FernUniversität in Hagen

FAKULTÄT FÜR
MATHEMATIK UND INFORMATIK

Klausur zum Kurs 1853 im Sommersemester 2012

Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Hinweise zur Bearbeitung

1. Prüfen Sie die Vollständigkeit Ihrer Unterlagen. Die Klausur umfasst auf insgesamt 22 Seiten :
   - 1 Deckblatt
   - Diese Hinweise zur Bearbeitung
   - 10 Aufgaben auf Seite 1-18
   - Zwei zusätzliche Seiten für weitere Lösungen

2. Füllen Sie jetzt bitte zuerst das Deckblatt aus:
   - Name, Vorname und Adresse,
   - Matrikelnummer, Geburtsdatum und Klausurort.

3. Schreiben Sie Ihre Lösungen mit Kugelschreiber oder Füllfederhalter (*kein Bleistift*) direkt in den bei den jeweiligen Aufgaben gegebenen Leerraum.
   Benutzen Sie auf keinem Fall die Rückseiten der Aufgabenblätter.
   Versuchen Sie, mit dem vorhandenen Platz auszukommen, sie dürfen auch stichwortartig antworten.
   Sollten Sie wider Erwarten nicht mit dem vorgegebenen Platz auskommen, benutzen Sie bitte die beiden an dieser Klausur anhängenden Leerseiten.
   Es werden nur Aufgaben gewertet, die sich auf dem offiziellen Klausurpapier befinden.
   Eigenes Papier ist nur für Ihre persönlichen Notizen erlaubt.

4. Kreuzen Sie die bearbeiteten Aufgaben auf dem Deckblatt an. Schreiben Sie unbedingt *auf jedes Blatt* Ihrer Klausur Ihren Namen und Ihre Matrikelnummer, auf die Zusatzblätter auch die Nummer der Aufgabe.

5. Geben Sie die gesamte Klausur ab. Lösen Sie die Blätter nicht voneinander.

6. Es sind *keine Hilfsmittel* zugelassen.

7. Lesen Sie vor der Bearbeitung einer Aufgabe den *gesamten* Aufgabentext sorgfältig durch.

8. Es sind maximal 100 Punkte erreichbar. Wenn Sie mindestens 40 Punkte erreichen, haben Sie die Klausur bestanden.

9. Sie erhalten die korrigierte Klausur zurück zusammen mit einer Bescheinigung für das Finanzamt und ggf. dem Übungsschein.

10. Legen Sie jetzt noch Ihren Studierendenausweis und einen amtlichen Lichtbildausweis bereit, dann kann die Arbeit beginnen. Viel Erfolg!

Aufgabe 1: Interfaces (1 + 1 + 1 + 1 + 1 + 5 Punkte)

a) Welche Art des Gebrauchs von Interfaces kommt bei Black-Box-Frameworks vor?

Antwort:

ermöglichende Interfaces (Server-Client-Interfaces, Server-Item-Interfaces)

b) Welche Art des Gebrauchs von Interfaces kommt bei der Interface Injection vor?

Antwort:

Server-Client-Interface, Server-Item-Interfaces

c) Welche Art des Gebrauchs von Interfaces kommt im Zusammenhang mit Factories vor?

Antwort:

Familieninterfaces

d) Welche Art des Gebrauchs von Interfaces sollte hauptsächlich in Bibliotheken vorkommen?

Antwort:

anbietende Interfaces (Familien-Interfaces, Client-Server-Interfaces)

e) Warum wird das idiosynkratische Interface in Sprachen wie Java und C# kaum verwendet?

Antwort:

In diesen Sprachen erfüllt das Klasseninterface im Wesentlichen den gleichen Zweck.

f) Gegeben ist das folgende Programmfragment. Um welche Art von Interface handelt es sich bei dem Interface `Connector` und bei dem Interface `Connectand`?
Geben Sie eine kurze Begründung an.

```
interface Connector {
    void connect(Connectand c);
}

interface Connectand {
    Address supplyAddress();
}

class Xxx implements Connector {
    ...
    public void connect(Connectand c){
        ...
        c.supplyAddress();
        ...
    }
}

class Yyy implements Connectand {
    ...
    public Address supplyAddress() {...}

    public static void main(String[] args) {
        ...
        Yyy y = new Yyy();
        Xxx x = new Xxx();
        x.connect(y);
        ...
    }
}
```

Antwort:

Es handelt sich um ein paarig auftretendes Client/Server-Server/Client-Interface.

Begründung:
In dieser Client/Server-Konstellation benötigt der Server `Xxx` fur die Ausführung seiner Dienste Information, die ihm nicht beim Aufruf übergeben wurde. In solchen Fällen wird der Server zur gegebenen Zeit beim Client `Yyy` rückfragen, um diese Information zu erhalten. Es ist dazu allerdings notwendig, dass der Server den Client kennt.

Aufgabe 2: Abstrakte Klassen (6 + 2.5 + 1.5 = 10 Punkte)

Abstrakte Klassen erlauben gegenüber Interfaces die Implementierung von nichtstatischen Feldern und Default-Methoden.

a.) Welche Vorteile hat das?

Antwort:

1. Eine gemeinsame Oberklasse kann bereits eine Basisfunktionalität zur Verfügung stellen, welche in allen erbenden Klassen gleich ist und ansonsten immer wieder neu implementiert werden müsste.
2. Die Auswirkungen nachträglicher Erweiterungen einer Schnittstelle können über Default-Implementierungen abgefedert werden.
3. Die Verwendung von abstrakten Klassen erlaubt die Definition von Konstruktoren, über welche die Mindestanforderungen bei der Erstellung von neuen Objektinstanzen gesteuert werden können. So kann etwa sichergestellt werden, dass ein neu erstelltes Objekt intern ausreichend initialisiert wird; Interfaces bieten keine derartige Möglichkeit.
4. Die Bereitstellung einer Default-Implementierung befreit die Nutzer einer Schnittstelle davon, zwingend eine eigene Implementierung dieser Methode bereitstellen zu müssen. Dadurch wird die Typkorrektheit des Programms erhalten.

b.) Welchen Nachteil kann das haben?

Antwort:

Die Bereitstellung einer Default-Implementierung befreit die Nutzer einer Schnittstelle davon, zwingend eine eigene Implementierung dieser Methode bereitstellen zu müssen. Auf der einen Seite wird dadurch zwar die Typkorrektheit des Programms erhalten, auf der anderen Seite besteht allerdings die Gefahr, dass eine derartige Änderung insbesondere bei der Verwendung von Bibliotheken übersehen wird. Unter Umständen entspricht die Default-Implementierung nämlich nicht dem Verhalten, welches für eine abgeleitete Klasse eigentlich erforderlich wäre; der Compiler hat aber keine Möglichkeit, eine vergessene Methodenimplementierung zu bemerken, wenn eine Default-Implementierung vorhanden ist.

c.) Könnte man in C# oder Java auf die Verwendung von Interfaces verzichten und stattdessen nur noch abstrakte Klassen verwenden, ohne sich in seinen Ausdrucksmöglichkeiten einzuschränken?

Antwort:

Der Verzicht auf Interfaces wäre in C# und Java ohne Verzicht auf Ausdrucksmöglichkeiten nicht möglich, da beide Sprachen keine Mehrfachvererbung erlauben, welche aber benötigt werden würde, um von mehreren abstrakten Klassen zu erben. Da eine Klasse aber mehrere Interfaces implementieren kann, ist ein Verzicht nicht möglich.

Aufgabe 3: Dependency Injection (2 + 3 + 2 + 3 Punkte)

a) Wozu benötigt man Dependency Injection in der Interface-basierten Programmierung?

Antwort:

Die konsequente Verwendung von Interfaces in Variablen und Methodendeklarationen erlaubt es, die Anzahl der Referenzierungen anderer Klassen und damit die Abhängigkeit von diesen (bzw. die damit verbundene Kopplung) zu verringern. Es bleibt jedoch die Abhängigkeit von einer Klasse, die durch den Aufruf des Konstruktors zum Zwecke der Erzeugung einer Instanz dieser Klasse entsteht. Diese läßt sich durch die sog. Dependency injection eliminieren.

b) Dependency Injection wird auch beim Testen verwendet. Beschreiben Sie den Zusammenhang zwischen Testen und Dependency Injection.

Antwort:

Idealerweise testet man beim Unit-Testen jede Klasse einzeln. Hängt die Funktionalität der zu testenden Klasse von anderen, mit der zu testenden kollaborierenden ab und kann deren Funktionalität selbst fehlerhaft oder nicht oder nur zeitweise verfügbar sein, ersetzt man deren Objekte, soweit sie zum Testen benötigt werden, durch sog. Mock-Objekte.
Wie kann man dem zu testenden Objekt die Mock-Objekte unterschieben?
Das funktioniert z.B mittels Dependency Injection:
Man könnte den Code so refaktorisieren, dass das zu testende Objekt die zu 'mockenden' Objekte nicht selbst erzeugt, sondern sie ihm von außen eingeflößt, oder injiziert, werden.

c) Wer führt die Injektion bei der Dependency injection durch? (Erklären Sie bitte auch den verwendeten Begriff.)

Antwort:

Der **Assembler**:
Der Assembler ist ein Programmstück (häufig eine eigenständige Klasse), das aufgerufen wird, um die Objekte (oder Komponenten), die miteinander kooperieren sollen, zu verdrahten. Der Assembler stellt also eine Konfiguration aus voneinander unabhängigen (in dem Sinne, dass sie keine expliziten Abhängigkeiten besitzen) Objekten her.

d) Was sind die Grenzen der Dependency Injection?

Antwort:

- Die Erzeugung der Abhängigkeit (also die Zuweisung des Objekts, zu dem die Abhängigkeit besteht, an eine Variable) ist von Bedingungen abhängig, deren

Erfüllung nur die abhängige Klasse selbst erkennen kann.

- Die Abhängigkeit wird nicht zu einem genau definierten, von außen feststellbaren Zeitpunkt (wie beispielsweise dem der Erzeugung des Client-Objekts) eingerichtet.
- Die Dependency injection ist unmöglich für temporäre Variablen und für durch formale Parameter hergestellte Abhängigkeiten.

Aufgabe 4: Design by contract (3 + 4,5 + 2,5 Punkte)

JAVA hat mit der Version 1.4 ein Schlüsselwort `assert` spendiert bekommen. Mit ihm können beliebige boolesche JAVA-Ausdrücke zu Zusicherungen (engl. assertions) gemacht werden.

a) Ein Problem dieser JAVA-Assertions zeigt das folgende Programmstück. Um welches Problem handelt es sich?

```
boolean bin_da() {
     assert bin_weg();
     return true;
}

boolean bin_weg() {
     assert bin_da();
     return true;
}
```

Antwort:

Es treten Seiteneffekte auf: Bei eingeschalteter Überprüfung der Zusicherungen terminiert das Programm mit einem Stapelüberlauf, obwohl (bei ausgeschalteter Überprüfung) die Zusicherungen erfüllt sind.

b) Welche weiteren Unzulänglichkeiten gibt es bei JAVA-`Asserts` zu beachten? Nennen Sie bitte drei davon.

Antwort:

1. Dem `Assert`-Statement sieht man nicht direkt an, ob man eine Vor -oder Nachbedingung formuliert.
2. Man hat keinen Zugriff auf das zurückgegebene Element, da ein `return`-Statement die Methodenausführung sofort beendet.
3. Der direkte Zugriff auf alte Werte einer Variablen ist nicht möglich.
4. Mangelnde Redundanz, da boole'sche JAVA-Ausdrücke zu Zusicherungen gemacht werden und damit keine von Java unabhängige Spezifikation möglich ist.

c) Substituierbarkeit ist nur dann gegeben, wenn das substituierende Objekt den Vertrag des substituierten einhält. Was muss also für die entsprechenden Vor- und Nachbedingungen gelten?

Antwort:

Das zu substituiernde Objekt darf nicht mehr an Voraussetzungen (Vorbedingungen) verlangen und nicht weniger an Leistung (Nachbedingungen) liefern.
Die Vorbedingungen der Methoden eines Subtyps müssen gleich oder schwächer sein, die Nachbedingungen gleich oder stärker.

Aufgabe 5: JUNIT (2 + 3 + 2 + 3 Punkte)

a) Warum ist der Name JUNIT eigentlich falsch?

Antwort:

Beim Unit-Testen wird per Definition immer genau eine Unit getestet. JUNIT tut jedoch nichts, dies zu erzwingen. JUNIT sollte eher JRegression heißen.

b) Ist JUNIT eine Bibliothek, ein Blackbox- oder ein Whitebox- Framework?
Begründen Sie Ihre Antwort.

Antwort:

Es liegt ein Framework vor. Dies erkennt man an der Umkehrung der Ausführungskontrolle: Die Anwendungsklassen (in diesem Fall die Testfälle) enthalten keine Ablaufsteuerung, die zu ihrer Ausführung führen - vielmehr wird die Methode `runBare()` vom Framework aufgerufen. Es ist dies ein Fall von Umkehrung der Ausfuhrungskontrolle.
Da zudem Vererbung zum Einsatz kommt, handelt es sich bei JUNIT um ein Whitebox-Framework.

c) Wann spricht man beim UNIT-Testen von einem *falsch negativem Ergebnis*?

Antwort:

Beide Fehler (der im Test und der in der getesteten Funktion) passieren unbemerkt den UNIT-Test.

d) Was versteht der Kurs 1853 unter einem Entwurfsmuster?

Antwort:

Unter einem Entwurfsmuster (engl. design pattern) versteht man eine Vorlage, anhand derer ein neues Exemplar gebaut werden kann. Dabei muß das neue Exemplar keine exakte Kopie des Musters sein - vielmehr weist das Muster bestimmte Freiheitsgrade auf, die die Programmiererin nutzen kann, um das Exemplar an ihre konkrete Problemstellung anzupassen.
Ein Entwurfsmuster beschreibt schemenhaft eine Lösung für ein Standardproblem.

A design pattern is a written document that describes a general solution to a design problem that recurs repeatedly in many projects.

Aufgabe 6: JUNIT und Entwurfsmuster (4.5 + 5.5 Punkte)

Im Design von JUNIT kommen sog. Entwurfsmuster zur Anwendung.

P1: Im JUNIT-Testframework können Testsuiten aus Tests und Testsuiten bestehen. Welches Entwurfsmuster kommt hier zur Anwendung?

Antwort:

COMPOSITE PATTERN

P2: Man muss dem Framework mitteilen, welche Methoden es sind, die die Testfälle repräsentieren. Jeder Testfall könnte in seiner eigenen Klasse implementiert und die Methode immer gleich, zum Beispiel test() genannt werden. Von welchem Pattern wäre das die Anwendung?

Antwort:

STRATEGY PATTERN

P3: In der Klasse TestResult benachrichtigt die Methode strtTest(test) die sog. Listener. Welches Entwurfsmuster findet in diesem Zusammenhang Anwendung?

Antwort:

OBSERVER PATTERN

P4: Die Methode runBare() der Klasse TestCase enthält das allgemeine Verfahren, nach dem getestet wird:

```
1 public abstract class TestCase ...

2     public void runBare() throws Throwable {
3         Throwable exception= null;
4         setUp();
5         try {
6             runTest();
7         } catch (Throwable running) {
8             exception= running;
9         }
10        finally {
11            try {
12                tearDown();
13            } catch (Throwable tearingDown) {
14                if (exception == null) exception= tearingDown;
15            }
16        }
```

```
17          if (exception != null) throw exception;
18      }
```

Beschreiben Sie anhand dieser Klasse das TEMPLATE METHOD PATTERN.

Antwort:

Das TEMPLATE METHOD PATTERN besteht im wesentlichen aus einer Methode, die fur eine bestimmte Funktionalität, den Zweck der Methode, ein Schema (engl. template) vorgibt. Die Template-Methode besteht dazu aus einer durch die Verwendung von Methodenaufrufen allgemein gehaltenen Ablaufbeschreibung. Die konkreten Handlungen müssen von Subklassen der Klasse, die die Template-Methode beherbergt, geliefert werden. Sie werden von der Template-Methode per offener Rekursion aufgerufen. In der Regel ist die Klasse, die die Template-Methode beherbergt, abstrakt und ihre Subklassen sind konkret.
Es gibt zwei Varianten von Methoden, die primitiven und die Hook-Methoden.
Die primitiven Methoden sind in der Klasse der Template-Methode gar nicht definiert; sie werden hier lediglich durch eine abstrakte Methodendeklaration vertreten.
Die Hook-Methoden geben ein Default-Verhalten vor, das von Subklassen uberschrieben oder erweitert werden kann.
Ein typisches Vorkommen des TEMPLATE METHOD Patterns liegt in Form der Methode `runBare()` vor. Die Methoden `setUp()`, `runTest()` und `tearDown()` sind Hook-Methoden, die in Subklassen von `TestCase` überschrieben werden können. Das Default-Verhalten von `setUp()` und `tearDown()` ist jeweils, nichts zu tun; das von `runTest()` kann in der Regel unverändert übernommen werden und wird nur selten überschrieben.
.

Aufgabe 7: Vorbedingungen fürs Refactoring (3 + 4.5 + 2.5 Punkte)

Betrachten Sie die folgenden (voneinander getrennten) Codebeispiele und geben Sie Gründe dafür an, warum das Refactoring **Replace Inheritance With Delegation** nicht anwendbar ist:

a.) **Codebeispiel 1:**

```
1    class A {
2       public void f() {}
3       public int g() { return i; }
4
5       protected int i = 0;
6    }
7
8    class B extends A {
9        public void f() {}
10       public int g() { return i + j; }
11
12       protected int j = 0;
13    }
14
15    class C {
16       public void f() {
17          A a = new B();
18          a.f();
19          a.g();
20       }
21    }
```

Antwort:

In Funktion f der Klasse C findet eine Zuweisung eines Objektes vom Subtyp B an eine Variable vom Supertyp A statt. Diese Zuweisung verhindert die Anwendung des Refactorings.
Zudem wird in Klasse B auf das Feld i von A zugegriffen, was ebenso für das Refactoring nicht erlaubt ist.

b.) **Codebeispiel 2:**

```
1    abstract class A {
2       public void f() {}
3       public abstract int g();
4
5       public int i = 0;
6
7    }
8
9    class B extends A {
10      public void f() {}
11      public int g() { return i; }
12    }
```

```
13
14    class C {
15       public static void f( A a ) {
16          a.f();
17          a.g();
18       }
19    }
20
21    class D {
22       public void f() {
23          B b = new B();
24          C.f( b );
25       }
26    }
```

Antwort:

Die Superklasse A ist abstrakt, was die Anwendung des Refactorings verhindert. Zudem wird eine Variable vom Typ B im Funktionsaufruf C.f(b) an eine Variable vom Typ A zugewiesen, was ebenso zur Anwendung des Refactorings nicht erlaubt ist. Schließlich wird in B auch noch auf das Feld i in A zugegriffen; auch wenn es öffentlich deklariert ist, würde es in B nach der Anwendung des Refactorings nicht mehr zur Verfügung stehen.

c.) **Codebeispiel 3:**

```
1     class A {
2        public void f() { g(); }
3        public void g() {}
4     }
5
6     class B extends A {
7        public void f() {}
8        public void g() {}
9     }
```

Antwort:

Die Subklasse B wird durch die Superklasse A in offener Rekursion aufgerufen.

Aufgabe 8: Refactoring durchführen (5 + 5 Punkte)

a.) Im folgenden Beispiel gibt es verschiedene Arten von Vögeln, die Kokosnüsse transportieren:

`European, African, Norweigian_Blue`

Wenden Sie das Refactoring **Bedingung durch Polymorphismus ersetzen (Replace Conditional with Polymorphism)** auf das folgende Beispiel an.

```
1 double getSpeed() {
2    switch (_type) {
3       case EUROPEAN:
4          return getBaseSpeed();
5       case AFRICAN:
6          return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
7       case NORWEIGIAN_BLUE:
8          return (_isNailed) ? 0 :getBaseSpeed(_voltage);
9    }
10    throw new RuntimeException("Should be unreachable");
11 }
```

Antwort:

```
1 abstract class Bird {
2    abstract double getSpeed();
3 }

4 class European extends Bird {
5    double getSpeed() {
6       return getBaseSpeed();
7    }
8 }

9 class African extends Bird {
10    double getSpeed() {
11       return getBaseSpeed() -getLoadFactor() * _numberOfCoconuts;
12    }
13 }

14 class NorweigianBlue extends Bird {
15    double getSpeed() {
16       return (_isNailed) ? 0 : getBaseSpeed(_voltage);
17    }
18 }
```

b.) Wenden Sie das Refactoring **Einführung eines Nullobjekzs (Introduce NULL Object)** auf das folgende Codefragment an. Das Codefragment zeigt ein Beispiel aus einem gedachten Kursbetreuungssystem:

```
1 Betreuerin betreuerin = kurs.betreuerin;
2 if (betreuerin == null)
3    antwort = "leider keine da, die eine Antwort wüßte";
4 else
5    antwort = betreuerin.getAntwort();
```

Antwort:

Das obige Codefragment ändert sich zu:

```
1 Betreuerin betreuerin = kurs.betreuerin;
2 antwort = betreuerin.getAntwort();
```

Hinzu kommt die folgende Klasse:

```
3 class NullBetreuerin {
4    String getAntwort() {
5       return "leider keine da, die eine Antwort wüßte";
6    }
7 }
```

Aufgabe 9: Metaprogrammierung (3 + 4 + 3 Punkte)

a) Was versteht der Kurs 1853 unter Metaprogrammierung?

b) Nennen Sie drei Beispiele für Metaprogramme aus unterschiedlichen Bereichen!

c) Beschreiben Sie ein *vollständig reflektives System.*

Lösung zu Aufgabe 9: Metaprogrammierung (15 Punkte)

a) **Metaprogrammierung:**
Von Metaprogrammierung spricht man, wenn ein Programm, das Metaprogramm, ein anderes (oder sich selbst) oder seine Ausführung beobachtet oder sogar manipuliert. Metaprogrammierung ist rekursiv anwendbar, was soviel heißt wie daß Metaprogramme selbst der Metaprogrammierung unterliegen können.

b) **Beispiele für Metaprogramme**:

1. Programmierwerkzeuge wie Compiler, Interpreter, Optimierer, Debugger
2. Werkzeuge zur Programmanalyse (Metrikwerkzeuge, Kontrollflußanalyse etc.)
3. Refactoring-Werkzeuge

c) **vollständig reflektives System:**
Ein vollständig reflektives System besitzt die Fähigkeit, seinen eigenen Aufbau und seine eigene Ausführung zu beobachten (die sog. Introspektion), seinen Ablauf zu beeinflussen (die sog. Interzeption oder Interzession) sowie ganz allgemein seine Bestandteile zu ändern, zu ergänzen oder auszutauschen (die sog. Modifikation). So kann ein vollständig reflektives System beispielsweise Methodenaufrufe abfangen und bei Bedarf deren Ausführung verhindern bzw. auf andere Methoden umleiten.

Aufgabe 10: Extreme Programming (10 Punkte)

Extreme Programming ist in hohem Maße von der Verfügbarkeit geeigneter Werkzeuge abhängig. Geben Sie vier Werkzeuge an und begründen Sie, warum sie für das Extreme Programming wichtig sind.

Antwort:

1. **Syntaxeditoren mit Auto-Vervollständigung und automatischer Codeformatierung:**
   Da es im Extreme Programming nur das Prinzip der gemeinsamen Verantwortung fur den Code gibt, also insbesondere niemand da ist, die allein bestimmte Standards wie z. B. die Benennung von Komponenten vorgeben kann, ist es nötig, das jede Programmiererin flexibel auf Umstrukturierungen und Namensänderungen reagieren kann.
   Ein Syntaxeditor mit automatischer Vervollständigung von Bezeichnern, wie er in modernen Entwicklungsumgebungen üblich ist, bietet dabei unschätzbare Hilfe. Um nicht ständig über andere als die gewohnten Formen der Codeformatierung zu stolpern, ist die automatische (Überprüfung der) Einhaltung von Codeformatierungsrichtlinien ebenfalls wünschenswert.

2. **Refactoring-Werkzeuge:**
   Wer planlos vorgeht, muss jederzeit zu Änderungen bereit sein. Zeiteinsparungen lassen sich in der Praxis durch den systematischen Einsatz von Refactoring- Werkzeugen erzielen.

3. **Versionskontrolle:**
   Nur wenn sichergestellt ist, dass jede jederzeit Zugriff auf die aktuellste Version des gesamten Programms hat, kann das Prinzip der gemeinsamen Verantwortung für den Code (das ja jeder erlaubt, jederzeit beliebige Änderungen durchzuführen) überhaupt funktionieren.
   Auch verlangen die kurzen Iterationszyklen, dass in regelmäßigen Abständen vollständig kompilierbare Versionen des Produkts vorliegen. Auch das lässt sich mit Hilfe einer Versionskontrolle vergleichsweise einfach bewerkstelligen.

4. **Builder:**
   Je nach eingesetzten Technologien sind auch Werkzeuge zur automatischen Erstellung einer ausführbaren Version des Produkts (sog. Builder) hilfreich. Dies ist insbesondere dann der Fall, wenn das Endprodukt aus mehreren Teilprodukten zusammengesetzt werden muss, die mit unterschiedlichen Werkzeugen oder in unterschiedlichen Umgebungen entwickelt werden.
   Da beim Extreme Programming regelmäßig und in kurzen Abständen vollständige Versionen des Produkts angefertigt werden müssen, ist ein immer wieder neues, manuelles Zusammenbauen nicht möglich.