Fenster angezeigt.

1.4.3 Führt die ersten 3 Zeilen der Methode mit Hilfe des Debuggers schrittweise aus, mit dem Button *step over (F6)*. Beobachtet, wie oben rechts die neuen lokalen Variablen erscheinen.

1.4.4 Springt nun in die Methode `verleiheAn` mit dem Button *step into (F5)*. Beobachtet dabei, wie sich die Variablenanzeige oben rechts ändert und was mit dem Aufruf-Stack passiert. Macht euch insgesamt klar, was die verschiedenen Fenster anzeigen und was die verschiedenen Buttons über dem Debug-View bedeuten. Lauft so lange durch das Programm, bis ihr die fehlerhafte Programmzeile findet. Diese verbessert ihr aber noch nicht in dieser Aufgabe! Wenn ihr in der letzten Zeile von `verleiheAn` angekommen seid, drückt ihr auf den Button *Resume (F8)*, um das Programm weiterlaufen zu lassen.

1.4.5 Führt bei der Abnahme vor, wie ihr den Debugger verwendet, um von eurem Haltepunkt bis zum Ende der Methode `verleiheAn` zu gehen. Erläutert bei der Abnahme, wann man den Debugger einsetzen sollte und wann nicht.

Aufgabe 1.5: Fehler durch Tests finden und beheben

Für die Klasse `VerleihServiceImpl` gibt es bereits eine Testklasse, die den Fehler aus Aufgabe 1.6 jedoch nicht findet! In einem solchen Fall wird zuerst ein Testfall geschrieben, der den Fehler aufdeckt, und danach wird der Fehler in der getesteten Klasse behoben.

1.5.1 Ergänzt die Testklasse `VerleihServiceImplTest` um einen passenden Testfall, der den gefundenen Fehler aufdeckt.

1.5.2 Da nun ein Test existiert, der den Fehler findet, könnt ihr diesen in `VerleihServiceImpl` beheben.

1.5.3 Schreibt mindestens einen weiteren Testfall. Es bietet sich das Testen des Zurücknehmens an.

Aufgabe 1.6 Vertragsmodell einsetzen

Innerhalb der Mediathek wird das Vertragsmodell verwendet. In Java gibt es keine Sprachunterstützung für das Vertragsmodell; stattdessen muss man sich mit Konventionen behelfen. In PM2 sehen diese Konventionen wie folgt aus: Der Dienstleister *deklariert* seine Vor- und Nachbedingungen gegenüber dem Klienten, indem er sie im javadoc-Kommentar einer Operation mit `@require` und `@ensure` angibt. Weiterhin *überprüft* der Dienstleister die Einhaltung der Vorbedingungen im Rumpf der implementierenden Methode mit assert-Anweisungen. Die eigenen Nachbedingungen werden *nicht* überprüft.

1.6.1 Die assert-Anweisungen eines Java-Programms werden nur dann ausgeführt, wenn man die JVM entsprechend konfiguriert; ansonsten werden sie ignoriert. Damit die Vorbedingungen auch wirklich vom Dienstleister überprüft werden, müssen wir Assertions in der Mediathek aktivieren. Lasst alle im Projekt enthaltenen Tests durchlaufen, indem ihr im Kontextmenü des *src*-Ordners *Run As->JUnit* Test auswählt. Einer der enthaltenen Tests prüft, ob Assertions aktiviert sind; er wird vermutlich *nicht* fehlschlagen, wenn ihr Eclipse Neon verwendet! Um den Effekt dieses Tests beobachten zu können, müssen wir kurzfristig eine Option für JUnit abschalten:
Wählt *Window->Preferences->Java->JUnit* und deselektiert rechts die Option „add -ea to VM arguments…“. Lasst die Tests erneut durchlaufen, ein Testfall sollte nur fehlschlagen.

Geht zu der Stelle im Quelltext des fehlgeschlagenen Tests und folgt der Anleitung in den Implementationskommentaren, um Assertions zu aktivieren. Führt die Tests erneut aus. Sie sollten jetzt alle durchlaufen. Der Parameter `-ea` (für *enable assertions*) wird ab jetzt bei jedem Programmstart an die virtuelle Maschine übergeben. Wenn ihr den Rechner wechselt, müsst ihr diese Einstellung erneut vornehmen, entweder in den *JRE*- oder den *JUnit-Preferences*!

1.6.2 Seht Euch den Konstruktor der Klasse `Verleihkarte` genau an. An welchen Stellen werden Vor- und Nachbedingungen *deklariert* und *überprüft*? Wie und wo werden sie innerhalb von Methodenkommentaren angegeben und nach welchen Regeln erfolgt die Überprüfung mit assert-Anweisungen? Wie sind diese Anweisungen syntaktisch aufgebaut? **Findet geeignete Dokumentation zu assert-Anweisungen in Java!**

1.6.3 Schaut euch die im Interface `VerleihService` deklarierten Vorbedingungen an. Diese werden noch nicht von der implementierenden Klasse `VerleihServiceImpl` geprüft. Dies müssen wir ändern! Arbeitet hierfür zuerst das Infoblatt *Vertragsmodell in Eclipse* durch. Jetzt könnt ihr alle fehlenden assert-Anweisungen einfügen. Erklärt euren BetreuerInnen an konkreten Beispielen aus dem VerleihService, warum die gemachten Vorbedingungen sinnvoll sind.

1.6.4 Das Interface `MedienbestandService` besitzt noch keine Vor- und Nachbedingungen. Schreibt sinnvolle Vertragsbedingungen (nicht nur, dass ein Parameter != null sein soll) in die Kommentare und ergänzt danach die benötigten assert-Anweisungen in der implementierenden Klasse `MedienbestandServiceImpl`.

Aufgabe 1.7 Typvererbung in der Mediathek

Die Mediathek soll nun auch Videospiele verleihen können. Da die Mediathek bereits das Interface `Medium` enthält, ist es recht einfach, neue Medienarten hinzuzufügen. Die Variablen, die für die Ausleihe verwendet werden, sind statisch auf `Medium` getypt.

1.7.1 Um Videospiele verleihen zu können, müssen diese im Programm repräsentiert werden. Hierfür muss es eine Klasse `Videospiel` mit zugehörigem Test geben. Schreibt im Sinne des Test-First-Ansatzes zuerst den Test mit einer leeren Videospiel-Klasse. Danach implementiert ihr die Methoden des Videospiels. Die Klasse `Videospiel` soll, genau wie die Klassen `CD` und `DVD`, das Interface `Medium` implementieren.

1.7.2 Zusätzlich sollen Videospiele noch das System speichern, auf dem das Spiel lauffähig ist, beispielsweise Xbox One, WII U oder PS4. Entscheidet welchen Typ ihr für das Speichern dieser Information verwenden wollt. Erweitert `VideospielTest` und implementiert danach die benötigten Änderungen im `Videospiel`.

1.7.3 Nun sollen auch Videospiele an der Oberfläche angezeigt werden. In der Klasse `MedienEinleser` gibt es eine Methode `leseMediumEin()`. Hier findet ihr eine auskommentierte Zeile, in der Videospiele erzeugt werden. Schaut euch die Parameterreihenfolge im Kontruktoraufruf an. Stimmt dieser mit eurem Konstruktor überein? Nehmt die Kommentarzeichen weg und probiert aus, ob nun auch Videospiele zur Ausleihe angeboten werden.
Notiert im Quelltext, an welchen Stellen sich der dynamische vom statischen Typ einer Variablen unterscheidet.

1.7.4 *Zusatzaufgabe*: Wandelt die if-Anweisung in der Methode `leseMediumEin()` in eine switch-Anweisung um. Seit welcher Java-Version ist dies möglich?

Aufgabe 1.8 Typzusicherung und dynamisch gebundener Operationsaufruf

Euch ist sicherlich aufgefallen, dass in der Programmoberfläche in dem rechten freien Bereich mit der Überschrift „Ausgewählte Medien“ stets nur eine weiße Fläche zu sehen ist. Dies wollen wir nun ändern und alle Details zu den selektierten CDs, DVDs und Videospielen in der Medientabelle anzeigen lassen.

1.8.1 Öffnet die Klasse `AusleihWerkzeug`. Sucht die Methode `zeigeAusgewaehlteMedien`. Sie wird immer aufgerufen, wenn sich die Selektion in der Medien-Tabelle ändert. Folgt dem Methodenaufruf `setMedien`. In dieser Methode ist bereits ein Quelltextgerüst enthalten. Hier wollen wir nun die spezifischen Medieninformationen an der Benutzungsschnittstelle anzeigen lassen. Macht zuerst zu jedem Medium einen Typtest auf `CD, DVD` und `Videospiel`. Führt danach eine Typzusicherung durch. Fragt dann die spezifischen Informationen zur CD, DVD bzw. zum Videospiel an den Exemplaren ab, und fügt diese Informationen geeignet in der `TextArea` ein. Eure Änderungen sollten in der Programmoberfläche zu einem sichtbaren Ergebnis führen.

1.8.2 Jetzt vereinfachen wir den Quelltext, indem wir einen dynamisch gebundenen Operationsaufruf einbauen. Ergänzt hierzu die Schnittstelle des Interfaces `Medium` um die Operation `String getFormatiertenString()`. Denkt an das Vertragsmodell. Schreibt einen Schnittstellenkommentar, der wiedergibt, dass eine implementierende Methode eine Text-Repräsentation mit allen Eigenschaften des speziellen Mediums zurückgibt.

1.8.3 *Erstellt eine auskommentierte Kopie der Methode `setMedien` für die spätere Abnahme der Aufgabe*. Baut nun euren in 1.8.2 geschriebenen Quelltext um, so dass die neue Operation `getFormatiertenString()` verwendet wird, um ein Medium als Text darzustellen. **Schreibt die Antworten zu den folgenden Fragen auf**: Warum braucht ihr nun kein `instanceof` mehr? Was ist dynamisches Binden?

Aufgabe 1.9: Kooperatives Arbeiten an Quelltexten

In den kommenden Wochen werdet ihr an der Mediathek weiterarbeiten. Sorgt ab jetzt nach *jeder Änderung* dafür, dass beide Programmierpartner die aktuelle Version der Mediathek zur Verfügung haben. Dazu gibt es verschiedene Möglichkeiten, von denen ihr euch mit mindestens zweien vertraut machen sollt: Dem Austausch über das Dateisystem (einfach) und der Benutzung eines Versionskontrollsystems (hier: Git, anspruchsvoller).

1.9.1 Ihr könnt das Projekt aus Eclipse exportieren mit dem Menüpunkt *File->Export*. Dort wählt ihr *General->Archive File*. **Führt bei der Abnahme vor, dass beide Teampartner das Projekt über ihren Account öffnen könnt.** Das gespeicherte Projekt dient euch auch als Sicherungspunkt.

1.9.2 Verwendet Git (integriert in Eclipse/über die Kommandozeile per git-Bash/per SourceTree), um ein Repository zu teilen (auf Bitbucket/Github/…). Checkt eure Version der Mediathek dort ein. **Führt bei der Abnahme vor, dass beide das Projekt aus dem Remote Repository laden können.**

Einen Einstieg in Git liefern die Folien aus PR1 (Level 4). Weiterhin solltet ihr die ersten beiden Kapitel des online unter **progit.org** verfügbaren Buches „Pro Git“ von Scot Chacon lesen. Wir empfehlen als minimalen Workflow, keine Branches zu verwenden; ihr seid aber frei, es anders zu machen, wenn ihr es besser wisst.