# Testkonzept

Beim XP-Paradigma werden Analyse, Design, Implementierung und Test nicht als voneinander abgegrenzte Phasen aufgefasst, sondern als parallel ablaufende Aktivitäten.

| **Was** | **Rolle** | **Wer** |
|---|---|---|
| Implementation planen | Programmierer | Haschke, Orlamünder |
| BB Testfälle entwickeln | Programmierer und Tester | Haschke, Orlamünder<br>Meißner, Schumann |
| Testtreiber entwickeln | Tester | Meißner, Schumann |
| Prototyp planen | Programmierer | Haschke, Orlamünder |

Testgetriebenes Programmieren ist eine Just-in-time-Designtechnik, um auf Just-in-time-Anforderungen einzugehen. Dafür wird zu jedem Projekt bereits vor Beginn der Implementierungsarbeiten ein Satz von Testbeispielen angelegt.
Idealerweise wird jede funktionale Programmänderung zuvor durch das Schreiben eines weiteren Tests motiviert. Dieser Test wird so entworfen, dass er zunächst fehlschlägt, weil das Programm die gewünschte Funktionalität noch nicht besitzt. Erst anschließend wird der Code, der den Test zum Laufen bringt, geschrieben.
Auf diese Weise wird die gesamte Programmentwicklung inkrementell durch das unmittelbare Feedback konkreter Tests angetrieben.
Zuerst die Tests zu schreiben, ist eine Möglichkeit, um herauszufinden, was programmiert werden muss und was nicht, und wie sichergestellt werden kann, dass tatsächlich programmiert wird, was programmiert werden soll.

**Der Zyklus des Testens und Programmierens:**
1. entwerfen eines Tests, der zunächst fehlschlagen sollte
2. gerade soviel Code schreiben, dass der Test tatsächlich fehlschlägt
3. gerade soviel Code schreiben, dass tatsächlich alle Tests durchlaufen

Diesen Prozess wiederholen wir, solange uns weitere Tests einfallen, die unter Umständen fehlschlagen könnten, bis der Code schließlich seine durch die Tests spezifizierten Anforderungen erfüllt.
Man schreibt diese Tests, bevor man den Code schreibt der diese Tests erfüllen soll, damit man sicherstellt, dass der Code einfach zu testen ist.
Außerdem weis man so, wann man ungefähr fertig ist.
Tests werden nur für solchen Code geschrieben, der auch fehlschlagen könnte. Nur wo auch wirklich etwas schief gehen kann, ist der Aufwand es wert. Im Zweifelsfall gilt allerdings: lieber einen Test zuviel, als einen zuwenig.
Zum effektiven Testen müssen Testfälle isoliert voneinander ausführbar sein. Es werden deshalb keine Annahmen über die Reihenfolge, in der Testfälle ausgeführt werden, getroffen. Voneinander abhängige Tests werden stattdessen gemeinsam in einem Testfall ausgeführt.
Klassen können auch zusätzliche öffentliche Methoden (public) hinzugefügt werden, wenn sie dadurch leichter getestet werden können.

**Ein typischer Testfall besteht aus vier Schritten:**
1. Testobjekte erzeugen
2. Operationen mit den Testobjekten durchführen
3. Ergebnisse dieser Operationen mit den erwarteten Ergebnissen vergleichen
4. (ggf. Dateien schließen)

**Dabei werden die Tests wie folgt organisiert:**

- Testfälle befinden sich in dem gleichen Paket wie der zu testende Code, also existiert für jede zu testende Klasse eine eigene Testfall-Klasse. Als Namenskonvention wird Test<NameDerZuTestendenKlasse> gewählt
- Alle Testfälle in einem Paket werden mit der suite-Methode zusammengefasst, es existieren also die main, modell, view und die controler Testpakete. Hierdurch werden alle Methoden, die mit "test" beginnen, automatisch aufgerufen, wenn der Testfall ausgeführt wird. Der Testfall kann so im Projektfortschritt erweitert werden.
- Testpakete sollten möglichst nach jedem Kompiliervorgang ausgeführt werden.
- Diese wiederum werden alle in der Testsuite Testprojekt zusammen-gefasst und zumindest einmal täglich ausgeführt. Sobald ein neues Paket angelegt wird, wird die suite-Methode dieser Klasse um das neue Testpaket erweitert.

Da die Testfälle so rekursiv organisiert sind, können, je nach Bedarf, unterschiedliche Teile des Projektes getestet werden (Klassen, Pakete, Projekt). Dadurch wird jeweils ein einzelner Softwarebaustein überprüft, und zwar isoliert von anderen Softwarebausteinen. Die Isolierung hat dabei den Zweck, komponentenexterne Einflüsse beim Test auszuschließen.
Außerdem gilt für die Tests der gleiche Qualitätsanspruch wie für den übrigen Code: selbstdokumentierend, kein duplizierter Code und möglichst einfach. Testcode sollte ebenfalls genauso regelmäßig und sorgfältig refaktorisiert werden wie der übrige Code.

Durch Testen kann man nicht beweisen das ein Programm 100% funktioniert. Aber man kann überprüfen, ob ein bestimmter Programmablauf so funktioniert wie man es will.

**Beispiel für eine GeoTest Klasse**

```
import junit.framework.*;

public class TestGeo extends TestCase {

  public TestGeo(String name) {
    super(name);
  }

  public void testName1() {
    assertTrue(…());
  }

  public void testName2() {
    assertEquals(…());
  }
}
```