Versionsverwaltung: Probleme beim Mergen von Branches

Arndt Franzen (www.arndt-georg-franzen.de)

16. Juni 2022

Ein paar Worte vorab ...

Dies ist ein Zettel und kein detailliert ausgearbeitetes Dokument. Vieles ist im Telegrammstil verfasst und bedarf daher u.U. weiterer Erläuterungen. Es stellt lediglich einen Arbeitsstand dar.

1 Einleitung

Im Folgenden geht es um die Probleme, die beim Mergen von Branches einer Versionsverwaltung auftreten können. Für die Beispiele habe ich Git in der Version 2.28.0 verwendet.

2 Fehlerebenen

Beim Mergen kann es auf drei Ebenen zu Problemen kommen

- Text-Ebene
- Syntax-Ebene
- Semantik-Ebene

Der folgende Quelltext `Calculator.java` dient dabei stets als Ausgangspunkt. Er implementiert einen Rechner mit einer Methode für die Berechnung der Summe sowie einer Methode für die Ausgabe des Ergebnisses. Der Einfachheit halber wird der Konsument, die `main`-Methode, in der `Calculator`-Klasse implementiert. Die Variable `tmpNumber3` wird lediglich benötigt, um einen Fehler auf semantischer Ebene zu erzeugen (Kap. 2.3). Die Leerzeilen in den Zeilen 7 und 14 werden benötigt, um keine Merge-Konflikte (Fehler auf Text-Ebene, siehe Kap.2.1) zu erhalten. Diese Einschränkungen haben allerdings keinen Einfluss auf die Allgemeinheit der Aussagen; sie dienen nur der Einfachheit der Beispiele. Das Beispiel sollte sehr einfach gehalten sein und ist nicht der Praxis entnommen; man würde die Addition nicht mit der zusätzlichen Variable implementieren. Es geht schließlich nur um das Prinzip, verschiedene potentielle Fehlerfälle darzustellen.

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.add(7, 12);
7
8      }
9
10     private int result;
11
12     public void add(int aNumber1, int aNumber2) {
13         int tmpNumber3 = 0;
14
15         result = aNumber1 + aNumber2 + tmpNumber3;
16         printResult();
17     }
18
19     public void printResult() {
20         System.out.println(result);
21     }
22 }
```

Der Quelltext liegt eingecheckt auf Branch A vor.

Kompilieren und Ausführen des Programms liefert den Wert 19.

```
$javac -d ..\generated *.java

$java -cp ..\generated test.Calculator
19
```

Anschließend wird wie folgt vorgegangen:

- Erstellen eines Branches B von Branch A.
- Änderung der Klasse `Calculator` auf Branch A.
- Änderung der Klasse `Calculator` auf Branch B.
- Mergen der Änderungen von Branch B auf Branch A.

2.1 Fehler auf Text-Ebene

Dieser Fall tritt i.d.R. auf, wenn auf beiden Branches Änderungen an der gleichen Zeile vorgenommen werden. Als Beispiel wird die Methode `add(-)` auf beiden Branches umbenannt, da sie mehr macht als nur eine Addition. Sie gibt zusätzlich das Ergebnis auf der Konsole aus. Das soll im Methodennamen berücksichtigt werden. Auf Branch A wird sie in `printSum(-)` und auf Branch B in `addAndPrintResult(-)` umbenannt.

Änderung auf Branch A (Änderung der Zeilen 6 und 12):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.printSum(7, 12);
7
8      }
9
10     private int result;
11
12     public void printSum(int aNumber1, int aNumber2) {
13         int tmpNumber3 = 0;
14
15         result = aNumber1 + aNumber2 + tmpNumber3;
16         printResult();
17     }
18
19     public void printResult() {
20         System.out.println(result);
21     }
22 }
```

Änderung auf Branch B (Änderung der Zeilen 6 und 12):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.addAndPrintResult(7, 12);
7
8      }
9
10     private int result;
11
12     public void addAndPrintResult(int aNumber1, int aNumber2) {
```

```
13          int tmpNumber3 = 0;
14
15          result = aNumber1 + aNumber2 + tmpNumber3;
16          printResult();
17      }
18
19      public void printResult() {
20          System.out.println(result);
21      }
22  }
```

Beide Beispiele kompilieren und laufen; sie geben „19“ auf der Konsole aus.

Beim Mergen kommt es zu einem Konflikt:

```
Auto-merging Calculator.java
CONFLICT (content): Merge conflict in Calculator.java
Automatic merge failed; fix conflicts and then commit the result.
```

```
package test;

public class Calculator {
    public static void main(String[] args) {
        Calculator tmpCalculator = new Calculator();
<<<<<<< HEAD
        tmpCalculator.printSum(7, 12);
=======
        tmpCalculator.addAndPrintResult(7, 12);
>>>>>>> test-02

    }

    private int result;

<<<<<<< HEAD
    public void printSum(int aNumber1, int aNumber2) {
=======
    public void addAndPrintResult(int aNumber1, int aNumber2) {
>>>>>>> test-02
        int tmpNumber3 = 0;

        result = aNumber1 + aNumber2 + tmpNumber3;
        printResult();
    }

    public void printResult() {
        System.out.println(result);
    }
}
```

2.2 Fehler auf Syntax-Ebene

Ein Fehler auf Syntax-Ebene bedeutet, dass es zu keinen Merge-Konflikten gem. Kap. 2.1 kommt. Allerdings kompilert der gemergte Quellcode nicht mehr. Dies kann vorkommen, wenn auf einem Branch eine Variable oder Funktion umbenannt wird, während auf dem anderen eine weitere Verwendung der Variable oder Funktion hinzugefügt wird. Ein weiterer Fehlerfall ist, wenn die Signatur auf einem Branch geändert wird, während eine weitere Verwendung auf dem anderen Branch hinzugefügt wird.

Auf Branch A wird die Methode `add(-)` in `printSum(-)` umbenannt. Auf Branch B wird lediglich ein Aufruf der Methode `add(-)` (mit „altem Namen“) hinzugefügt.

Änderung auf Branch A (Änderung der Zeilen 6 und 12):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.printSum(7, 12);
7
8      }
9
10     private int result;
11
12     public void printSum(int aNumber1, int aNumber2) {
13         int tmpNumber3 = 0;
14
15         result = aNumber1 + aNumber2 + tmpNumber3;
16         printResult();
17     }
18
19     public void printResult() {
20         System.out.println(result);
21     }
22 }
```

Änderung auf Branch B (Hinzufügen der Zeile 8):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.add(7, 12);
7
8          tmpCalculator.add(21, 29);
9      }
10
11     private int result;
12
13     public void add(int aNumber1, int aNumber2) {
14         int tmpNumber3 = 0;
15
16         result = aNumber1 + aNumber2 + tmpNumber3;
17         printResult();
18     }
19
20     public void printResult() {
21         System.out.println(result);
22     }
23 }
```

Mergeergebnis:

```
package test;

public class Calculator {
    public static void main(String[] args) {
        Calculator tmpCalculator = new Calculator();
        tmpCalculator.printSum(7, 12);

        tmpCalculator.add(21, 29);
    }

    private int result;

    public void printSum(int aNumber1, int aNumber2) {
        int tmpNumber3 = 0;

        result = aNumber1 + aNumber2 + tmpNumber3;
        printResult();
```

```
    }

    public void printResult() {
        System.out.println(result);
    }
}
```

Beim Kompilieren kommt es zu einer Fehlermeldung:

```
$javac -d ..\generated *.java
Calculator.java:8: error: cannot find symbol
                tmpCalculator.add(21, 29);
                             ^
  symbol:   method add(int,int)
  location: variable tmpCalculator of type Calculator
1 error
```

2.3 Fehler auf Semantik-Ebene

Auf dieser Ebene liefert Git keine Merge-Konflikte, und auch die Syntax der gemergten Klasse ist fehlerfrei. Das Programm verhält sich allerdings fehlerhaft.

Während man in den ersten beiden Fällen durch Git bzw. den Compiler auf den Fehler hingewiesen wird, können Fehler auf der semantischen Ebene lediglich in Tests entdeckt werden.

Im Folgenden werden zwei Möglichkeiten von Fehlern auf Semantik-Ebene betrachtet:

- Änderung des Schnittstellenvertrags einer Methode
- Änderung der Implementierung (der Logik) einer Methode

2.3.1 Änderung des Schnittstellenvertrags

Die Methode `add(-)` erfüllt zwei Aufgaben. Zum einen addiert sie die beiden Argumente, zum anderen gibt sie das Ergebnis auf der Konsole aus. Da das nicht schön ist, werden das Rechnen und Schreiben getrennt. Daher wird auf Branch A der Aufruf der Methode `printSum(-)` vom Produzenten in den Konsumenten verlegt. Damit wird der SSV von „Die Methode addiert die beiden übergebenen Zahlen und gibt diese auf der Konsole aus“ in „Die Methode addiert die beiden übergebenen Zahlen“ geändert. Auf Branch B wird die Methode `add(-)` erneut aufgerufen, allerdings mit dem alten SSV.

Änderung auf Branch A (Entfernen der ursprünglichen Zeile 16 und Hinzufügen von Zeile 7):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.add(7, 12);
7          tmpCalculator.printResult();
8
9      }
10
11     private int result;
12
13     public void add(int aNumber1, int aNumber2) {
14         int tmpNumber3 = 0;
15
16         result = aNumber1 + aNumber2 + tmpNumber3;
17     }
18
```

```
19      public void printResult() {
20          System.out.println(result);
21      }
22  }
```

Ausgabe: 19

Änderung auf Branch B (Hinzufügen von Zeile 8):

```
1   package test;
2
3   public class Calculator {
4       public static void main(String[] args) {
5           Calculator tmpCalculator = new Calculator();
6           tmpCalculator.add(7, 12);
7
8           tmpCalculator.add(21, 29);
9       }
10
11      private int result;
12
13      public void add(int aNumber1, int aNumber2) {
14          int tmpNumber3 = 0;
15
16          result = aNumber1 + aNumber2 + tmpNumber3;
17          printResult();
18      }
19
20      public void printResult() {
21          System.out.println(result);
22      }
23  }
```

Ausgabe: 19 und 50

Mergeergebnis:

```
package test;

public class Calculator {
    public static void main(String[] args) {
        Calculator tmpCalculator = new Calculator();
        tmpCalculator.add(7, 12);
        tmpCalculator.printResult();

        tmpCalculator.add(21, 29);
    }

    private int result;

    public void add(int aNumber1, int aNumber2) {
        int tmpNumber3 = 0;

        result = aNumber1 + aNumber2 + tmpNumber3;
    }

    public void printResult() {
        System.out.println(result);
    }
}
```

Ausgabe: 19. Die „50“ von Branch B wird nicht mehr ausgegeben.

Das klingt jetzt nicht dramatisch, da es sich nur um die Ausgabe auf der Konsole handelt. Aber man betrachte die folgende Klasse aus einem Online-Shop

```
public class OrderRepository {
```

```
    private Order order

    public void createOrderFromCart(Cart aCart) {
        ...
    }

    public void storeInDatabase() {
        ...
    }
}
```

Also `OrderRepository`, `createOrderFromCart(-)` und `storeInDatabase()` statt `Calculator`, `add(-)` und `printResult()`. Der fehlenden Ausgabe einer Zahl auf der Konsole entspricht das Fehlen von Bestellung-Datensätzen in der Datenbank.

2.3.2 Änderung der Programmlogik

Es wird die Berechnung der Summe in der Methode `add(-)` auf beiden Branches unterschiedlich implementiert.

Änderung auf Branch A (Hinzufügen der Zeilen 14 und 15):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.add(7, 12);
7
8      }
9
10     private int result;
11
12     public void add(int aNumber1, int aNumber2) {
13         int tmpNumber3 = 0;
14         tmpNumber3 = aNumber1;
15         aNumber1 = 0;
16
17         result = aNumber1 + aNumber2 + tmpNumber3;
18         printResult();
19     }
20
21     public void printResult() {
22         System.out.println(result);
23     }
24 }
```

Ausgabe: 19

Änderung auf Branch B (Hinzufügen der Zeilen 15 und 16):

```
1  package test;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          Calculator tmpCalculator = new Calculator();
6          tmpCalculator.add(7, 12);
7
8      }
9
10     private int result;
11
12     public void add(int aNumber1, int aNumber2) {
13         int tmpNumber3 = 0;
14
```

```
15          tmpNumber3 = aNumber2;
16          aNumber2 = 0;
17          result = aNumber1 + aNumber2 + tmpNumber3;
18          printResult();
19      }
20
21      public void printResult() {
22          System.out.println(result);
23      }
24  }
```

Ausgabe: 19

Mergeergebnis:

```
package test;

public class Calculator {
    public static void main(String[] args) {
        Calculator tmpCalculator = new Calculator();
        tmpCalculator.add(7, 12);

    }

    private int result;

    public void add(int aNumber1, int aNumber2) {
        int tmpNumber3 = 0;
        tmpNumber3 = aNumber1;
        aNumber1 = 0;

        tmpNumber3 = aNumber2;
        aNumber2 = 0;
        result = aNumber1 + aNumber2 + tmpNumber3;
        printResult();
    }

    public void printResult() {
        System.out.println(result);
    }
}
```

Ausgabe: 12

2.3.3 Bemerkung

In den beiden vorliegenden Fällen sind die Problemstellen natürlich aufgrund der Kürze der Beispiele schnell identifiziert. Aber wenn sich die Änderungen über mehrere verschiedene Komponenten erstrecken, treten die Fehler erst im Zusammenspiel dieser in Erscheinung.

3 Zeitliche Betrachtung der Entwicklung von Features

Wenn man mit einem Branch arbeitet, ist der Ablauf für die Entwicklung eines „Features“ wie folgt (Git-Nomenklatur bzgl. Push und Pull):

1. Aktualisierung des eigenen Workspaces („Push“)
2. Entwicklung des Features
3. Test des Features

4. Commit und Push
5. Aktualisierung des eigenen Workspaces („Pull“)
6. Test des Features

Der Test des Features in Schritt 6 erfolgt, um zu prüfen, dass das Feature auch mit anderen, parallel eingecheckten Änderungen noch funktioniert. Schlägt der Test fehl, müssen die Commits zwischen den Zeitpunkten 1 und 4 auf ihren Einfluss auf das Feature untersucht werden. Durch einen automatisierten Tests wird sichergestellt, dass durch spätere Commits von anderen Entwicklern das Feature weiterhin funktioniert.

Bei der Arbeit auf verschiedenen Branches (hier A und B), ist der Ablauf i.d.R. wie folgt:

1. Erstellung eines Branches A vom Master-Branch
2. Erstellung eines Branches B vom Master-Branch
3. Entwicklung von mehreren Features auf den beiden Branches:
   (a) Aktualisierung des eigenen Workspaces („Pull“)
   (b) Entwicklung des Features
   (c) Test des Features
   (d) Commit und Push
   (e) Aktualisierung des eigenen Workspaces („Pull“)
   (f) Test des Features

   Diese „Schleife“ entspricht der Entwicklung eines einzelnen Features auf einem Branch (s.o.).
4. Merge von Branch A auf Master
5. Merge von Branch B auf Master
6. Test der unter Punkt 3 entwickelten Features

Wir gehen im Folgenden davon aus, dass wir auf Branch A entwickelt haben. Die automatisierten Tests in Schritt 3 stellen zunächst lediglich sicher, dass spätere Commits auf Branch A das Feature nicht negativ beeinflussen, wie es bereits bei der Arbeit auf nur einem Branch der Fall war.

Sind die Tests in Punkt 6 fehlerhaft, müssen nun sämtliche Commits zwischen den Zeitpunkten 1 und 5 auf ihren Einfluss auf das Feature untersucht werden. Das bedeutet, dass die gesamte Entwicklung, d.h. sämtliche Änderungen, auf Branch B zu betrachten ist.

Zusammenfassend ist zu sagen, dass bei der Arbeit auf einem Branch bei Fehlschlagen eines Feature-Tests ein deutlich kürzerer Zeitraum und deutlich weniger Commits (und damit Codeänderungen) auf ihren Einfluss auf das Feature untersucht werden müssen, als bei der Verwendung mehrerer Branches. In einem solchen kürzeren Zeitraum kann man sich auch besser an die eigenen Änderungen erinnern. Das gilt natürlich auch für die anderen Entwickler, die die Codestellen bearbeitet haben, die den Einfluss auf das Feature besitzen. Im Gegensatz dazu brechen die (automatisierten) Tests bei der Arbeit mit mehreren Branches erst nach dem Merge. Und dann sind sämtliche Änderungen auf dem gemergten Branch zu betrachten.

Natürlich kann es auch bei der Arbeit auf einem Branch zu Problemen kommen, wenn der das Feature beeinträchtigende Code erst nach dem letzten Test (Punkt 4) eingecheckt wird. Allerdings wird dies durch automatisierte Tests aufgedeckt, und der Zeitraum und die Anzahl der Commits und damit zu betrachtenden Codeänderungen sind stark eingeschränkt. Wenn bei jedem Commit die automatisierten Tests gestartet werden, handelt es sich sogar nur um einen einzigen Commit.

4 Konsequenzen

Aufgrund dieser Betrachtungen bevorzuge ich die Verwendung eines einzigen Branches. Features können mit Feature-Toggles entwickelt werden.

Wenn doch Branches verwendet werden, sollten sämtliche Änderungen manuell gemerged werden; automatische Merges müssen aufgrund von Fehlermöglichkeiten auf semantischer Ebene (Kap. 2.3) vermieden werden. Nach Fertigstellung eines Features auf einem Entwicklungsbranch, wird dieses direkt in den `master`-Branch gemerged und getestet. Auch sollten die Änderungen auf dem `master`-Branch häufig in den eigenen Branch geholt werden.

Hintergrund ist, dass man sich noch an die Änderungen erinnern und insbesondere die Auswirkung von fremden Änderungen auf die eigenen Änderungen beurteilen kann (siehe auch obige Betrachtung).

Wird erst kurz vor einem Release (oder einem anderen „Meilenstein“) also ggf. erst nach Wochen der Entwicklung gemerged, fallen i.d.R. sehr viele Mergestellen aufgrund der Vielfalt der umgesetzten Features an, so dass man die Auswirkungen der Änderungen von Feature 1 auf Feature 2 nicht (mehr) gut beurteilen kann. Zudem kann man sich oftmals auch an die Änderungen zu den einzelnen Features nicht mehr richtig erinnern.

Zusammenfassend also

- Automatisierte Tests schreiben
- Nur einen Branch verwenden und Feature-Toggles verwenden
- Wenn doch mehrere Branches: häufig und manuell mergen