

1. Bringe zunächst das System zum Laufen und finde heraus, wie es funktioniert ✓

2. Überprüfe durch manuelles Testen am laufenden System, ob alle Anforderungen implementiert sind wie oben beschrieben ✓

3. Dokumentiere sowohl Einhaltung als auch evtl. Abweichungen von den beschriebenen Anforderungen, bzw. der Systembeschreibung → Dokumentation wird bewertet (Inhalt, nicht Form)

✓

Die Klasse **Karte** ist sehr einfach - sie symbolisiert ein Konto und

- verfügt über die Eigenschaft PIN.
Anforderung erfüllt: `private String pin;`
- Diese Eigenschaft wird beim Anlegen einer Karte gesetzt
Anforderung erfüllt: `public Karte(String pin){ ... this.pin = pin;}`
- und kann danach nicht mehr verändert werden.
Anforderung erfüllt: Attribut ist Package-private, Call-Hierarchie zeigt ledigliche Verwendung im Konstruktor und in Methode `istKorrekt()`
- Jede Karte verfügt über die Möglichkeit, eine am Automat eingetippte PIN mit der in der Karte gespeicherten PIN abzugleichen.
Anforderung erfüllt:

```
if (karte.istKorrekt(pin)) {  
    pinKorrekt = true;  
} else ...
```
- Der Automat darf nur dann Geld auszahlen, wenn die korrekte PIN eingegeben wurde.
Anforderung erfüllt:

```
if (pinKorrekt) {  
    bargeld -= summe;  
    return summe;  
}
```


Variable `pinKorrekt` wird vorher in Methode `eingeben()` gesetzt.
- Eine PIN muss vierstellig sein und aus Ziffern bestehen (keine Buchstaben, keine Sonderzeichen)
Anforderung erfüllt: Beim Anlegen einer PIN ist die Länge der PIN durch die for-Schleife `for (int i = 0; i < 4; i++){...}` auf genau 4 festgelegt, zudem wird bei einer PIN-Eingabe auf eine Länge von 4 Zeichen geprüft: `if (pin.length() != 4) {...}` Beim Erstellen der PIN in `erzeugePin()` werden zusätzlich durch die `Math.random();`-Funktion ausschließlich Zahlen in die PIN geschrieben. Beim Einlesen einer eingegebenen PIN durch den Nutzer wird die Eingabe zu einem Integer umformatiert: `Integer.parseInt(pin)`. Ist dies nicht möglich, wird eine `NumberFormatException` geworfen. So sind Buchstaben und Sonderzeichen ausgeschlossen.
- Es muss möglich sein, vierstellige PINs anzulegen, die mit einer oder mehreren Nullen beginnen.
Anforderung nicht erfüllt: Es konnte festgestellt werden, dass die Routine zum Erzeugen einer PIN die Ziffer Null als Bestandteil der PIN gar verhindert! Trotz der

bereits gegebenen Funktionalität von `Math.random()` unter anderem die GKZ 0,0 zurückzugeben, wird in der darauffolgenden Potenzierung die Ziffer 0 herausgefiltert, da nur Zahlen ab 1 zum Verlassen der do-while-Schleife führen und eine 0,000 gar in einer Endlosschleife enden würde:

```
do {  
    ziffer *= 10;  
} while (ziffer < 1.0);
```

Die Klasse **Anwendung** steuert den Geldautomat.

- Sie ist als textbasierte Konsolenanwendung ausgeführt und Anforderung erfüllt.
- bietet dem Benutzer alle Möglichkeiten, um am Geldautomat Geld abzuheben. Anforderung erfüllt: Mit viel Wohlwollen möglich, das Programm erlaubt keine Fehlritte. Um Geld auszuzahlen, muss die Reihenfolge "Automat bestücken", "Karte einschieben" "PIN eingeben" und "Geld auszahlen" explizit eingehalten werden, da alles andere zu unbehandelten Exceptions führt.

Die Klasse **Geldautomat** schließlich ist das Zentrum.

- Sie implementiert die Funktionalitäten, die von der Klasse Anwendung gesteuert werden. Anforderung erfüllt.
- Der Automat startet im Zustand "Kein Geld". Anforderung erfüllt: `private int bargeld = 0;` (in Klasse Geldautomat)
- Der Automat kann nur bestückt werden, wenn sich keine Karte in ihm befindet. Anforderung erfüllt:

```
if (this.karte != null) {  
    throw new IllegalStateException("Automat darf nicht  
    während einer Transaktion bestückt werden!");  
} (in Klasse Geldautomat, Methode bestücken(int bargeld)
```
- Wird eine Karte in einen leeren Automat eingeschoben, wird sie wieder ausgeworfen und eine Fehlermeldung ausgegeben. Anforderung nicht erfüllt:

```
public void einschieben(Karte karte) {  
    if (this.karte != null) {  
        throw new IllegalStateException("Es befindet sich bereits eine  
        Karte im Automat!");  
    }  
    this.karte = karte;  
}
```

Im Code (Siehe oben) wird ausschließlich geprüft ob eine Karte im Automaten ist und nicht, ob dieser bestückt ist. Das Programm läuft normal weiter.

- Befindet sich eine Karte im Automaten, muss die passende PIN eingegeben werden, damit mit der Auszahlung fortgefahren werden kann.

Anforderung ist teilweise erfüllt:

Szenario 1: Abgehobener Betrag ist kleiner oder gleich als der Füllstand

```
if (pinKorrekt) {  
    bargeld -= summe;  
    return summe;  
} (in Klasse Geldautomat, Methode auszahlen(summe))
```

In diesem Szenario ist die Anforderung erfüllt.

Szenario 2: Abgehobener Betrag ist größer als der Füllstand:

```
if (bargeld < summe) {  
    int ausbezahlt = bargeld;  
    bargeld = 0;  
    return ausbezahlt;  
}
```

In diesem Szenario ist die Anforderung nicht erfüllt, da direkt der Kontostand 0 zurückgegeben wird und die Überprüfung aus Szenario 1 somit nie erreicht wird.

- und es kann so lange Geld ausbezahlt werden, wie der Automat Bargeldreserven hat.

Anforderung erfüllt:

```
if (bargeld < summe) {  
    int ausbezahlt = bargeld;  
    bargeld = 0;  
    return ausbezahlt;  
} (in public int auszahlen(int summe))
```

- Der jeweilige Auszahlungsbetrag muss dabei zwischen 5 und 500 Taler liegen.

Anforderung erfüllt:

```
public int auszahlen(int summe) {  
    if (summe < 5 || summe > 500) {  
        throw new IllegalArgumentException(  
            "Betrag muss zwischen 5 und 500 Geld liegen!");  
    }  
}
```

- Da der Automat nur Scheine und keine Münzen hat, muss jeder Auszahlungsbetrag durch 5 teilbar sein.

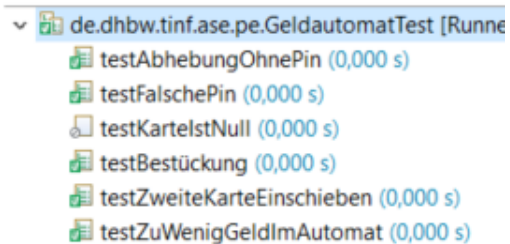
Anforderung nicht erfüllt: dies wird an keiner Stelle im Code geprüft.

- Wird in einer Transaktion mehr Geld angefordert als der Automat Bargeld besitzt, zahlt der Automat das aus, was er noch hat und geht dann in den Zustand "Kein Geld" über.

Anforderung erfüllt, siehe Anforderung: "und es kann so lange Geld ausbezahlt werden, wie der Automat Bargeldreserven hat"

4. Lass die Unit Tests laufen, bzw. bringe sie zum Laufen (d.h. die vorhandenen Tests müssen vor der Weiterentwicklung zunächst erfolgreich durchlaufen). ✓

```
@Test (expected = IllegalStateException.class)
@Ignore
public void testKarteIstNull() {
    Geldautomat geldautomat = new Geldautomat();
    geldautomat.bestücken(1000);
    geldautomat.einschieben(null);
    fail("Expected IllegalArgumentException!");
}
```



Ein Test wird ignoriert durch eine bereits vorhandene “@Ignore”-Annotation. Dies begründet sich wohl in der Tatsache, dass der Test nicht realitätsnah ist. Die Methode einschieben() der Klasse Geldautomat wird ein einziges Mal in der Klasse Anwendung in der Methode karteEinschieben() aufgerufen. Dabei kann der hier dargestellte Testfall nicht auftreten, da in der Zeile zuvor das Objekt instanziiert wird. Sollte es hierbei zu einem Fehler kommen (karte = null), kann die Methode einschieben() nicht erreicht werden.

Alternativ könnte man nun in der Methode einschieben() den Parameter auf Null prüfen, jedoch ist die dabei auszugebende Fehlermeldung sehr fragwürdig (“Es darf keine leere Karte eingeschoben werden”).

Deshalb hat man sich dazu entschieden, diesen Test vollständig zu entfernen.

5. Ergänze – falls notwendig - die vorhandenen Tests, so dass alle Anforderungen über Unit Tests abgedeckt sind (alle Unit Tests müssen am Ende erfolgreich durchlaufen) → Testabdeckung und Lauffähigkeit der Tests werden bewertet

✓

Folgende Tests sind hinzugefügt worden:

- Prüfen der PIN auf Vierstelligkeit
 - Testen einer PIN mit weniger als 4 Zahlen → testPin3Stellig()
 - Testen einer PIN mit mehr als 4 Zahlen → testPin5Stellig()
- Prüfen, ob die PIN nur aus Ziffern besteht → testPinNurZiffern()
- Prüfen, ob die PIN aus einer oder mehreren führenden Nullen bestehen kann
 - Da die einzigen beiden Methoden, die an der Erstellung einer PIN beteiligt sind (karteEinschieben() und erzeugePin()) private sind, konnte zunächst kein Test geschrieben werden. Nach Entfernen des Sichtbarkeitsmodifizierers “private” der Methode erzeugePin() ist diese package-private und kann getestet werden.
 - → testPinMitFuehrendenNullen()
 - Sobald die Methode erzeugePin() korrigiert wurde (kann auch PINs mit einer oder mehreren führenden Nullen zurückgeben), wird dieser Test schließlich ohne Fehler erfolgen (nur wenn bei 1000 Versuchen eine PIN mit mind. einer führenden Null erstellt wird).
- Prüfen, ob der Automat nur bestückt werden kann, wenn sich keine Karte in ihm befindet → testBestueckenNurOhneKarte()
- Prüfen, ob die Karte wieder ausgeworfen wird, wenn sie in einen leeren Automaten eingeschoben wird → testKarteInLeerenAutomat()

- Sobald die Methode einschieben() um eine Abfrage der Geldmenge im Automat erweitert wird, wird dieser Test schließlich ohne Fehler erfolgen.
- Bei testZuWenigGeldImAutomat() muss noch die PIN eingegeben werden, damit Geld abgehoben und der Test richtig durchlaufen kann.
- Prüfen, ob der Auszahlungsbetrag zwischen 5 und 500 Talern liegt
 - → testAuszahlenWenigerAls5()
 - → testAuszahlenMehrAls500()
- Prüfen, ob geprüft wird, ob der Auszahlungsbetrag durch 5 teilbar ist → Passend zu den Sollszenarien 1-3 in Aufgabe 6: testAuszahlungsbetragDurch5Teilbar(), testAuszahlenBestandWenigerAls5(), testTeilweiseAuszahlung()
- Gemäß Aufgabe 7, muss überprüft werden, ob bei dreimaliger Falscheingabe der PIN ein Statusübergang zu "Bereit" durchgeführt wird.
 - → testDreimalFalschePin()

6. Verbessere das Design des Systems durch die Anwendung eines oder mehrerer passender Entwurfsmuster. Begründe deine Entscheidung für/gegen den Einsatz bekannter Muster → Verwendung von Entwurfsmuster(n) und Dokumentation werden bewertet



- Bug mit Auszahlung des kompletten Betrags ohne PIN, wenn man mehr auszahlen möchte als vorhanden war, korrigiert.
- Beschluss für Thematik mit Auszahlungsbetrag größer als Geldbestand und mit Betrag Vielfaches von 5:
 - Ist-Szenario:
 - z.B. Bestand = 3 Taler, Auszahlungsbetrag = 5 Taler
 - Funktioniert mit dem zu Beginn gegebenen Code; Auszahlungsbetrag ist Vielfaches von 5, der Automat gibt aus, was er hat und geht dann in den Zustand "Kein Geld" über. Es wird nicht mehr geprüft, ob der tatsächliche Auszahlungsbetrag ein Vielfaches von 5 ist. 3 Taler können aber de facto eigentlich nicht ausgegeben werden, da der Automat nur Scheine beinhaltet -> Konflikt
 - Sollszenarien:
 - Szenario 1: Bestand = 10 Taler, Auszahlungsbetrag = 6 Taler
 - Schlägt fehl: Auszahlungsbetrag muss Vielfaches von 5 sein.
 - Szenario 2: Bestand = 3 Taler, Auszahlungsbetrag = 5 Taler
 - Schlägt fehl: 5 Taler sind zwar ein Vielfaches von 5, aber 3 kann nicht ausbezahlt werden. Es wird also nichts ausbezahlt.
 - Szenario 3: Bestand = 6 Taler, Auszahlungsbetrag = 10
 - Schlägt teilweise fehl: Der Auszahlungsbetrag ist zwar Vielfaches von 5, Bestand ist aber zu niedrig. Deshalb werden nur 5 Taler ausgezahlt.
- **Design Patterns:**
 - **Singleton:**
 - Käme lediglich für die Klasse Geldautomat in Frage, da GeldAutomat pro Programmdurchlauf (main()) einmalig instanziiert wird.
 - Wird umgesetzt.

- Umsetzung: Durch das Umsetzen des Singleton in der Klasse Geldautomat musste für die korrekte Durchführung der Tests die Methode resetAutomat() erstellt werden.
- **Decorator:**
 - Ungeeignet, da es keine Umstände gibt, in denen eine bestehende Klasse um bestimmte Methoden/Attribute erweitert werden sollte.
- **Observer:**
 - Ungeeignet, pro Klasse wird jeweils nur ein Objekt erstellt, die Hierarchie mit 1:1:1 überschaubar. Falls eine "Benachrichtigung" nötig ist kann dies auf direktem Wege erfolgen.
- **Abstract Factory:**
 - Ungeeignet, da es keine Objekte gibt, die etwas gemeinsam haben (keine common-Themes)
- **Factory Method:**
 - Ungeeignet, es ist keine große Vielfalt vorhanden.
- **Command:**
 - Ungeeignet. Es könnte sich vielleicht für Kontoauszüge eignen, aber dieses Feature ist nicht vorhanden.
- **Facade:**
 - Nicht nötig, die Klasse Geldautomat stellt diese Funktionalität quasi schon bereit
- **Adapter:**
 - Unerwünschte/unerwartete Schnittstellen/Methoden können in unserem Fall direkt verändert/angepasst werden, es bedarf keinem Adapter-Pattern.
- **Visitor:**
 - Ungeeignet. Beim Geldautomat gibt es keine Vererbung.
- **Strategy:**
 - Ungeeignet, es gibt in diesem Fall nicht mehrere Algorithmen die zur Laufzeit abhängig vom Anwendungsfall verwendet werden müssten.
- **State:**
 - Durch die Vorgabe der Zustände im gegebenen Zustandsdiagramm und um diese bestmöglichst einzuhalten, empfiehlt sich der Einsatz dieses Patterns. Dies beschränkt nur auf die Klasse Geldautomat.
 - Wird umgesetzt.
 - Umsetzung: Es musste eine abstrakte Klasse eingesetzt werden, da mehrere Attribute den States zur Verfügung stehen müssen. Die Methode info() wurde atomisiert, die Bestandteile sind in den verschiedenen info()-Methoden der States zu finden.

7. Implementiere die oben beschriebene Weiterentwicklung. Stelle sicher, dass die neue Funktionalität durch ausreichende Abdeckung mit lauffähigen Unit Tests abgesichert und dokumentiert ist → Neue Funktionalität, Testabdeckung und Lauffähigkeit der Tests werden bewertet

- In Zukunft soll der Benutzer nur noch drei Versuche haben, die korrekte PIN einzugeben. Nach drei erfolglosen Versuchen soll der Automat die Karte wieder ausgeben und der Vorgang damit abgebrochen werden. Am Ende soll der Automat

im Zustand "Bereit" sein. Der Benutzer soll über den Abbruch des Vorgangs informiert werden (Ausgabe einer Fehlermeldung und der Aufforderung, die Karte zu entnehmen). ✓

- Vierstellige PINs können führende Nullen enthalten, siehe Nr. 3 ✓

8. Selbstverständlich muss die Anwendung weiterhin lauffähig sein → Lauffähigkeit der Anwendung wird bewertet

✓

Angepasstes Zustandsdiagramm

