Aufgabe 1: Programmieren mit Smalltalk (10 Punkte = 3 + 3 + 4 Punkte)

a) Schreiben Sie einen Block, der nach dem Satz des Pythagoras für die zwei Katheten a und b die Hypotenuse c berechnet.

Lösung:

```
| pythagoras |
pythagoras := [ :a :b | ((a * a) + (b * b)) sqrt].

pythagoras
value: 10 value: 10
```

b) Beschreiben Sie die Vorgänge bei der Auswertung des folgenden Ausdrucks: Wer referenziert was, wer sendet Nachrichten an wen, wer reagiert wie?

```
| block |
block:= [3].
Dialog warn: block value printString.
```

Lösung:

Die Variable `block` referenziert einen Block.
Diesem wird die Nachricht `value` gesendet, und er antwortet mit einem Objekt, in diesem Beispiel mit der Zahl 3.
Diesem Objekt wird die Nachricht `printString` gesendet, worauf es eine texuelle Repräsentation seiner selbst liefert (einen String, in diesem Beispiel '3').
Und der wiederum wird der Nachricht `warn:` als Parameter mitgegeben.

c) Schreiben Sie für die Klasse `Integer` eine rekursive Methode namens factorial, die für ein Integer-Objekt n $n!$ berechnet.
Hinweis: $1! = 1, \quad n! = n(n-1)!$

Lösung:

```
factorial
   self > 1
      ifTrue: [^ (self - 1) factorial * self].
 ^ 1
```

Aufgabe 2: Klassen und Metaklassen (10 Punkte = 4 + 4 + 2 Punkte)

a) Beantworten Sie die folgenden Fragen zum Thema Klassen und Metaklassen für die Programmiersprache Smalltalk.

1. Wo sind Klassenvariablen definiert?
2. Wo sind Instanzvariablen definiert?
3. Auf was kann die Metaklasse (als Objekt) zugreifen?
4. Wieviele Instanzen gibt es von einer Metaklasse?

Lösung:

1. Klassenvariablen sind in der Metaklasse definiert.
2. Instanzvariablen sind in einer Klasse definiert.
3. Die Metaklasse kann auf Klassenvariablen und Klassenmethoden zugreifen.
4. Von der Metaklasse gibt es nur eine Instanz.

b) Um was für eine Klasse handelt es sich bei der unten auszugweise angegebenen Klasse `Collection`? Begründen Sie Ihre Antwort!

Klasse	Collection
benannte Instanzvariablen	
indizierte Instanzvariablen	nein
Instanzmethoden	

```
1    add: anObject
2         "Answer anObject. Add anObject
3         to the receiver collection."
4       ^ self implementedBySubclass

5    addAll: aCollection
6         "Answer aCollection. Add each element of
7          aCollection to the elements of the receiver."
8       aCollection do: [ :element | self add: element].
9       ^ aCollection
```

Lösung:

`Collection` ist ein typisches Beispiel für eine abstrakte Klasse in Smalltalk. Dies erkennt man

a) am Fehlen von Instanzvariablen

b) an der Implementation der Methode `add:`: Hier wird, anstatt etwas Entsprechendes zu tun, die Methode `implementedBySubclass` aufgerufen, die eine Fehlermeldung ausgibt.

c) Was ist

```
Metaclass class class ?
```

Lösung:

```
Metaclass class class == Metaclass
```

Aufgabe 3: Spezialisierung/Generalisierung, Vererbung (10 P = 8+2 P)

a) Die Programmiererinnen Jenny und Daniela möchten innerhalb eines Java-Programms geometrische Objekte modellieren. Für diese Objekte will man Fläche und Umfang berechnen. Unter anderem sollen eine Klasse `Quadrat` sowie eine Klasse `Rechteck` implementiert werden.
Jenny ist ein großer Freund von Vererbung und ist der Meinung, dass `Rechteck` Subtyp von `Quadrat` sein solle.
Daniela ist begeistert von Subtyping und meint, dass `Rechteck` Supertyp von `Quadrat` sein solle.
Erklären Sie, wie die unterschiedlichen Meinungen zustande kommen, indem Sie die Vorteile, Nachteile, Probleme von Vererbung und Generalisierung/Spezialisierung bedenken.

Lösung:

`Rechteck` Subtyp von `Quadrat`:
Jenny hat mit ihrer Argumentation insofern recht, als dass eine Klasse `Quadrat` Zustand und Verhalten implementiert, welches in `Rechteck` geerbt werden kann, so zum Beispiel die Breite.

```
class Quadrat {
    protected int breite;
}

class Rechteck extends Quadrat{
    protected int laenge;
}
```

Da in Java Vererbung und Subtypbeziehung miteinander einhergehen, wird somit `Quadrat` Supertyp von `Rechteck`.

`Quadrat` Subtyp von `Rechteck`:
Danielas Argumentation kann ebenso einleuchten: Wenn eine Methode (z.B. zum Zeichnen von geometrischen Objekten) ein Rechteck als Parameter erwartet, sollte diese Methode auch ein Quadrat als Parameter akzeptieren, schließlich ist vom mathematischen Standpunkt her jedes Quadrat auch ein Rechteck. Auch kann an jeder Stelle, an der ein Rechteck erwartet wird, schon ein Quadrat verwendet werden, indem man ein Rechteck mit gleicher Länge und Breite wählt.

Desweiteren könnte sie argumentieren, dass Jennys Lösung das Prinzip der Substituierbarkeit verletzt:
Es kann zu Problemen kommen, wenn ein Quadrat als Methodenparameter erwartet und ein Rechteck übergeben wird, denn ein Rechteck ist kein Quadrat.

b) Welche Konsequenzen hätte es, wenn man eine Methode, die eine Subklasse von der Superklasse geerbt hat, in der Subklasse löschen würde?

Lösung:

Das Prinzip der Substituierbarkeit wäre verletzt, denn die Subklasse könnte wegen der fehlenden Methode die Superklasse nicht mehr ersetzen.

Aufgabe 4: dynamisches Binden und Typprüfung (10 P = 4 + 3 + 3 P)

a) Es gelte:

```
Karpfen ist Subklasse von Fisch,
Fisch ist Subklasse von Tier,
Tier ist Subklasse von Object.
```

Wir haben folgende Deklaration/Initialisierung in Java:

```
Fisch f = new Karpfen();
```

Und ein in einer Variablen o referenziertes Objekt habe diese Methoden:

```
void m(Fisch a) { ... }
void m(Karpfen a) { ... }
```

An welche Methode bindet der folgende Aufruf?

```
o.m(f);
```

Begründen Sie Ihre Antwort.

Lösung:

Ein solcher Aufruf würde an die erste Methode binden, aber nicht an die zweite.
Begründung:
Denn für diese Frage zählt der Deklarationstyp von f und der ist Fisch.
Es handelt sich hier um einen Fall von Überladung. Der dynamische Typ des übergebenen Parameters - Karpfen - spielt dabei in Java zur Compilezeit keine Rolle.

b) Wie erreicht man dynamisches Binden in C++?

Lösung:

Obwohl die Zuweisungskompatibilität in C++ wie in JAVA über die Typkonformität an die Typerweiterung gebunden ist und somit einer Variable eines Typs auch Objekte seiner Subtypen zugewiesen werden können, werden in C++ Methoden zunächst einmal statisch gebunden.
Das bedeutet im Klartext, dass auf einem Objekt immer die Methode aufgerufen wird, die in der Klasse definiert ist, deren Typ die Variable (und nicht das Objekt, auf das sie verweist) hat.
Der tatsächliche Typ eines Objekts wird also ignoriert, es sei denn, die betreffende Methode wurde mit `virtual` (oder `dynamic`) deklariert.
Bei virtuellen Methoden wird hingegen wie in JAVA zur Laufzeit geprüft, welchen Typs das Objekt ist, und dann zur entsprechenden Methodenimplementierung verzweigt. Zu

diesem Zweck hält das Laufzeitsystem eine sog. Virtual function table, in der die zum Objekt passende Implementierung nachgeschlagen werden kann.

c) 1. Was können Sie über die dynamische Typprüfung in C++ sagen?

Lösung:

Die Beschreibung der dynamischen Typprüfung in C++ fällt knapp aus: Es gibt keine.

2. Wozu ist in C++ die Information *Runtime Type Information* (RTTI) gut?

Lösung:

Bibliotheksfunktion, die es erlaubt, für Objekte mit dynamisch gebundenen Methoden herauszufinden, Instanzen welcher Klassen sie sind.

Aufgabe 5: Typen und Klassen (10 Punkte = 2 + 4 + 4 Punkte)

a) Was tut der Typ eines Programmelements?

Lösung:

Er schränkt die Menge der Objekte, für die ein Programmelement stehen kann, und die Menge von Nachrichten, die an dieses geschickt werden können, ein.

b) Typen und Klassen dienen unterschiedlichen Zwecken:

- Welchem Zweck dienen Klassen?
- Welchem Zweck dienen Typen?

Lösung:

- Klassen dienen der Angabe von Implementierungen und damit als Container von ausführbarem Code.
- Typen dienen der Formulierung von Invarianten, die für Variablenbelegungen gelten müssen und deren Verletzung auf einen (logischen oder semantischen) Programmierfehler hinweist.

c) Typen und Klassen spielen unterschiedliche Rollen zur Laufzeit eines Programms:

- Welche Rolle spielen Klassen?
- Welche Rolle spielen Typen?

Lösung:

- Klasseninformation beeinflußt die Ausführung des laufenden Programms insofern, als sie Grundlage des dynamischen Bindens ist und in einem Programm als Eigenschaft von Objekten abgefragt werden kann.
- Typinformation beeinflußt die Ausführung eines laufenden Programms insofern, als sie ein Programm bei Verletzung einer Invariante abbrechen läßt (durch einen dynamischen Typtest) und damit einem anderen, schwieriger zuordnenbaren Fehler zuvorkommt.

Aufgabe 6: Typsystem (10 Punkte = 5 + 5 Punkte)

a) Wofür benötigt man ein Typsystem? Nennen Sie bitte die im Kurstext genannten fünf Gründe!

Lösung:

1. Typisierung regelt das Speicher-Layout.
2. Typisierung erlaubt die effizientere Ausführung eines Programms.
3. Typisierung erhöht die Lesbarkeit eines Programms.
4. Typisierung ermöglicht das automatische Finden von logischen Fehlern in einem Programm.
5. Die Verwendung eines der heute üblichen Typsysteme ermöglicht Modularisierung von Programmen, nämlich wenn ein Typ zugleich eine Schnittstelle oder ein Interface ausdrückt.

---

b) Gegeben sei Java-Programmcode ohne parametrische Polymorphie.
Die statische Typprüfung funktioniert bei solchem Code ohne parametrische Polymorphie sicher, mit zwei Ausnahmen:

1. Um welche Ausnahmen handelt es sich?

   Lösung:

   - **ArrayStoreException:**
     bei Arrays, die über eine Referenz angesprochen werden, die einen anderen Deklarationstyp hat, als der, mit dem der Array erzeugt wurde.
   - **ClassCastException:**
     bei expliziten Casts, weil diese die statische Typprüfung aushebeln.

2. Wie reagiert Java in diesen Fällen?

   Lösung:

   Beide Fälle werden zur Laufzeit erkannt:
   Die **ArrayStoreException**, wenn versucht wird, dem referenzierten Array ein Element zuzuweisen, das durch seinen tatsächlichen Elementtyp nicht gedeckt ist, die **ClassCastException**, wenn an der Stelle, wo gecastet werden soll, das referenzierte Objekt einen Typ hat, der nicht Subtyp des Typs ist, auf den gecastet werden soll.

Aufgabe 7: Problem der Substituierbarkeit (10 Punkte = 6 + 4 Punkte)

Subtype Requirement:
Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.
Eine Subtypenrelation zwischen S und T, die das obige Subtype requirement erfüllt, definieren Liskov und Wing durch eine Reihe von Bedingungen.

a) Nennen Sie diese Bedingungen.

Lösung:

1. Überschreibende Methoden in S erhalten das Verhalten der überschriebenen Methoden in T. Dazu gehört:
   - Kontravarianz der Argumenttypen der überschreibenden Methode
   - Kovarianz des Ergebnistyps der überschreibenden Methode
   - Kovarianz der Ausnahmen der überschreibenden Methode (die Typen der deklarierten Exceptions sind entweder Subtypen von den Typen der Exceptions der überschriebenen Methode oder die Exceptions werden gar nicht geworfen;)
   - Vorbedingungen der überschriebenen Methode implizieren Vorbedingungen der überschreibenden:

     $$pre_m^T(self : S) \Rightarrow pre_m^S(self : S)$$

   - Nachbedingungen der überschriebenen Methode werden von Nachbedingungen der überschreibenden Methode impliziert, also gilt hier:

     $$post_m^S(self : S) \Rightarrow post_m^T(self : S)$$

2. Die Invarianten von S implizieren die von T.

b) Welches Problem bleibt trotzdem noch bestehen, obwohl diese Bedingungen erfüllt sind? Beschreiben Sie das Problem.

Lösung:

Eine methodenweise Betrachtung von Bedingungen für die Substituierbarkeit reicht wegen des Aliasing und der damit verbundenen Möglichkeit des zusätzlichen Methodenaufrufs nicht aus:
Aufgrund des in der objektorientierten Programmierung weit verbreiteten Aliasing kann ein Objekt vom Typ S, das von einem Klienten wie ein Objekt vom Typ T betrachtet wird, von einem weiteren Klienten wie ein Objekt vom Typ S (oder wie von einem anderen Supertypen als T) betrachtet werden. Dadurch können dann auch Methoden auf dem Objekt aufgerufen werden, die Zustandsänderungen des Objekts verursachen, die nicht durch die mit T verbundenen Methodenspezifikationen (deren Vor- und Nachbedingungen) abgedeckt sind, ja die ein Verhalten bewirken, das mit dem von T nicht kompatibel und das fur Benutzerinnen des Objekts, die es als ein T ansehen, nicht akzeptabel ist.

Aufgabe 8: Parametrischer Polymorphismus u. Java (10 P = 3 + 4 + 3 P)

a) Nennen Sie einen Standardanwendungsfall für den einfachen parametrischen Polymorphismus und beschreiben Sie diesen Anwendungsfall kurz.

Lösung:

Collections sind ein Standardanwendungsfall. In der Collection sollen Objekte eines bestimmten Typs gespeichert werden. Oft möchte man sich aber nicht auf einen bestimmten Typ für diese Objekte festlegen, sondern die Collection, z.B. ein Dictionary, allgemein definieren und erst später angeben, für welche Typen von Objekten sie verwendet werden soll.

b) Gegeben ist der folgende Programmausschnitt in Java:

```
class Tier{}
class  Loewe extends Tier {}
...
Tier[] tiere;
Loewe[] loewen;
...
```

Es werden die folgenden beiden Zuweisungen gemacht:

```
tiere = loewen;
tiere[1] = new Tier();
Loewe loewe = loewen[1];
```

Sind diese Zuweisungen in Java erlaubt? Geben Sie eine kurze Begründung an.

Lösung:

Die Zuweisung `tiere = loewen` kann man in Java durchführen.
Die anschließende Zuweisung `tiere[1] = new Tier()` führt dann in Java allerdings prompt zu einem Laufzeitfehler (eine sog. Array store exception),
denn:
`tiere` ist ja lediglich ein Alias auf ein Array mit loewen, so daß die Zuweisung ein Tier anstelle eines Löwen an Arrayposition 1 setzt und das Array `loewen`, das ja per Deklaration nur Löwen zu enthalten verspricht, damit nicht mehr typkorrekt ist.
Würde man also diese Zuweisung `tiere[1] = new Tier()` zulassen, dann würde in der Folge die scheinbar korrekte Zuweisung `Loewe loewe = loewen[1]`, bei der `loewe` ein Tier zugewiesen wird, die Typinvariante von `loewe` verletzen.

c) Ist der folgende schreibende Zugriff auf die Liste in Java erlaubt? Begründen Sie kurz Ihre Antwort.

```
List<? extends Integer> liste = new ArrayList<Integer>();
liste.add(new Integer(1));
```

Lösung:

Bei der obigen Liste handelt es sich um eine Liste mit Wildcards, die eine obere Schranke für den Typparameter enthält. Bei solchen Listen ist nur lesender Zugriff erlaubt, schreibender hingegen verboten.

Begründung:
Für die Frage, ob der Compiler den schreibenden Zugriff in der zweiten Zeile erlaubt, ist ausschließlich der Deklarationstyp der Variablen 'liste' relevant. Dies ist der Typ

```
List<? extends Integer>
```

Dieser Typ umfasst alle Listen, denen bei der Instanziierung ein Typparameter mitgegeben wurde, der Subtyp von Integer ist (was übrigens den Typ Integer einschließt). Dazu gehören insbesondere auch solche Listen, bei denen der Parametertyp ein (fiktiver) Subtyp von Integer ist. In eine solchen Liste darf dann aber kein Integer-Exemplar 'gefüllt' werden.
Der Compiler kann daher keinen schreibenden Zugriff erlauben.

Aufgabe 9: Parametrischer Polymorphismus und Friends (10 P = 6 + 4 P)

a) Erläutern Sie anhand des folgenden Java-Programmstücks, warum man den beschränkten parametrischen Polymorphismus benötigt:

```
interface SortedList<E extends Comparable> {
   void insert(E element);
   void remove(E element);
   ...
}
```

Lösung:

Man kauft sich mit einfachem parametrischen Polymorphismus außerhalb der Typdefinition Typsicherheit zum Preis der mangelnden Typsicherheit innerhalb: Solange man keine Aussagen über den konkreten Typ, der für einen Typparameter eingesetzt wird, machen kann, kann man bei der Implementierung einer Klasse, die den parametrischen Typ definiert, auch keine Eigenschaften der Objekte, die von dem (unbekannten) Typ sein sollen, voraussetzen. Was man vielmehr braucht, ist beschränkter parametrischer Polymorphismus.
Die möglichen Werte der Typvariable E werden dadurch auf Typen eingeschränkt, die `Comparable` (direkt oder indirekt) erweitern. Die Implementierung der Methode zum Einfügen und Entfernen von Elementen in sortierten Listen kann also davon ausgehen, dass alle Objekte, die in einer solchen Liste gespeichert sind, die Nachricht `compareTo(.)` verstehen, die vom Interface `Comparable` vorgeschrieben wird.

b) Das Friends-Konzept in C++ hat einen direkten Bezug zur objektorientierten Programmierung. Erläutern Sie dieses Konzept.
Was kann man damit erreichen?

Lösung:

In der Praxis kommt es häufig vor, daß ein bestimmtes Teilproblem nicht von einer Klasse allein, sondern nur durch das Zusammenspiel mehrerer Klassen gelöst werden kann. Während diese Klassen untereinander eng kooperieren müssen und deswegen (relativ) intime Kenntnis voneinander benötigen (will sagen, auf Elemente zugreifen können müssen, die anderen Klassen verborgen bleiben sollten), gilt das für andere Klassen nicht unbedingt. Die Schnittstelle solcher kooperierenden Klassen sollte also nicht absolut, sondern relativ zu anderen Klassen definierbar sein.

Aufgabe 10: EIFFEL (10 Punkte = 4 + 2 + 4 Punkte)

a) Warum kann man EIFFEL als eine Sprache für objektorientierte Analyse und Design ansehen?

Lösung:

Wegen der Integration von Zusicherungen:
Zusicherungen, als Verträge zwischen dienstanbietenden und dienstnehmenden Klassen interpretiert, erlauben, das `Was` einer Software zumindest teilweise unabhängig vom `Wie` zu spezifizieren.
Die im Kurstext behandelten Typsysteme erlauben zwar auch schon, Zusicherungen auszudrücken, aber die sind jeweils auf die möglichen Werte einer Variable bezogen und bleiben dabei sowohl voneinander als auch von der Zeit unabhängig.
EIFFEL erlaubt darüber hinaus, nahezu beliebige Bedingungen fur Variablen- und Rückgabewerte von Methoden auszudrücken, die sowohl auf andere Werte als auch auf den zeitlichen Verlauf (vorher/nachher) Bezug nehmen können.

b) Wo sind in EIFFEL Vor- und Nachbedingungen angesiedelt, wo Invarianten?

Lösung:

- Vor- und Nachbedingungen sind in Methoden angesiedelt.
- Invarianten sind in Klassen angesiedelt.

c) Was sind die prominentesten Eigenschaften des Typsystems von EIFFEL?

Lösung:

1. Mehrfachvererbung
2. beschränkter parametrischer Polymorphismus
3. das Unterdrücken von Instanzvariablen und Methoden in Subklassen (Löschen von Methoden)
4. kovariante Redefinition, unterstützt durch sog. verankerte Typen