Lösung 1: ObservableValueBase

Die Implementierung der Klasse `ObservableList<E>` als Erweiterung der Klasse `ObservableValueBase<List<E>>` ist unten aufgeführt. Die Methode `getValue` wird verwendet um zu prüfen, ob sich der Wert, der beobachtet wird, verändert hat oder nicht.

```
public class ObservableList<E> extends ObservableValueBase<List<E>> {
     private final List<E> list = new ArrayList<E>();
     private boolean valid = true;

     private void markInvalid() {
          if(valid) {
               valid = false;
               fireValueChangedEvent();
          }
     }

     public boolean add(E e) {
          boolean res = list.add(e);
          if(res) markInvalid();
          return res;
     }

     public boolean remove(Object o) {
          boolean res = list.remove(o);
          if(res) markInvalid();
          return res;
     }

     public E get(int index) {
          valid = true;
          return list.get(index);
     }

     public Object[] toArray() {
          valid = true;
          return list.toArray();
     }

     public <T> T[] toArray(T[] a) {
          valid = true;
          return list.toArray(a);
     }

     public int size() {
          valid = true;
          return list.size();
     }

     @Override
     public List<E> getValue() {
          valid = true;
          return new ArrayList<E>(list);
     }

     @Override
     public void addListener(InvalidationListener listener) {
          valid = true;
          super.addListener(listener);
```

```
    }

    @Override
    public void addListener(ChangeListener<? super List<E>> listener) {
        valid = true;
        super.addListener(listener);
    }
}

public class AdderObserver implements ChangeListener<List<Integer>> {

    @Override
    public void changed(ObservableValue<? extends List<Integer>> observable,
                        List<Integer> oldValue, List<Integer> newValue) {
        int sum = 0;
        for (int n : newValue.toArray(new Integer[newValue.size()])) {
            sum += n;
        }
        System.out.println("AdderObserver: new sum is " + sum);
    }
}

public class ConsoleObserver implements InvalidationListener {

    @Override
    public void invalidated(Observable source) {
        System.out.println("ConsoleObserver: List has changed");
    }
}
```

Bemerkungen:

- Die Methode `getValue()` muss jeweils neue Instanzen zurückgeben, denn der alte und der neue Wert werden mit `equals` verglichen. Nur wenn dieser Vergleich `false` zurückgibt wird eine Invalidation- bzw. Changed-Notifikation ausgelöst. Falls die Methode `getValue()` immer dieselbe Referenz zurück gibt, dann ergibt der Vergleich auf dem alten und neuen Wert immer `true`.
- Damit ein Invalidation-Listener bei einer Invalidierung nur einmal aufgerufen wird muss die Klasse sich merken, ob der beobachtete Wert noch gültig ist. Die Klasse wird beim Aufruf einer Mutationsoperation (`add` und `remove`) invalidiert und eine Leseoperation (`get`, `toArray` und `size`) validiert das Observable wieder. Auch das Registrieren eines Listeners hat eine Validierung zur Folge, damit ein neu registrierter Invalidation-Listener bei der nächsten Invalidierung auch benachrichtigt wird. Daher werden die beiden `addListener`-Methoden überschrieben.
- Es ist wichtig, dass in der Methode `markInvalid` das `valid`-Flag *vor* dem Aufruf der Methode `fireValueChangedEvent` auf `false` gesetzt wird, denn ein Listener kann ja den Wert des Observable abfragen was eine Validierung zur Folge hat, daher wäre ein Setzen des valid-Flags nach der Notifikation falsch.

Regel:

- Ein Observable kann als Erweiterung der Klasse `ObservableValueBase` realisiert werden. Bei der Methode `getValue` muss jedoch für jeden geänderten Wert ein neues Objekt zurückgegeben werden; am besten wird dabei ein unveränderbares Objekt (Immutable) zurückgegeben. Immutables werden wir im Kontext des Prototype-Patterns noch anschauen.
- Die Erweiterung der Klasse `ObservableValueBase` ist selber für das Nachführen des invalid-Zustandes verantwortlich, d.h. wenn eine bereits invalidierte Instanz erneut geändert wird, dann werden *keine* Notifikationen mehr verschickt.

Lösung 2: Observer & Ressourcen

Bei jedem Klick auf den "New"-Button wird ein neues Fenster erzeugt, das viel Speicher verbraucht. Zudem wird jedes neue Fenster im "Close All"-Button als ActionListener registriert. In der Methode `actionPerformed` wird dann die `dispose`-Methode aufgerufen, die das Fenster abräumen soll.

Werden immer neue Fenster angelegt, so läuft man bald in ein `java.lang.OutOfMemoryError`. Das war zu erwarten. Doch leider bringt der CloseAll-Button nicht die gewünschte Abhilfe: es ist nicht mehr möglich weitere Fenster zu öffnen, da immer noch zu wenig Speicher vorhanden ist.

Das Problem liegt darin, dass beim Schliessen zwar `dispose()` aufgerufen, aber das Fenster beim Close-All-Button nicht abmeldet wird. Dieser Button hält also immer noch eine Referenz auf das geschlossene Fenster und verhindert so, dass der Garbage Collector den Speicher frei gibt.

Die Lösung besteht also einfach darin, überall, wo das Fenster als Listener angemeldet wurde, es beim Schliessen auch wieder abzumelden.

```
class SimpleFrame extends JFrame implements ActionListener {
    byte[] buf = new byte[1024*1024];

    public void actionPerformed(ActionEvent evt){
        JButton closeAllButton = (JButton)evt.getSource();
        closeAllButton.removeActionListener(this);
        dispose();
    }

}
```

Das Problem bei dieser Lösung ist jedoch, dass solange die Fenster nur über die Close-Box geschlossen werden, die Listener nicht abgemeldet werden. Das Speicherproblem bleibt bestehen – bis wieder einmal der Close-All-Button gedrückt wird.

Um das Problem generell zu beheben, muss sichergestellt werden, dass beim Schliessen des Fensters die `dispose`-Methode aufgerufen wird, in dem im Konstruktor der Klasse `SimpleFrame` folgende Anweisung ausgeführt wird:

```
setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

In der Methode `dispose` kann dann der Listener (in unserem Fall also die `SimpleFrame`-Instanz) aus dem `closeAllButton` entfernt werden.

```
@Override
public void dispose() {
    closeAllButton.removeActionListener(this);
    super.dispose();
}
```

Regel:
Observer müssen sich beim Subject abmelden, bevor der Speicher der Observer-Objekte wieder freigegeben werden kann.

In JavaFX wird dieses Problem beim Binding gelöst indem die Listener über schwache Referenzen angebunden werden. Sobald diese nicht mehr über eine starke Referenz erreichbar sind werden sie aus dem Observable entfernt und freigegeben. Schwache Referenzen werden wir im Kontext des Prototype-Patterns kurz anschauen. Falls hingegen in einem Observable bzw. einem ObservableValue explizit ein InvalidationListener bzw. ein ChangeListener registriert wird so gilt dieselbe Regel, d.h. auch diese Listener müssen explizit mit der Methode removeListener entfernt werden bevor sie vom Garbage Collector rezikliert werden können.

Lösung 3: Nebeneffekte von Änderungen

Wenn sich ein Observer beim Observable abmeldet während er notifiziert wird, wird er aus der Liste der Observer entfernt. Das führt dazu, dass alle Iteratoren auf dieser Liste ungültig werden. Es ist also nicht mehr möglich, auch die Observer zu benachrichtigen, die weiter hinten in der Liste sind.

Dies kann behoben werden, indem man während der Notifikation über eine Kopie der Observer-Liste iteriert. Wird nun ein Observer abgemeldet, so verändert sich zwar die (originale) Observer-Liste, nicht aber die Kopie, über welche gerade iteriert wird.

```
    public void notifyObservers() {
        Observer[] copy;
        synchronized(this){ copy = observers.toArray(new Observer[observers.size()]);}
        for (Observer obs : copy) { obs.update(this); }
    }
```

Eine andere Lösung ist, die während einer Notifikation eingehenden Mutationen zu speichern und später auszuführen. Dazu wird eine Klasse `Mutation` definiert, in der hinzuzufügende bzw. entfernende Observer gespeichert werden zusammen mit der Art der Mutation. Am Ende der Notifikations-Runde werden dann die gemerkten Mutationsaufträge ausgeführt. Damit das auch mit allfälligen rekursiven Observer-Aufrufen (durch Observer, die selbst das Model ändern) funktioniert, verwendet man am besten eine Variable `level`, welche die aktuelle Rekursionstiefe angibt.

```
    private List<Mutation> pendingMutations = new LinkedList<Mutation>();
    private int level = 0;

    public void addObserver(Observer o) {
        if(level > 0) pendingMutations.add(new Mutation(o, true));
        else observers.add(o);
    }

    public void removeObserver(Observer o) {
        if(level > 0) pendingMutations.add(new Mutation(o, false));
        else observers.remove(o);
    }

    public void notifyObservers() {
        level++;
        for (Observer obs : observers) obs.update(this);
        level--;
        if(level == 0) {
            for(Mutation m : pendingMutations) {
                if(m.add) observers.add(m.observer);
                else observers.remove(m.observer);
            }
            pendingMutations.clear();
        }
    }

    private static class Mutation {
        private Observer observer;
        private boolean add;
        public Mutation(Observer obs, boolean a) { observer = obs; add = a }
    }
}
```

Alternativ kann auch während einer Notifikation (level > 0) in den Methoden add- und removeObserver die Liste der Observer kopiert werden. Das ist die Lösung die mit JavaFX implementiert ist.

Regel:
Durch An- oder Abmeldung von Observern sollte nie eine Liste verändert werden, über die gerade iteriert wird, um alle Observer aufzurufen.

Lösung 4: Zyklische Abhängigkeit

Der Zyklus von Observer-Aufrufen und Modell-Änderungen muss unterbrochen werden. Dazu gibt eine Reihe verschiedener Lösungs-Möglichkeiten zur Auswahl:

1) Änderung im `ColorModel`

```
public void setColor(Color color) {
  if (!color.equals(this.color)) {
    this.color = color;
    notifyObservers(color);
  }
}
```

2) Änderung in der Observable-Implementierung in der Klasse `RedScrollbar`:

```
@Override
public void update(Observable source, Object arg) {
  int newValue = ((Color)arg).getRed();
  if(getValue() != newValue) {
    setValue(newValue);
  }
}
```

3) Änderung in der Klasse `Scrollbar`

```
public void setValue(int value) {
  if(this.value != value) {
    this.value = value;
    this.notifyObservers(value);
  }
}
```

4) Änderung in der Scrollbar-Listener-Implementierung

```
boolean updating = false;
sb.addObserver(new Observer() {
  @Override
  public void update(Observable source, Object arg) {
    if(!updating) {
      updating = true;
      Color current = model.getColor();
      model.setColor(new Color((int)arg, current.getGreen(), current.getBlue()));
      updating = false;
    }
  }}
);
```

5) Generelle Lösung im Kontext von AWT/Swing:

   Bei AWT wird (im Gegensatz zu Swing) der in einem Scrollbar registrierte `AdjustmentListener` nur dann ausgelöst, wenn der Scrollbar mit der Maus oder der Tastatur verändert wird, nicht aber wenn die Methode `setValue` aufgerufen wird. Damit wird der Zyklus auch aufgebrochen.

   Analog ist es bei anderen Controls (z.B. `JCheckBoxMenuItem`) möglich, die Benutzerinteraktion mit einem `ActionListener` anstelle eines `ItemListeners` zu kontrollieren. Der vom Control bei der Änderung der Farbe verschickte `ItemEvent` wird dann nicht verwendet, und der Zyklus ist auch aufgebrochen.

Regel:
Es sollte nur dann eine Notifikation stattfinden, wenn sich am Modell wirklich etwas verändert hat!

Lösung 5: Was ist das Modell, wie wird es verändert?

Das Problem ist, dass zu viele Notifikationen verschickt werden. Wenn z.B. die Farbe über einen `RadioButton` eingestellt wird, erhält die Farbfläche *drei* update-Aufforderungen. Der Grund ist, dass alle drei Farbkanäle (nacheinander) gesetzt werden, und bei jedem Setzen ein Update erzeugt wird. Dabei werden Zwischenzustände auf dem Weg zum Ergebnis einer Operation sichtbar, die für sich gesehen gar keinen Sinn ergeben bzw. „aus logischer Sicht" gar nicht eingenommen werden.

Das Modell wurde zu fein-granular entworfen. Eigentlich wird hier das Geheimnisprinzip verletzt, denn auf logischer Ebene wollen wir *eine Farbe* setzen und nicht drei Farbkanäle verwalten. Wir gewähren also dem Klienten des Farbmodells einen Einblick, wie dieses intern aufgebaut ist (nämlich mit den drei Farben rot-grün-blau und nicht irgendwie anders CYMK oder HSV, etc.) und in welcher Reihenfolge diese Kanäle verändert werden. Der Beobachter (in diesem Falle die Farbfläche) sieht somit bei einer logischen Farbänderung (gelber `RadioButton`) alle 3 nötigen Vorgänge um das Farbmodell anzupassen und zieht u.U. falsche Schlüsse (im Beispiel „meint" der Listener, der auf ROT achtet, dass diese Farbe hätte eingestellt werden sollen). Dabei wären die Zwischenschritte zu verbergen und eine update-Meldung sollte erst dann verschickt werden, wenn das Modell seine *Zustandsänderung abgeschlossen* hat und wieder *konsistent* ist.

Die Notifikation von falschen Zwischenwerten kann behoben werden, indem in der Methode `setColor` nicht einfach die einzelnen Farbkanäle gesetzt (und notifiziert) werden, sondern zuerst die Farbkanäle geändert und danach die Notifikationen ausgeführt werden.

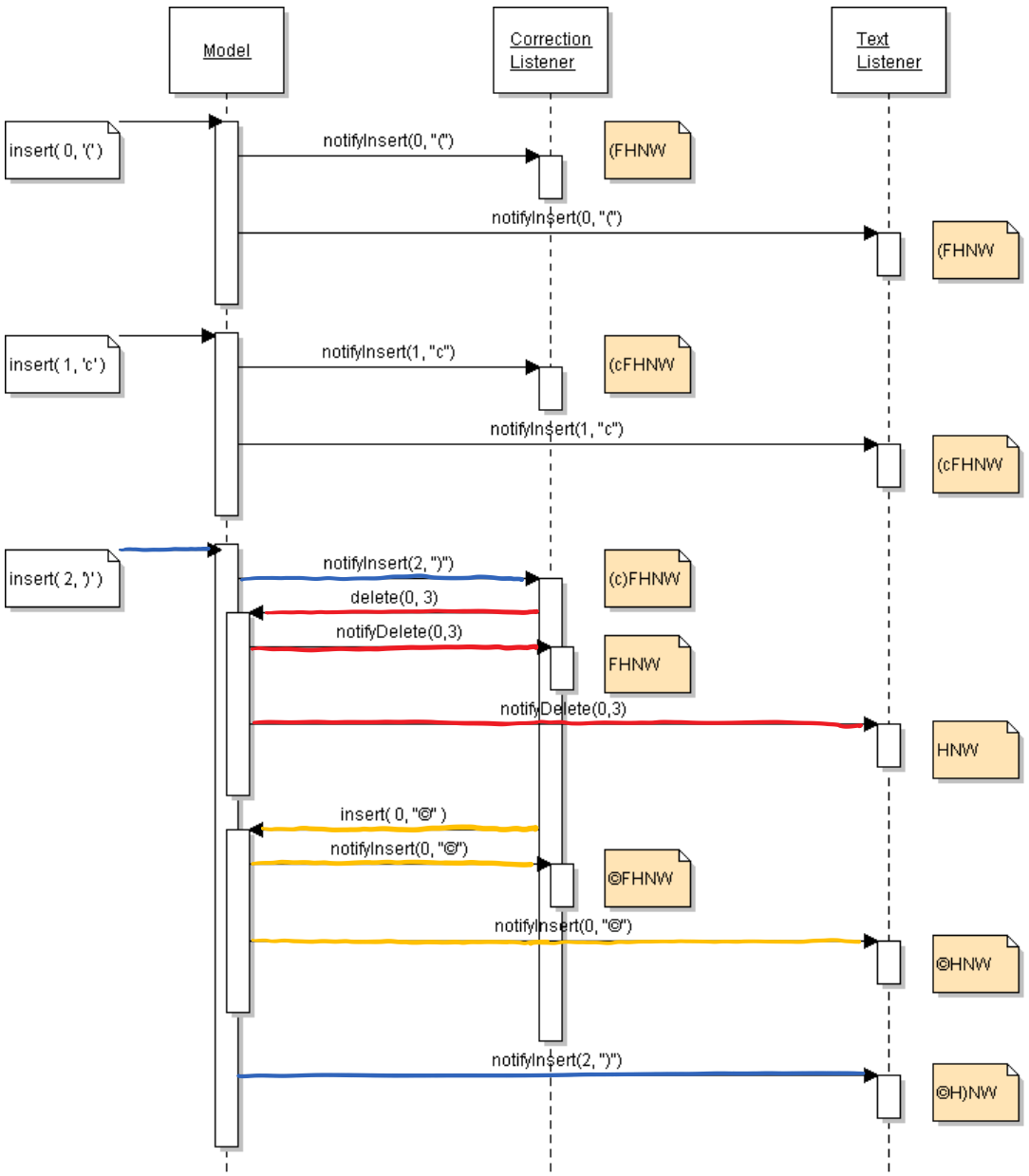
Die folgende Implementierung stellt dabei zusätzlich sicher, dass ein Farbkanal nicht mehrfach notifiziert wird.

```
public void setColor(Color c) {
   EnumSet<ColorChannel> s = EnumSet.noneOf(ColorChannel.class);
   if (red != c.getRed()) s.add(ColorChannel.RED);
   if (green != c.getGreen()) s.add(ColorChannel.GREEN);
   if (blue != c.getBlue()) s.add(ColorChannel.BLUE);
   red = c.getRed();
   green = c.getGreen();
   blue = c.getBlue();
   this.color = new Color(red, green, blue);
   notifyListeners(s);
}

public void setRed(int red) {
   this.red = red;
   this.color = new Color(red, green, blue);
   notifyListeners(EnumSet.of(ColorChannel.RED));
}

public void setGreen(int green) { /* analog */ }
public void setBlue(int blue) { /* analog */ }

private void notifyListeners(EnumSet<ColorChannel> channels) {
   for (ColorListener l : listeners.keySet()) {
      if (!Collections.disjoint(listeners.get(l), channels))
         l.colorValueChanged(color);
   }
}
```

Regeln:

- Zustandsänderungen sollten von aussen wenn immer möglich nur als ein Schritt beobachtbar sein.
- Der beobachtbare Zustand sollte jederzeit logisch konsistent sein.

Lösung 6: Kausalität von Änderungen



Hier überholen sich Events.

Der `CorrectionListener` wird vor dem `TextListener` aufgerufen. Somit korrigiert er den Text bevor der `TextListener` die Möglichkeit hatte, die Eingabe von ")" darzustellen. Erst nachdem im Modell die Ersetzung schon stattgefunden hat, wird der `TextListener` auch noch über die Eingabe von ")" informiert. Der `TextListener` zeigt somit etwas an, das so gar nicht mehr im Modell vorhanden ist.

Die Lösung besteht darin, während einer laufenden Notifikations-Runde ausgelöste Änderungen am Modell verzögert erst nach Abschluss aller Notifikationen auszuführen. Das Modell muss dazu Änderungsaufträge, die eingehen, während Observer notifiziert werden, in geeigneten Objekten speichern und später ausführen. Die dafür nötigen Warteschlangen-Mechanismen lassen sich auf verschiedene Arten implementieren. Mögliche Lösungen finden Sie auf dem Dateiserver als `copyright.zip`.

Regel:
Änderungen am Modell so lange verzögern, bis alle Observer über alle vorangegangenen Änderungen informiert worden sind.