AnalogClock-Tutorium

AnalogClock-Tutorium

Pascal Vollmer

Version 0.7.3.
02.08.2011

Dieses Tutorium wurde mit open source-Software erstellt. Für die Textverarbeitung wurde LyX verwendet, für Vektorgrafik Xfig, für Bildverarbeitung gimp. Ich bedanke mich bei den Entwicklern.

Für Hinweise zur Verbesserung dieses Tutoriums an diese Adresse bin ich dankbar.

1 AnalogClock-Tutorium

Es ist Zeit, nach unserem zweijährigen Smalltalk-Programmierkurs am Gymnasium ein etwas größeres Projekt anzugehen und das, was wir bisher gelernt haben, im Zusammenhang anzuwenden.

Wir wollen eine analoge Uhr programmieren - auf den ersten Blick nichts Besonderes aber auf den zweiten doch reich an interessanten Aufgaben.

- Eine Uhr ist ein Beispiel für ein grafisches Objekt mit Zustand (STATE) und Regeln für die Zustandsänderung (BEHAVIOR). Wir werden sehen, dass einige Zustandsvariablen sichtbare Bestandteile der Uhr repräsentieren. Es gibt aber auch andere, die kein sichtbares Gegenstück haben.

- Jeder weiß, wie man eine analoge Armbanduhr bedient. Die Grundfunktionen der Uhr werden leicht zu beschreiben sein. Auch wird es einfach sein, Tests auf korrektes Funktionieren aufzuschreiben. Umso schneller können wir mit weitergehenden Funktionen experimentieren.

- Die Analoguhr ist ein Vertreter einer ganzen Klasse von runden Zeigerinstrumenten, wie sie z.B. in Fahrzeugen aller Art vorkommen oder in Anzeigetafeln von Prozessleitständen.

Unsere Software findet sich unter AnalogClock und kann in ein Squeak- oder Pharo-Image geladen werden.

Mit der analogen Uhr haben wir ein Entwicklungsvorhaben ausgewählt, zu dem es innerhalb der Squeak-Welt einige vergleichbare Projekte gibt:

- [32], S. 42ff. zeigt, wie der Stiftroboter die Zeiger einer analogen Uhr zeichnen kann,

- für Etoys gibt es eine Analoguhr von der University of Illinois (Create an Analog Clock) ,

- in einigen Distributionen von Squeak und auf SqueakSource ist ein `WatchMorph` enthalten (WatchMorph),

- auf SqueakSource findet sich eine kreisförmiges Anzeigeeinheit (Gauge).
  Mit `GaugeSliderMorph new openInWorld` kann man sie "in die Welt setzen".

In den kommenden Abschnitten werden wir nach und nach Entscheidungen erläutern, die im Verlauf eines solchen Projekts zu treffen sind. Das schließt auch Fehlentscheidungen und Fehler ein, denen wir exemplarisch nachgehen werden, um schließlich zu erklären, wie wir sie korrigiert haben.

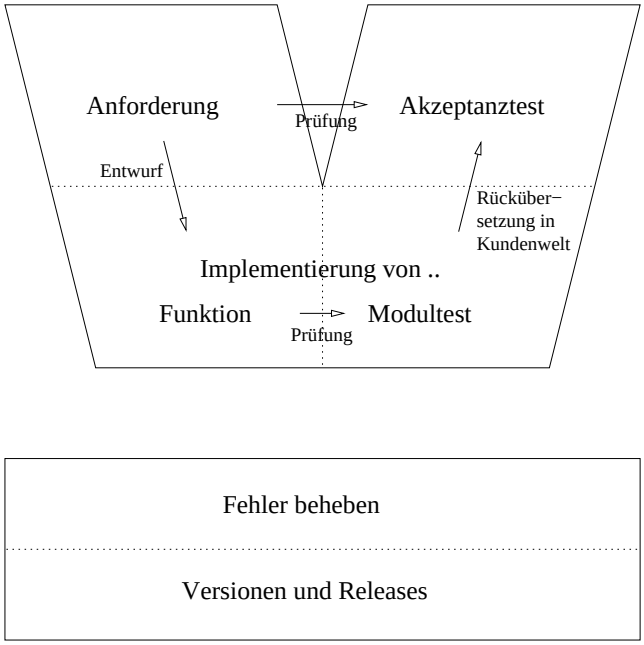

Abbildung 1.1: Arbeitsschritte

1.1 Ablauf der Entwicklung

Für unser weiteres Vorgehen betrachten wir eine Anordnung von vier aufeinander bezogenen Arbeitsschritten.

1. Wir beginnen bei den Anforderungen (REQUIREMENT). Alles, was wir von unserer Entwicklung fordern, wollen wir in möglichst klaren Worten aufschreiben.

2. Die Anforderungen werden in ein Programm übersetzt (PROGRAM, meist synonym: IMPLEMENTATION, APPLICATION). Wir wissen bereits, dass sich das Programm aus kleineren Einheiten (UNIT) zusammensetzt: aus Klassen.

3. Zumindest für die komplizierteren Klassen wollen wir prüfen, ob sie auch tatsächlich das tun, was wir erwarten. Dafür schreiben wir Modultests (UNIT TEST).

4. Schließlich interessiert uns für das Programm insgesamt, ob die aufgeschriebenen Anforderungen erfüllt werden. Besser noch, ob es alle Wünsche zufrieden stellt, die man vernünftiger Weise mit unserem Programm verbindet. Das wollen wir in einem Abnahmetest (ACCEPTANCE TEST) prüfen.

Wir betrachten nun, wie diese vier Arbeitsschritte aufeinander bezogen sind.

- Die Anforderungen werden mittels eines Objektentwurfs (DESIGN) in ein Programm übersetzt.

- Der Modultest prüft das Programm.
- Während die Modultests die Funktion einzelner Klassen nachweisen, weist der Abnahmetest die Funktion des ganzen Programms nach.
- Der Abnahmetest prüft die Umsetzung aller Anforderungen und weitergehend die Erfüllung aller Kundenwünsche.

Dazu treten noch zwei weitere unterstützende Arbeitsbereiche, die Fehlerbehebung und das Erstellen von Versionen und Releases. Sie seien hier nur genannt, wir werden uns später mehr mit ihnen beschäftigen.

Kein Koch der Welt kann mit seinen zwei Händen gleichzeitig in vier Töpfen rühren. Ganz ähnlich verhält es sich mit den hier genannten vier Arbeitsbereichen. Es ist ganz normal, dass ein Projekt etwa bei der Programmierung vorübergehend weiter ist als beim Modultest.

Aber was zeitweilig ok ist, muss nicht durchgängig gelten. Es ist sinnvoll, auf besondere Zeitpunkte hinzuarbeiten, an denen man nicht den Fortschritt sondern die Abstimmung aller Arbeitsbereiche in den Vordergrund stellt. Beim Kochen wäre das der Moment, an dem ein Gang eines Menüs serviert wird, z.B. die Vorspeise. Wer Spargel mit Sauce béarnaise angekündigt hat, sollte schon zusehen, dass beides fertig ist, wenn es an's Servieren geht.

Im Verlauf eines Softwareprojektes ist dieser Zeitpunkt die Software-Abgabe (Release). Ein Abgabestand unserer Software soll sich dadurch auszeichnen, dass die Arbeitsprodukte der vier Arbeitsbereiche

- vollständig sind,
- aufeinander abgestimmt sind und
- eine definierte, nachprüfbare Qualität haben.

Den hier skizzierten Ablauf der Entwicklung werden wir in den folgenden Abschnitten 1.2 bis 1.5 kommentieren. Die Projektdokumentation selbst (siehe 1.9) folgt dem gleichen Schema.

1.2 Anforderungen

Die grundlegenden Anforderungen an unsere Software lassen sich in wenigen Sätzen beschreiben.

Es ist eine analoge Uhr darzustellen mit kreisförmigem Ziffernblatt. Darauf sind zu sehen: Stunden-, Minuten- und Sekundenzeiger. Die drei Zeiger werden als geschlossene Polygonzüge dargestellt. Eine rautenförmige Standardform für die Zeiger ist im Programm vorgegeben. Der Benutzer kann aber auch eigene Zeigerformen für seine Uhr zeichnen - solange sie mit Polygonzügen darstellbar sind.

Der Sekundenzeiger springt im Sekundentakt, Minutenzeiger und Stundenzeiger werden im Minutentakt aktualisiert.

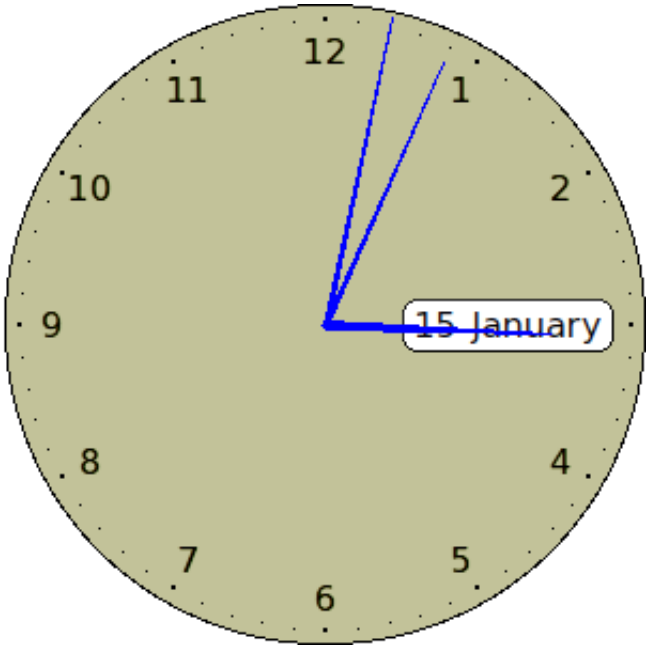

Abbildung 1.2: Eine Analoguhr. Es war eben noch vier nach drei. Vor zwei Sekunden.

Auf Wunsch des Benutzers kann eine Darstellung von aktuellen Terminen hinzugefügt werden. Dazu wird eine Spirale auf das Ziffernblatt gezeichnet. Die anstehenden Termine werden in die Spirale so eingefügt, dass die nächstliegenden außen zu sehen sind, weiter weg liegende innen.

Die Uhr kann in Lage und Größe verändert werden.

Zu dieser kleinen Liste kommen noch viele weitere Anforderungen hinzu. Der vollständige Anforderungskatalog findet sich im Projekthandbuch. Die Quellen des Projekthandbuchs finden sich im Paket `PVpubAnalogClock` im Klassenkommentar zu `AnalogClockProjectManual`.[1]

1.3 Implementierung der Funktion

In dem Übersetzungsschritt, der von den Anforderungen zum Programm führt, wird festgelegt, wie die Objekte zueinander in Beziehung treten. Die Beziehungen können vielfältiger Art sein.

- (logisch) Klassen treten in Beziehung als Diensterbringer und Dienstnutzer,
- (zeitlich) Klassen sorgen dafür, dass verschiedene Objekte in einer bestimmten zeitlichen Abfolge erzeugt oder zum Löschen freigegeben werden,
- (räumlich) Klassen mit sichtbaren Anteilen können relativ zueinander bewegt werden.

[1] Wie man das Projekthandbuch aus den Quellen erstellt, kann man in 1.9.4 nachlesen.

Wir wollen einige dieser Frage in 1.3.1 an einem bereits fertigen Programm untersuchen. Ab 1.3.2 steigen wir dann in unseren eigenen Objektentwurf ein.

1.3.1 Wie kommt ein Morph auf den Schirm?

Bevor wir mit uns in die Implementierung stürzen, lohnt es sich, anzuschauen, wie eine bereits vorhandene ähnliche Anwendung die grafischen Bestandteile einer Analoguhr auf den Bildschirm bringt und koordiniert. Die Anwendung heißt `WatchMorph.` Sie lässt sich von WatchMorph laden und wird im Klassen-Browser in die Kategorie `Morphic-Demo` eingeordnet.

1.3.1.1 Beispiel WatchMorph

Die Klasse `WatchMorph` stellt eine einfache analoge Uhr dar. Die Anwendung wird mit `WatchMorph new openInWorld` gestartet.

Wie werden die Elemente der Uhr dargestellt? Wenn wir uns ein wenig in den Methoden von `WatchMorph` umsehen, können wir die Methoden `WatchMorph>>step` und `WatchMorph>>drawOn:` finden. Diese beiden Methoden sind es, die der Uhr Leben einhauchen. Die erste ist eine Art Schrittmacher denn sie reagiert auf einen Zeittakt. In ihr kann die grafische Darstellung vorbereitet werden. Die andere zeichnet die vorbereiteten Inhalte auf den Schirm.

Um das Zusammenspiel beider Methoden zu verstehen, unternehmen wir eine Wanderung in das Gebirge der "Klassenbibliothek". Das sind die Klassen, die mit Squeak mitgeliefert werden. Der Teil der Klassenbibliothek, den wir besuchen werden, heißt "`Morphic`", er ist für die Benutzerschnittstelle verantwortlich.

Um die Methoden `step` und `drawOn:` zu verstehen, wollen wir zu den Quellen gehen, zu den Stellen nämlich, die die Aufrufe dieser Methoden ursprünglich auslösen.

Wir starten im Tal bei der Methode `WatchMorph>>step` und steigen auf, bis wir in das Gipfelgebiet kommen, aus dem wichtige Aufrufe für alle Anwendungen kommen. Wir werden in diesem Gebiet auch den Ausgangspunkt für die Aufruffolge finden, an deren Ende `WatchMorph>>drawOn:` steht. Dieser Aufruffolge nachgehend werden wir schließlich wieder im Tal ankommen.

Wir sagten bereits, dass die `step`-Methode in definierten Abständen von der Klassenbibliothek aufgerufen wird. Bei allen Morphs ist diese Methode vorgesehen, alle zeitlich veränderlichen Morphs implementieren sie. Wer genau ruft sie auf? Wir beginnen mit dem Aufstieg in das Klassengebirge, den wir mit einigen Auslassungen wiedergeben:

- `WatchMorph>>step`
- (...)
- `WorldState>>runStepMethodsIn: aWorld`
- (...)
- `WorldState>>doOneCycleFor:`

- (...)
- `PasteUpMorph>>doOneSubCycle`
- `MenuMorph>>invokeModalAt:in:allowKeyboard:`

Wir sind wir nun oben am Gipfel angelangt, beim Start einer jeden Anwendung (durch `MenuMorph`). Die Klassen `PasteUpMorph` und `WorldState` sind eng miteinander verbunden. Jene enthält die Daten und besorgt die Steuerung des Welt-Morphs. Diese stellt die Welt grafisch dar, auf die die vielen einzelnen Morphs "aufgeklebt" werden (PASTE UP). Die Welt ist die eine Grafik, die den ganzen Bildschirm von Squeak ausfüllt und zuunterst liegt.

Nach einer kleinen Rast am Gipfel nun zum Abstieg, den wir auf einem anderen Weg vornehmen. Wer sorgt dafür, dass letztlich in der Uhrenanwendung `WatchMorph` die Zeichenroutine `drawOn:` aufgerufen wird? Die Initiative geht hier von der Klasse aus, die für die grafische Darstellung verantwortlich ist, also von `PasteUpMorph:`

- `PasteUpMorph>>doOneSubCycle`
- (...)
- `WorldState>>doOneCycleFor:`
- (...)
- `PasteUpMorph>>displayWorld`
- (...)
- `WorldState>>drawWorld:submorphs:invalidAreaOn:`
- `PasteUpMorph>>drawOn:`
- (...)
- `PasteUpMorph>>drawSubmorphsOn: aCanvas`
- `Canvas>>fullDrawMorph: anObject`
- `Morph>>fullDrawOn: aCanvas`
- `Canvas>>drawMorph: aMorph`
- `Morph>>drawOn:aCanvas`

Welt-Modell (`WorldState`) und Welt-Darstellung (`PasteUpMorph`) sind eng miteinander verzahnt. Auf jede Neuberechnung von geometrischen Daten folgt ein Zeichenbefehl, der die neuen Geometrien zur Ansicht bringt.

Dieser Arbeitsteilung gesellt sich eine weitere hinzu. Zwischen einem darstellbaren Ding (`Morph`) und dem darstellenden Gerät (`Canvas`)[2] müssen wiederum feine Abstimmungen stattfinden. Dies drückt sich in der dargestellten Abfolge aus, die schließlich bei einem `drawOn:` endet.

Diese kurze Darstellung zeigt, dass beide betrachteten Methoden, `step` und `drawOn:` von der Klassenbibliothek aufgerufen werden. Wir können diese Methoden für unsere Zwecke spezialisieren. Darum, dass sie aufgerufen werden, müssen wir uns keine Gedanken machen, denn dafür sorgen die Klassen aus dem `Morphic`-Paket.

1.3.2 Eingebettete bewegte Morphs

Nun zu unseren ersten eigenen Überlegungen. Wir beginnen unsere Implementierung mit einer Klasse `AnalogClock`, die wir von `CircleMorph` ableiten. Das soll das Ziffernblatt sein. Drei Zeiger sollen auf ihm umlaufen.

Was die Zeiger angeht, wollen wir etwas anders vorgehen als die Anwendung `WatchMorph.` Dort sind die Zeiger keine eigenen Objekte. Es werden vielmehr nach jedem `step` durch `drawOn:` Linien für die drei Zeiger gezeichnet.

Dieser Objektentwurf hat zwei Nachteile. Wenn `drawOn:` ausgeführt wird, wird die gesamte Uhr gezeichnet. Etwas unwirtschaftlich, denn von Sekunde zu Sekunde ändert sich z.B. die Ziffernblattbeschriftung nicht. Weiter ist die Form des Zeigers im Programm "fest verdrahtet". Wenn die Programmiererin dem Zeiger eine andere Form geben möchte, müsste sie den Code von `WatchMorph>>drawOn:` ändern.

Wir wollen anders verfahren und aus jedem Zeiger ein eigenes Objekt machen. Damit können wir später dem Benutzer auch die Möglichkeit einräumen, bei Bedarf eigene Vielecke als Formen für die Zeiger zu übergeben. Wir sehen also für den Uhrzeiger ein `Morph` vor.

Wir wollen die Zeiger von `PolygonMorph` ableiten, um die Zeigerform als Polygonzug frei gestalten zu können. Für die drei Zeiger schaffen wir entsprechende Instanzvariablen.

Die Zeiger sollen sich über dem Ziffernblatt drehen. Sie sind, so könnte man sagen, in das Ziffernblatt eingebettet (TO EMBED). Die Einbettung drückt sich darin aus, dass die Zeiger sich mit verändern, wenn die Uhr in Lage oder Größe verändert wird.

Wie konstruiert man eine Gruppe von Morphs, bei denen mehrere eingebettete Morphs auf ein enthaltendes Morph geometrisch bezogen werden?

Wir werfen ein Blick in die Darstellung [5]. Dort steht, dass einem `Morph` mit `addMorph:` eingebettete Morphs (`submorphs`) zugeordnet werden können. Submorphs sind Morphs, die in einem übergeordneten Morph enthalten sind. Alle Submorphs zusammen genommen werden in einem geordneten Objektbehälter `(OrderedCollection)` aufgelistet.

Wenn mehrere eingebettete Morphs jeweils eine eigene Methode `drawOn:` haben - in welcher Reihenfolge werden diese dann aufgerufen? `AnalogClock` ist das einzige Morph unserer Anwendung, das in der Liste der Submorphs der Welt steht. Es wird als Einziges von `PasteUpMorph>>drawSubmorphsOn:` erfasst. Die in `AnalogClock` eingebetteten Morphs kommen erst bei `Morph>>fullDrawOn:` in's Spiel.

[2] ein Abstraktum, das für Bildschirm, Pixeldatei, Druckerdatei stehen kann; wörtlich übersetzt ist es eine Leinwand

Wir erstellen einen ersten Zeiger, den Sekundenzeiger, und ordnen ihn der `AnalogClock` zu. Wir sorgen bei der Konstruktion des Zeigers dafür, dass er vom Zentrum des Ziffernblattes aus radial ausgerichtet ist. Das geschieht in `AnalogClock>>initializeWatchHands`.

Die meisten Anwendungsfälle (USE CASE), die man sich für eine Uhr ausdenken kann, betreffen das Zusammenspiel der grafischen Elemente untereinander, also vor allem von Ziffernblatt und Zeiger. Darüber hinaus gibt es aber auch das Zusammenspiel der gesamten Uhr mit der Umgebung in die sie eingebettet ist.

Was geschieht z.B. wenn der Benutzer mit dem Halo-Element "change size" die Größe der Uhr verändert? Wird dann die Größe des Zeigers entsprechend mit verändert? Unser Objektentwurf muss auch mit einem solchen Anwendungsfall zurecht kommen.

Wir haben also zwei Anforderungen, die im Zusammenspiel zwischen einbettendem Element (`AnalogClock`) und eingebettetem bewegten Element (`WatchHand`) zu behandeln sind:

1. Relative Bewegung der Zeiger zum Ziffernblatt und
2. Synchronisation der Geometrie von Zeiger und Ziffernblatt (bezüglich gemeinsamer Punkte, der Ausrichtung und der Größe).

Die erste Anforderung betrachten wir im folgenden Abschnitt 1.3.2.1, die zweite Anforderung wird in 1.3.2.2 behandelt.

1.3.2.1 Synchronisierte Animation

Wenn ein Morph "in die Welt gesetzt" wird, beginnt die Ausgabe von step-Aufrufen mit der Rückkehr aus `openInWorld`. Das Stepping bezieht sich immer auf die Submorphs eines Morphs, das in `World` geöffnet wurde (siehe in `Morph>>openInWorld:` die Anweisung `aWorld startSteppingSubmorphsOf: self`).

Will man von der Standardperiode von 1000 ms abweichen, muss man im Submorph eine Methode `stepTime` bereitstellen.

1.3.2.1.1 Ein step-Vorgang für mehrere Instanzen Es ist normalerweise vorgesehen, dass nach `openInWorld` eines Morphs für alle seine Submorphs `step` ausgeführt wird. Das ist für die meisten Anwendunen günstig. In unserem Fall, wo die Zeiger Submorphs der Uhr sind, hat es aber auch Nachteile.

Wenn jeder Zeiger seine eigene `step`-Methode hat, dann würde die Uhrzeit zu leicht unterschiedlichen Zeiten bestimmt. Das hätte schlimmstenfalls zur Folge, dass die Stellungen der Zeiger logisch nicht miteinander vereinbar sind. Der Sekundenzeiger steht vielleicht bereits auf der 60, der Minutenzeiger schaltet aber erst eine Sekunde später weiter, dann nämlich wenn der Sekundenzeiger auf die Position 1 schaltet. Das würde der Erwartung widersprechen, die verlangt, dass der Minutenzeiger genau dann weiterschaltet, wenn der Sekundenzeiger von der Position 59 auf die Position 60 wechselt.

Wir wollen diesen Punkt nun positiv, nämlich als Anforderung formulieren. Alle Zeigerelemente werden zwar nacheinander gezeichnet - das ist unvermeidbar. Aber es muss

dabei eine einzige Uhrzeit zugrundeliegen. Nur dann sind die Positionen der Zeigerelemente zueinander stimmig.

Wie erreicht man, dass mehrere Submorphs von einem einzigen `step`-Aufruf versorgt werden? Man sorgt dafür, dass `AnalogClock` nur ein einzigs Submorph erhält, das eine `step`-Methode enthält. In unserem Fall wird diese Rolle von `Mechanics` übernommen, das für das Uhrwerk steht. `Mechanics` wird dann pro Zyklus als einziges Morph aufgerufen. Er stellt als Einziges die Uhrzeit fest und versorgt dann die Zeiger, die auch Submorphs von `AnalogClock` sind, mit einer Uhrzeit, die für alle gleich ist.

1.3.2.1.2 Einen step-Vorgang testen

Die `step`-Methode ist von `Morphic` für Animationen vorgesehen. Wie testet man eine selbst geschriebene `step`-Methode?

Der einfachste Schritt ist, die `step`-Methode direkt aufzurufen: `aMorph step.` Etwas näher an die Realität kommt man, wenn man dafür sorgt, dass alle Morphs weiter geschaltet werden: `aMorph world runStepMethods`. Schließlich verhält man sich genau so, wie es die Klassenbibliothek im schnellen Durchlauf macht, wenn man `aMorph world doOneCycleNow.` aufruft.[3]

1.3.2.1.3 Jitter oder: Kleinvieh macht auch Mist oder: am Übergang von digital zu analog

Bei Anwendungen wie der Analog-Uhr kommt es darauf an, dass die `step`-Methoden regelmäßig aufgerufen werden. Jeder, der einmal einen Sekundenzeiger beobachtet hat, der eine Sekunde überspringt, wird die Uhr für etwas unzuverlässig halten. Niemand wird mit einem Sekundenzeiger zufrieden sein, der mal kürzer und mal länger für die Sekunde braucht.

Schwankungen in der Zeitdauer nennt man auf Englisch Jitter. Wir wollen den Jitter des Sekundenzeigers messen.

Dazu wird in der Methode `drawOn:` von `WatchHand` eine Messung eingebaut. Sie hält den Zeitpunkt des Zeichnens fest - `secondsLastDrawMs` - und auch den zeitlichen Abstand zur letzten Zeichenoperation - `secondsDrawDeltaMs`. Natürlich macht diese Messung vor allem beim Sekundenzeiger Sinn.

Man ruft nun die Uhr auf, öffnet einen Inspektor auf den Sekundenzeiger und beobachtet diese beiden Variablen.

Der Wert für `secondsDrawDeltaMs` , der in Millisekunden angegeben wird, sollte nicht zu sehr um 1000 streuen. Aus dieser Anforderung läßt sich auch leicht ein Unit-Test bauen.

1.3.2.2 Synchronisierte Geometrie

Ein Morph ist in ein anderes eingebettet. Im einfachen Fall stellt sich das so dar:

```
c := CircleMorph new.
c openInWorld.
p := PolygonMorph new.
c addMorph: p.
```

[3] von http://www.visoracle.com/squeakfaq/morphic-step-0.html

Wir bewegen den Kreis, verändern ihn in seiner Größe. Es zeigt sich, dass das kleine Dreieck dabei unverändert an der oberen linken Ecke der den Kreis umgebenden Rechtecklinie (`bounding box`) verharrt.

Das lässt sich verbessern. Wenn wir statt `addMorph:` `addMorphCentered:` verwenden. Das Polygon ist nun zu Beginn zentriert. Leider hält auch diese verbesserte Eigenschaft nur so lange, bis wir die Uhr vergrößern.

Wie können wir das Verhalten unserer Zeiger weiter verbessern? Wie läßt sich Position und Größe der Zeiger mit Position und Größe des einbettenden Morphs synchronisieren? Wir betrachten einen verwandten Fall, für den es bereits eine Implementierung gibt.

Wenn die eingebetteten Morphs rechteckförmig und unbewegt sind, gibt es eine Lösung in der Klassenbibliothek. Man gibt über eine `LayoutPolicy` an, wie zu verfahren ist. Zu Einzelheiten siehe: LayoutPolicy. Ein gutes Beispiel ist in How to lay out submorphs 4 enthalten. Wir geben es hier wieder in einer Form, die zur Ausführung in einen Workspace kopiert werden kann.

```
  m := Morph new.
  m openInWorld.
  m color: Color green.
  m extent: 300 @ 300.
  m layoutPolicy: ProportionalLayout new.
  bm := Morph new.
  bm openInWorld.
  m addMorph: bm fullFrame: ( LayoutFrame fractions: (0@1 corner: 1@1) offsets:
(0@100 negated corner: 0@0) ).
  bm color: (Color blue alpha: 0.5).
  sm := Morph new.
  sm openInWorld.
  m addMorph: sm fullFrame: ( LayoutFrame fractions: (0@0 corner: 1@1) offsets:
(0@0 corner: 0@100 negated)).
  sm color: (Color red alpha: 0.5).
  sm layoutPolicy: ProportionalLayout new.
  tm := Morph new.
  tm openInWorld.
  sm addMorph: tm fullFrame: (LayoutFrame fract0@0ions: (0@0 corner: 0.9@0.9)
offsets: nil).
  tm color: (Color yellow alpha: 0.5).
  tm layoutPolicy: TableLayout new.
  tm listDirection: #topToBottom.
  tm listCentering: #center.
  fm := Morph new.
  fm openInWorld.
  tm addMorph: fm.
  fm extent: 100@2.
  fm color: Color black.
  fm hResizing: #spaceFill.
```

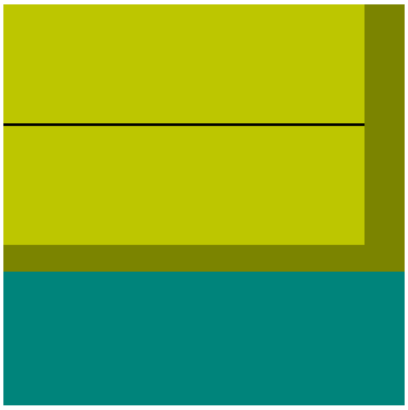

Abbildung 1.3: Eingebettete `RectangleMorphs`

Was geschieht hier? Das einbettende Morph legt mit `m layoutPolicy: ProportionalLayout new` eine Layout-Art fest. Die eingebetteten Rechtecke sollen in Länge und Breite proportional zu den Rechtecken vergrößert werden, in die sie eingebettet sind. Das einzubettende Morph wird mit `addMorph: fullFrame:` zugeordnet. Hier wird außer den beiden Partnern die an der Einbettung direkt beteiligt sind noch eine Geometrie-Information einbezogen, es ist ein Objekt der Klasse `LayoutFrame`.

Ein `LayoutFrame` gehört zusammen mit `Point` und `Rectangle` in die Klassenkategorie `Graphics-Primitives`. Man erzeugt es mit seiner Klassenmethode `fractions: offset:`. Mit dem Parameter `fractions:` gibt man die Position des einzubettenden Rechtecks in Anteilen des einbettenden Morphs an. Für die Anteilsziffern sind Werte zwischen 0 und 1 anzugeben. Mit LayoutFrame fractions: (0.1@0.1 corner: 0.5@0.3) setzt man die obere linke Ecke des neuen Morphs je ein Zehntel von der oberen linken Ecke des äußeren Morphs in x- und y-Richtung ab. Die untere rechte Ecke des neuen Morphs wird 5 Zehntel in x-Richtung und 3 Zehntel in y-Richtung von der oberen linken Ecke des äußeren Morphs abgesetzt.

Für den Parameter `offsets:` gibt man keine Anteilsziffern sondern absolute Zahlen an. Man zählt die Zahl der Pixel. Auf diese Weise kann man einen Punkt des neuen Morphs mit einem festen Versatz gegenüber dem äußeren Morph versehen. Das geschieht bei `LayoutFrame fractions: (0@1 corner: 1@1) offsets: (0@100 negated corner: 0@0).` Diese Angabe schafft einen Fläche, die sich von der unteren Grundlinie aus um 100 Pixel nach oben erstreckt.

Wenn ein Morph so verändert wird, dass das Auswirkungen auf das Layout der eingebetteten Morphs haben könnte, wird `layoutChanged` aufgerufen. Wenn man sich die Sender von `layoutChanged` ansieht, kann man sehen, dass dies in einer Reihe von Situationen erforderlich ist.

Gehen wir noch kurz durch, was in diesem Beispiel nacheinander geschieht. Es werden Morphs erzeugt und in bereits vorhandene Morphs eingebettet:

1. `bm`, ein dunkelgrünes Morph wird unten und

2. `sm`, ein braunes Morph wird oben in das grasgrüne `m` eingebettet. `m` hat für sich ein `ProportionalLayout` festgelegt. Für `bm` wird mit `fractions:` ein entartetes Rechteck, nämlich eine Linie angegeben, für `offsets:` gibt es nur eine Angabe in Pixel. Bei `sm` ist es ähnlich, nur dass hier die untere rechte Ecke nicht bis zur unteren rechten Ecke des äußeren Morphs reicht sondern bereits 100 Pixel weiter oben liegt. Verändert man `m` in der Größe, dann sieht man, dass weder `bm` noch `sm` folgen. Obwohl mit `ProportionalLayout` definiert sind sie über den `offsets:` Parameter auf eine Pixelzahl festgelegt.

3. `tm`, ein ockerfarbenes Morph wird seinerseits in das braune `sm` eingebettet, für das auch ein `ProportionalLayout` eingestellt ist. `tm` wird aber über `fractions:` eingestellt, es hat eine untere rechte Ecke, die sich proportional zu der Fläche des äußeren Morphs `sm` einstellt. Schließlich wird

4. `fm`, ein schwarzes, strichartiges Morph in das ockerfarbene `tm` eingebettet.

Wir näheren uns unserem Ziel, die Geometrie zweier grafischen Elemente zu synchronisieren. Aber wie lässt es sich wirklich erreichen? Wie kann der Fall mit behandelt werden, bei dem sich die grafischen Elemente zueinander bewegen?

1.3.2.2.1 Ein Objektentwurf für eingebettete, bewegte Morphs Eine praktische Untersuchung wird uns helfen, zu verstehen, was während unserer Problem-Anwendungsfälle "move" und "resize" geschieht.

Wir stellen einen Kreis dar und betten in ihn einen Polygonzug ein. Wir lassen uns eine kleine Rückmeldung geben, wann der Kreis neu gezeichnet wird. Dazu überschreiben wir bei `CircleMorph` die Methode `drawOn:` :

```
drawOn: aCanvas
super drawOn: aCanvas.
Transcript show: 'c'.
```

Diese Methode wird aktiv, wenn Kreise zu zeichnen sind. Sie ruft als erstes das Standardverhalten aller Morphs auf. Als Zusatz beim Zeichnen von Kreisen gibt sie auf dem Transcript ein kleines 'c' aus. Was geschieht nun bei Bewegungen und Größenänderungen? Während des Verschiebens wird `drawOn:` nicht aufgerufen, nur am Beginn und Ende. Während des Vergrößerns wird die Methode aber ständig aufgerufen.

Wenn wir also bei Verschiebungen oder Größenänderungen `drawOn:` selbst kontrollieren, haben wir es in der Hand, dass alle eingebetteten Morphs richtig gezeichnet werden, dass mithin keine Zeichnungsfragmente irgendwo liegen bleiben. Haben wir Verschiebungen oder Größenänderungen und zusätzlich eine veränderte Uhrzeit, dann wird zusätzlich `step` aufgerufen und in dessen Folge wiederum `drawOn:`.

Soweit so gut. Unser Plan ist, uns bei `AnalogClock` darauf zu verlassen, dass das geerbte `drawOn:` das Nötige macht. Für die in `AnalogClock` eingebetteten Morphs fügen wir am Beginn von `drawOn:` ein `syncGeometryWith:` ein. Dort wird geprüft, ob die Geometriedaten, die sich das abhängige Morph gemerkt hat, noch übereinstimmen mit den

Geometriedaten des einbettenden Morphs. Ist dies der Fall, dann ist keine Größensynchronisierung nötig. Stimmen sie nicht überein, dann müssen die Größen des eingebetteten Morphs mit der Größe des Besitzermorphs synchronisiert werden.

Ein Problem bleibt. Wir zeichnen z.B. den Stundenzeiger bei jedem `step` neu, obwohl er sich viele Male gar nicht bewegen muss[4]. Die `drawOn`:-Methode dürfte ruhig etwas "bequemer" (LAZY) sein, und könnte sich darauf beschränken, nur dann den Zeichenstift zu bemühen, wenn auch wirklich etwas Neues zu zeichnen ist.

Unsere kleine Verbesserung könnte man als "bequemes Zeichnen" bezeichnen. Wir merken uns bei den Zeigern den Radianwert der Vorperiode (`radianLag1`)[5]. Beim Zeichnen mit `rotateLazyAt:` wird zunächst mit `rotationDue` geprüft, ob der aktuelle Winkel im Bogenmaß (`radian`) von dem beim letzten Durchgang verwendeten (`radianLag1`) abweicht. Nur in diesem Fall wird `rotateAt:` aufgerufen.

Wir können als Ergebnis festhalten: wenn Pick up-, drag- oder resize-Operationen laufen, wird eingebetteten Morphs immer dann die Gelegenheit zur Selbstdarstellung gegeben, wenn das einbettende Morph sie bekommt. Ob sie die Gelegenheit wahrnehmen, liegt daran, wie sie ihre aktuelle Geometrie im Vergleich mit der des einbettenden Morphs sehen. Und daran, ob die Uhrzeit eine veränderte Position verlangt.

Diese `drawOn:`-Methode für eingebettete Morphs spart uns viele unnötigen Zeichnungen. Ein willkommener Nebeneffekt ist, dass wir damit die Freiheit gewinnen, den step-Takt zu erhöhen. Ein schnellerer step-Takt führt zu häufigeren Aufrufen von `drawOn:` aber die kosten nicht viel Rechenzeit, denn in den meisten Fällen wird es heißen: hier ist nichts zu tun, der Zeiger steht bereits so, wie er stehen soll. Was sich aber mit dem schnelleren Takt verbessern lässt, ist die Akkuratheit der Zeigerbewegung gegenüber der Referenzuhr. Auf diesen Aspekt des Jitters sind wir bereits im Abschnitt 1.3.2.1.3 eingegangen.

1.3.3 Morphic und MVC

Über das "model-view-controller" Architekturmuster (MVC-PATTERN) findet man in der Literatur und im Netz viel Information, über Morphic wenig. Wie läßt sich Morphic dem verbreiteten MVC-Architekturmuster gegenüberstellen? Wieviel MVC steckt in Morphic? Wie läßt sich ein Morph verstehen, wenn man in der Begriffswelt von MVC denkt?

Ein Morph ist für die grafische Darstellung eines Grafikelements verantwortlich und für die Reaktion auf Benutzeraktionen. Es ist eine Kombination aus View und Controller. Wo bleibt dann der dritte Teil dieses Architekturmusters, das Modell?

Man kann dem Morph ein Modell zuordnen. Das ist auf verschiedene Weise möglich.

- Eine Gruppe von Methoden und mit ihnen verbundenen Instanzvariablen werden dem Morph hinzugefügt,

[4] Es ist reizvoll darüber nachzudenken, wie oft ein Stundenzeiger innerhalb einer Stunde wirklich neu zu zeichnen ist. Eine unabhängige Größe müsste das Rechteck sein, das die Uhr umgibt und die Zahl der Pixel, die es einschließt. Eine weitere die Form des Stundenzeigers.

[5] "lag" heißt Zeitverzögerung und wir können uns lag wie einen Operator vorstellen, der zu einer gegebenen Variable radian den Wert ermittelt, den diese Variable in der letzten Periode hatte.

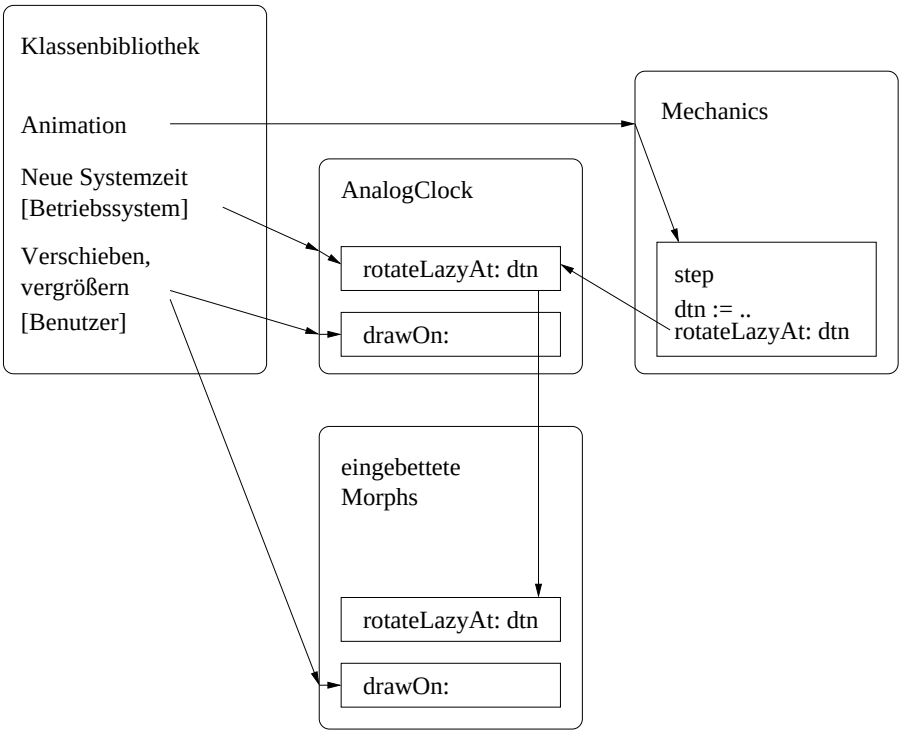

Abbildung 1.4: Äußere Ereignisse, die AnalogClock verarbeitet

- Ein MorphExtension fügt einem Morph zusätzliche Attribute hinzu, dazu kann auch spezifisches Verhalten treten,
- Eine eigene Klasse kann erstellt und in der Rolle eines Modells mit dem Morph verbunden werden.

Das Morph in der Rolle als Sicht muss dann auf Veränderungen des Modells hören und gegebenenfalls reagieren.

Weitere Auskünfte zu Morphic finden sich in [56], S. 35.

1.3.4 CRC-Karten

Wir schreiben für jede Klasse (CLASS) ihre Verantwortlichkeiten auf (RESPONSIBILITY) sowie die Klassen, mit denen sie zusammenarbeiten muss, damit sie ihre Verantwortung wahrnehmen kann (COLLABORATION). Mit einem Kürzel, das aus den Anfangsbuchstaben der drei englischen Schlüsselwörter gebildet wurde, bezeichnet man diese Sammlung von Designinformationen auch als CRC-Karten. Wenn man über den Objektentwurf nachdenkt oder darüber mit Anderen spricht, kann es nützlich sein, diese Informationen auf Karten zu schreiben. Denn Karten lassen sich beliebig neu anordnen.

Die CRC-Informationen finden sich im Klassenkommentar[6].

1.3.4.1 Wofür sind einzelne Klassen verantwortlich?

Neben den verteilten Informationen auf den CRC-Karten kann es aber auch sinnvoll sein, Angaben zum Entwurf an einer Stelle zu bündeln, z.B. in einer Tabelle. Wir zeigen in Tabelle 1.2 einige wichtige Gemeinsamkeiten und Unterschiede zwischen den Klassen unseres Pakets.

In Spalte 1 und 2 sieht man, von welchen Klassen aus der Klassenbibliothek wir die Klassen unserer Anwendung abgeleitet haben. Die Grenze zwischen der ersten und der zweiten Spalte ist auch die Grenze zwischen mitgelieferten und selbst geschriebenen Klassen. Ab Spalte drei sind verschiedene Eigenschaften und Verantwortlichkeiten aufgeführt.

Es sind zum Beispiel alle Klassen gekennzeichnet, die sichtbar und bewegt sind. Nicht alle sind submorphs von `AnalogClock`. Wir haben die `SpiralEvents` als submorphs `Spiral` zugeschlagen. Damit liegen alle Dinge, die mit der Spiralform zu tun haben, in einer Hand, in der Hand von `Spiral`. Auf die Weise ist eine zweistufige Hierarchie von submorphs entstanden.

Wie oben bereits angesprochen ist die Klasse `Mechanics` ein submorph von `AnalogClock`, der aber keine Ausdehnung hat und damit unsichtbar bleibt. Diese Konstruktion dient dazu, dass `Mechanics` eine `step`-Methode bekommt und alle taktbedürftigen Klassen mit dem gleichen zentralen Takt versorgen kann.

`SpiralCalendar` ist eine Kombination aus Model und Controller. Die Klasse hat als Partner `Spiral`, der für die Sicht verantwortlich ist.

[6] Wenn man eine Klasse frisch anlegt oder bei einer bereits angelegten den Klassenkommentar leert, gibt Squeak eine Vorlage für Grundinformationen vor. Hinter diese Grundinformationen haben wir die CRC-Informationen geschrieben.

Elternklasse der Klassenbibliothek	Klasse der Anwendung	Anwendungsstart	sichtbar und bewegt	submorph von AnalogClock	submorph von Spiral	zentraler Takt	Collection aller SpiralEvents
Circle-Morph	Analog-Clock	x					
Circle-Morph	Mechanics			x		x	
Polygon-Morph	SpiralEvent		x		x		
Polygon-Morph	WatchHand		x	x			
WatchHand	Spiral		x	x			
Object	Spiral-Calendar						x

Tabelle 1.2: Klassen und Verantwortlichkeiten

1.4 Modultest

Während der Entwicklung wird "Spielraum" um die neu zu entwickelnde Funktionalität herum benötigt. Man möchte zum Beispiel sehen, wie es aussieht, wenn sich der Sekundenzeiger alleine bewegt. Oder man möchte die Uhr schnell vor- oder zurücklaufen lassen. Diese Anforderungen kommen nicht von unserem Auftraggeber, es sind unsere eigenen. Sie unterstützen die Entwicklung. Wenn wir sie früh formulieren und umsetzen, hilft uns das, einzelne Aspekte der Funktionalität isoliert zu testen. Und das macht uns sicherer und schneller.

Wir hatten in 1.1 zwischen Modultests und Abnahmetests unterschieden. Modultests prüfen einzelne Bestandteile der Software, Abnahmetests zielen auf die ganze Anwendung.

Ein weiteres Unterscheidungsmerkmal ist aus praktischer Sicht wichtig. Der Modultest soll nicht auf einen menschlichen Betrachter angewiesen sein. Er soll vielmehr dem Entwickler eine rasche Rückmeldung geben, dass seine jüngste Änderung nichts angetastet hat, was bisher funktionierte.

Der Abnahmetest hingegen ist für denjenigen, der die Entwicklung in Auftrag gegeben hat. Dieser Test muss nicht automatisiert laufen und darf auch längere Zeit brauchen. Es sind der Auftraggeber und seine Vorstellungen, die für die Gestaltung des Abnahmetests maßgeblich sind.

Auf den Unit Tests gehen wir in diesem Abschnitt ein. Der Abnahmetest folgt im nächsten.

Jeder Schüler weiß: eine Mathe-Hausaufgabe muss geprüft werden. Jeder Handwerker weiß: eine neu installierte Rohrleitung muss geprüft werden. In diesem Sinne ist auch Software kein Deut anders wie eine Mathematikaufgabe oder eine handwerkliche Dienstleistung zu sehen.

Mit dem Schreiben von Modultests befasst sich [16], S. 155ff. Einen besonderen Aspekt, nämlich die Zuordnung von Unit Tests zu Methoden, betrachtet [42].

Man kann sich fragen: muss denn jede Methode getestet werden? Oder genügt eine Auswahl? Müssen alle Aufrufmöglichkeiten durchgespielt werden oder reicht ein Teil davon? Diese Fragen lassen sich so beantworten: Module, die schwierig zu verstehen oder aus anderen Gründen fehlerträchtig sind, sollten durch Modultests abgesichert werden. Es gibt Anhaltspunkte dafür, wann eine Methode schwer verständlich und damit fehlerträchtig ist.

1. die Methode selbst ist kompliziert, z.B. weil der verwendete Algorithmus anspruchsvoll ist,

2. es gibt kein 1:1-Verhältnis zwischen Anforderung und implementierter Methode. Z.B. wird eine Anforderung in drei Methoden umgesetzt oder aber drei Anforderungen werden in einer Methode umgesetzt.

Die vollständige Testsuite der Modultests findet sich in `AnalogClockUnitTest`. Sie wird mit dem `TestRunner` ausgeführt.

1.4.1 Unit Tests bei grafischen Oberflächen

Unit-Tests für grafische Oberflächen sind nicht ganz einfach zu schreiben. Das liegt daran, dass es meistens aufwändig ist, in einer Zusammenstellung vieler einzelner visueller Elemente wie z.B. unserer Uhr durch automatische Analyse der Bildschirmdaten die einzelnen Elemente und ihre Eigenschaften zu erkennen. Welchen Winkel hat der Minutenzeiger gerade? Wenn wir hinsehen, können wir schnell urteilen, ob der Winkel ok ist. Will man das automatisch erkennen, muss man sich mit der Programmierung einer Bilderkennung viel Mühe geben - sehr viel sogar.

Um dieses Problem zu umgehen, klammern wir die grafische Darstellung aus unseren Tests aus und testen unsere Software bis knapp vor der Darstellung. Wenn die errechneten Geometriedaten einen Winkel von 17 Grad ausweisen, unterstellen wir, dass der am Bildschirm dargestellte Zeiger auch tatsächlich im 17 Grad-Winkel zu sehen ist.

1.5 Abnahmetest

Einen Abnahmetest wollen wir wie eine kleine Demonstration für den Auftraggeber verstehen. Der Test sollte die Wunschvorstellungen des "Kunden" treffen. Und er sollte sich dazu eignen, einzeln vom Auftraggeber abgenommen werden zu können. Eine Abnahme setzt sich oft zusammen aus einer Reihe von solchen Demonstrationen und zusätzlich aus freien Prüfungen, bei denen der Auftraggeber nach seinem Belieben Aspekte der Anwendung untersucht und bewertet.

Wir haben zu diesem Zweck `AnalogClockAcceptanceTest` geschrieben. Diese Tests sind im Großen und Ganzen Varianten der Tests in `AnalogClockUnitTest`. Anstelle einer unmittelbaren rechnerischen Prüfung tritt hier eine Sichtprüfung.

Darüber hinaus kann man sich Tests vorstellen, die besser ganz von Hand ausgeführt werden. Wir schildern exemplarisch zwei Testfälle.

1.5.1 Resize-Verhalten

1.5.1.1 Klein nach groß

Aktion AnalogClock mit ResizeHandle etwa auf Bildschirmgröße bringen. Abwarten bis Zeiger nach rechts unten zeigt. Dann in schneller Folge: mit rechter Maustaste Halo erzeugen und mit ResizeHandle sehr klein machen.

Prüfkriterium Bleiben Pixelspuren des großen Zeigers übrig? Wird der neue Zeiger richtig gezeichnet?

1.5.1.2 Groß nach klein

Aktion AnalogClock mit ResizeHandle so klein wie möglich machen.

Prüfkriterium Wird die Uhr in jedem Fall korrekt angezeigt? Gibt es einen "Divide by Zero"-Fehler?

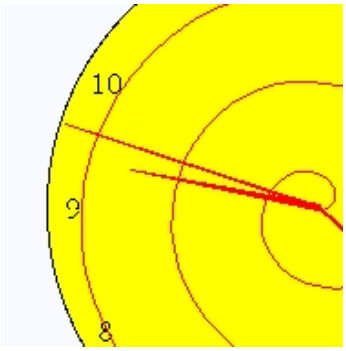

Abbildung 1.5: Der Sekundenzeiger hinterläßt eine waagrechte Lücke an der Ziffer ”9”

Die vollständige Testsuite der Abnahmetests findet sich in `AnalogClockAcceptanceTest`. Sie kann mit dem `TestRunner` ausgeführt werden. Freie Tests können mit dem `ConfigPanel` durchgeführt werden, das einige Funktionen zugänglich macht. Man aktiviert das `ConfigPanel`, indem man im Menü-Halo den Punkt ”`show ConfigPanel`” auswählt.

1.6 Fehler beheben

Wir haben nun einen ganzen Entwicklungszyklus durchlaufen: von den Anforderungen über die Implementierung zu den Modultests und schließlich zu den Abnahmetests.

In diesem Abschnitt wollen wir auf die Fehlerbehebung eingehen. Danach, in Abschnitt 1.7 werden wir zeigen, wie Versionen entstehen, die nach und nach mehr Anforderungen umsetzen, auch die Anforderungen nach guter Qualität.

Keine Software-Entwicklung ohne Implementierungsfehler. Wir haben natürlich während der Entwicklung viele Fehler gemacht. Einen Überblick gibt die Fehlerliste im Projekthandbuch (siehe 1.9.3).

Exemplarisch wollen wir hier auf einen Fehler näher eingehen, dessen Analyse eher kompliziert und langwierig war.

Ein Problem, das sich bei den ersten Versionen der Anwendung hartnäckig gehalten hat, war, dass nach dem Wegbewegen des Sekundenzeigers manchmal die darunter liegenden Ziffern des Ziffernblatts beschädigt wurden wie dies Abbildung 1.5 an der Ziffer ”9” zeigt. Es wurden einzelne Pixel der Schriftzeichen gelöscht und es kam die Farbe des darunter liegenden Ziffernblatts zum Vorschein. Die Ziffern sahen ”angeknabbert” aus.

Der Fehler trat mit der Konfiguration aus

- Version `Squeak3.9, latest update: #7067, Current Change Set: Atomic` und
- der Paket-Version `PVpub-AnalogClock.6.mcz`

auf. Zu dieser Zusammenstellung zweier Versionen wird später noch etwas zu sagen sein.

Was ist hier wohl falsch gelaufen? Wie ist die Schrittfolge beim Zeichnen? Wie kommt man einem solchen Fehler auf die Schliche?

Erstaunlicher Weise war dieses Verhalten nicht zu beobachten, sobald man die Uhr mit der Maus angefasst hatte und verschob. Unter der Regie dieses Betriebszustands wurden alle Pixel korrekt wiedergegeben.

Hilfreich bei der Analyse dieses Fehlers war die Debug-Funktion `Preferences/debugShowDamage`. Schaltet man sie ein, wird jeder rechteckige Bereich vor dem Neuzeichnen kurz markiert. Diese Debug-Funktion wird bei `WorldState>>displayWorld:submorphs:` aufgerufen.

Mit Hilfe dieser Debug-Funktion lässt sich zum Beispiel nachweisen, dass ständig neu gezeichnet wird, wenn man die Eckpunkte der Zeiger-Polygone in der `drawOn:`-Methode setzt. Setzt man sie hingegegen schon vorher und ruft in der `drawOn:`-Methode lediglich das `drawOn:` des Mutter-Polygons auf, dann werden nur genau drei Zeiger gezeichnet.

Briffault ([18], S. 100f.) zeigt, wie man sich Ausschnitte des Displays als Bitmap speichern kann. Das kann helfen, den Fehler anhand seines visuellen Erscheinungsbildes genauer zu verstehen.

```
1
2 | im |
3 im ← Form fromUser.
4 GIFReadWriter putForm: im onFileNamed: 'test.gif'.
```

Listing 1.1: Ausschnitt des Displays als Bitmap speichern

Außerdem kann man bei `Canvas>>frameAndFillRectangle:fillColor:borderWidth:borderColor:` nachvollziehen, wie Rechteck-Bereiche neu gezeichnet werden.

Und was konnte mit diesen Hilfen als Fehlerursache ermittelt werden?

Unser Fehler war: `fullBounds` der `TextMorph` waren als `Float` berechnet worden. Dies geschah in `Dial>>setFiguresFromCenter:andRadius:`.

Richtig ist, hier durchgängig ganze Zahlen zu verwenden; es gibt schließlich kein Drittel-Pixel! Zusatz von `rounded` ergab den Erfolg.

Es steht noch eine Bemerkung zur Zusammenstellung von Versionen aus. Laden wir die oben genannte Version der Anwendung in ein aktuelles Squeak, sagen wir `sq4.2-10966`, dann ist der Fehler nicht zu beobachten, dafür aber ein anderes Fehlverhalten. Zu einer Fehlerbeschreibung gehört also auf jeden Fall eine Zusammenstellung der Versionen aller relevanten Bestandteile des untersuchten Systems, eine Konfiguration. Wer es vollständig machen will, fängt ganz außen an und nennt die Rechner-Hardware, das Betriebssystem mit Version, die Version des Squeak-Images und die Version der Anwendung mit allen für sie erforderlichen Paketen.

1.7 Versionen und Releases

Als Versionen wollen wir solche Software-Stände unserer funktionalen - und Testsoftware bezeichnen, deren Qualität beschrieben ist. Die Versionen legen wir mit dem Tool `Monticello` auf SqueakSource ab.

Als Releases wollen wir ausgewählte Versionen auszeichnen, die auf vorher bestimmte Ziele hin geplant sind und die einem breiteren Kreis an Nutzern zur Verfügung gestellt werden.

Was soll man darunter verstehen, dass die Qualität einer Version beschrieben ist? Das meint vor allem, dass von Version zu Version in einer einheitlichen Weise einige Kennwerte genannt sind, die die Veränderungen zwischen den Versionen charakterisieren. Wieviele und welche Kennwerte nützlich sind, hängt von den Qualitätszielen ab, die man verfolgt und auch ein wenig von den Werkzeugen, die einem zur Verfügung stehen.

Einige Kennwerte können auf einfache Weise erhoben werden. Sie können dabei helfen, die Software zu bewerten. Insbesondere können sie einen darauf aufmerksam machen, dass Rückschritte in der Software-Qualität eingetreten sind.

1.7.1 Anteil implementierter Anforderungen

Im Projekthandbuch sind die Anforderungen aufnotiert, die wir implementieren wollen. Ein einfacher Gradmesser für den Projektfortschritt ist der Anteil implementierter Anforderungen.

1.7.2 Codezeilen (Lines of Code - LOC)

Die meisten Kennwerte verändern sich in Abhängigkeit von der Zahl der Codezeilen, die untersucht wurden. Deshalb sollte man diese Zahl von Anfang an ermitteln. Abhängige Messwerte können dann auf die Zahl der Codezeilen bezogen werden.

Die Codezeilen werden nach Instanz- und Klassen-Codezeilen unterschieden. Mit `AnalogClock Loc` rufen wir eine Klassenmethode auf, die die Zahl der Codezeilen in das Transcript schreibt. Entsprechende Werte für die Testklassen liefert `AnalogClockTest Loc`.

Interessant ist der Anteil der Zeilen in den Testklassen am Gesamt aller Codezeilen: UTLOC / (FuncLOC + UTLOC). Das zeigt, wieviel Anstrengungen auf das Schreiben von Tests verwendet wurden.

1.7.3 Testabdeckung

Eine Anwendung sollte von Unit Tests und Abnahmetests begleitet sein. Da Anwendung und Tests aufeinander bezogen sind, muss für eine Version der Anwendung auch die Versionskennung der zugehörigen Tests angegeben werden. Der interessierte Nutzer kann dann Anwendung und Tests laden und die Tests mit dem `Testrunner` durchführen.

Die Beziehung zwischen Anwendung und Tests kann näher untersucht werden. Der `Testrunner` stellt den Prozentsatz von Methoden fest, die beim Ausführen der Tests durchlaufen werden (METHOD COVERAGE).

1.7.4 Fehler

Die Zahl nicht behobener Fehler auf 1000 Zeilen Code gibt uns ein Gefühl dafür, wie viele Unsicherheiten wir an Bord haben. Fehler, die bekannt aber noch nicht verstanden sind, enthalten eine Menge Unsicherheitspotenzial. Wir wissen nicht, was auf uns wartet.

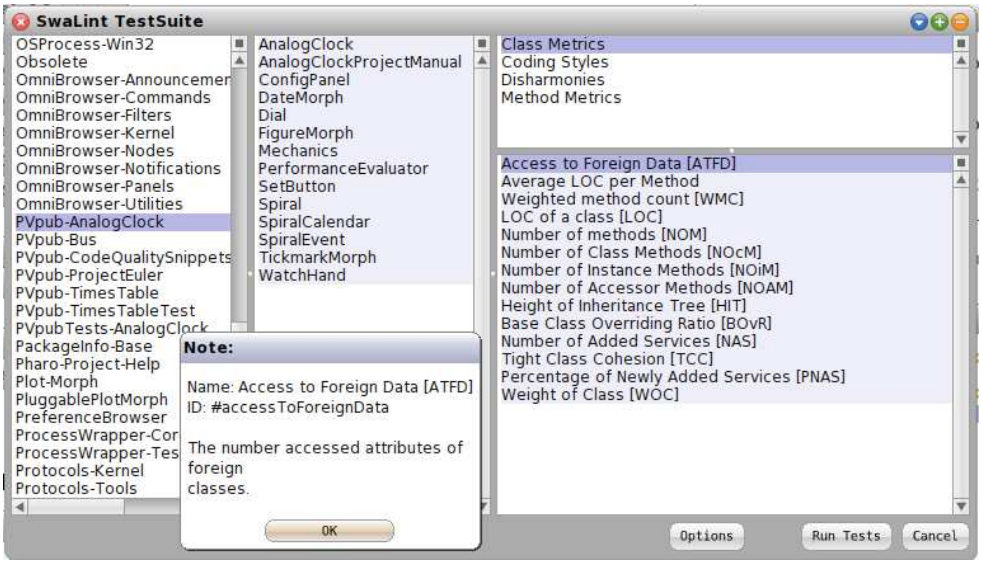

Abbildung 1.6: SWALint

Im Verlauf eines Projekts gewinnt man ein Gefühl dafür, bei welchen Fehlerzahlen man besser keine neuen Dinge entwickeln sondern sich an die Fehlerkorrektur machen sollte.

1.7.5 Fusseln entfernen mit SwaLint

`SwaLint` ist ein Paket, mit dem man eine größere Zahl von Entwurfsregeln prüfen und objektorientierte Metriken bestimmen kann.[7]

Wir sehen in Abbildung 1.6 das Fenster, das sich öffnet, wenn man im Workspace `SwaLint open` eingibt. Das Gleiche erreicht man, wenn man die Anwendung über der Welt-Menü öffnet. Die Benutzerin trifft ihre Wahl indem sie beim linken Teilfenster beginnt und dann im Uhrzeigersinn zum Fenster rechts unten voranschreitet. Zu jedem Listenelement im Fenster rechts unten lässt sich eine kurze Erläuterung aufrufen. Die Markierung lässt sich in drei Stufen mit klicks verändern. Die dunkle Markierung bringt eine Erläuterung.

SwaLint demonstriert, wie gut Smalltalk fähig ist, Smalltalk-Code zu bewerten. Da die Grammatikregeln von Smalltalk einfach sind, ist es auch möglich, die Bedeutung und Wirkung einzelner Anweisungen zu beurteilen. Damit lassen sich missverständliche, fehlerträchtige oder umständliche Konstruktionen ermitteln und dem Programmierer anzeigen.

Wir konzentrieren uns auf SmallLint, eine Teilmenge der Prüfungen, die SwaLint anbietet. Die meisten SmallLint-Regeln sind auf der Webseite LintChecks wiedergegeben.

Es lohnt sich, die SmallLint-Regeln ernst zu nehmen und früh anzufangen, die Zahl der Regelverstöße auf einem Minimum zu halten.

[7] lint ist das englische Wort für Fussel.

Korrigiert man die Verstöße gegen SmallLint-Regeln, dann muss man einige Codestellen neu überdenken. Das verschafft einem in der Regel die Chance, schlecht verständlichen Code zu verbessern. Auch insofern ist es als sinnvolle Code-"Hygiene" anzusehen, die SmallLint-Regeln zu beachten.

Der Ungeübte kann SmallLint als einen guten Sprachtrainer ansehen.

SmallLint wirkt mit den Modultests zusammen. Man möchte auf keinen Fall Einbußen in der Funktion erleiden, wenn man den Code verändert. Wir sichern uns gegen Rückschritte ab durch schnelle und effektive Tests. Es empfiehlt sich, nach jeder SmallLint-Korrektur den `TestRunner` aufzurufen und eventuell auftretende Fehler gleich zu beheben. Erst wenn das aktuelle Lint-Problem gelöst und alle Modultests lauffähig sind, sollte man zum nächsten Lint-Problem voranschreiten.

Es kann einem so vorkommen, dass man sich damit in eine enge Jacke zwängt und zu viele entwicklerische Freiheiten beschnitten bekommt. In gewisser Weise verhält es sich aber genau andersherum. Vernachlässigt man die Lint-Regeln, dann schränkt man seine zukünftigen Freiheitsgrade bei der Entwicklung stark ein.

Als Beispiel für den Umgang mit Lint wollen wir einen Regelverstoß näher besprechen.

1.7.5.1 Fussel "lange Methode" entfernen

Die Lint-Warnung "long method" wird gegeben, wenn Methoden 10 Anweisungen enthalten oder mehr.[8] Die Erfahrung der Smalltalk-Entwickler lehrt, dass man für einen logisch zusammenhängenden "Gedanken" mit weniger auskommt.

Man sollte also die beanstandeten Methoden kürzen. Die folgende kleine Liste zeigt einige Möglichkeiten:

- (Methode wird nur von einer anderen Methode aufgerufen) man verschiebt den Anfang bzw. das Ende der Methode in die rufende,
- man betrachtet die temporären Variablen und gliedert Anweisungsfolgen in eigene Methoden aus, die sich die gleichen Untermengen von temporären Variablen teilen,
- man betrachtet die Aufrufparameter und gliedert Anweisungsfolgen, die sich die gleichen Parameter teilen, in eigene Methoden aus,
- man vermeidet, temporäre Variablen zu bilden, die später nur an einer einzigen Stelle benötigt werden, z.B. für Zwischenergebnisse,
- man setzt Objektbehälter (Collections) ein.

[8] Es gibt zwei Prüfungen mit dem Namen "Long Methods" unter dem Dach von SwaLint. Coding Styles/Long Methods/#longMethods arbeitet mit 7 Zeilen pro Methode und zählt Kommentare mit. SmallLint/Long Methods/#smallLintLongMethods arbeitet mit 10 Zeilen pro Methode und zählt keine Kommentare mit. Die Konvention von CodingStyles geht lt. Kommentar auf Lorenz und Kidd 1996 zurück. Wie haben die Konvention von SmallLint verwendet.

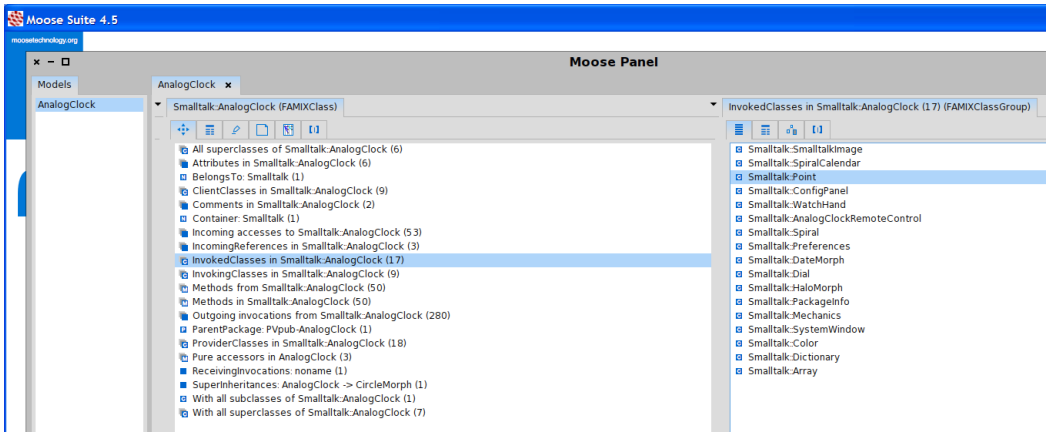

Abbildung 1.7: Moose bestimmt das Fan out von AnalogClock

1.7.6 Kennwerte bestimmen mit Moose

Eine Klasse sollte sich auf die Zuarbeit weniger Klassen im eigenen Paket beschränken. Diese Überlegung leuchtet ein, wenn man sich klarmacht, was geschieht, wenn man die Zusammenarbeit zwischen Klassen später einmal neu ordnen möchte. Je mehr andere Klassen eine Klasse als Dienstleister beschäftigt, desto schwieriger wird es, die Beziehungen zwischen Klassen neu zu ordnen.

Wie sich die Verbindung einer Klasse zu anderen, sie unterstützenden Klassen auffächert, gibt der Kennwert Ausgangsfächerung an (FAN OUT). Dem gegenüber steht das FAN IN, das zählt, von wievielen anderen Klassen die betrachtete aufgerufen wird. Die beiden Maßzahlen kennzeichnen die Binnenstruktur eines Pakets.

Unter praktischen Gesichtspunkten ist vor allem das Fan out relevant. Man kann annehmen, dass ein hohes Fan out nicht nur die (zukünftige) Änderbarkeit der Software einschränkt sondern auch die Fehlerzahl eher nach oben treibt. Deshalb überwachen wir diese Kennzahl.

Technisch lässt sich das Fan out mit dem Werkzeug `Moose` bestimmen. Man erhält das Werkzeug von www.moosetechnology.org (Stand 17-07-11). Eine Einführung in die verfügbaren Analyseinstrumente findet man auf http://www.themoosebook.org/book (Stand 17-07-11).

Da das Werkzeug auf der Smalltalk-Distribution Pharo läuft, ist es lediglich nötig, unsere Anwendung in ein Pharo-Image zu laden. Dann wird sie in Moose importiert und kann analysiert werden. Die Auswertung für das Fan out zeigt Abbildung 1.7.

1.7.7 Qualitätsziele

Bis hierher haben wir ein paar Kennwerte vorgeschlagen. Was aber sollen wir anstreben? Welcher Wert kann als gut, welcher muss als schlecht gelten?

Ohne Qualitätsziele macht das Erheben von Kennwerten nur wenig Sinn. Während der Entwicklung wird man ab und an über die Qualitätsziele nachdenken. Wir haben uns

Metrik	Dokument oder Werkzeug	Min.	real	Max.
Req. implemented / total	Projekthandbuch	70%	71%	90%
UTLOC/(FuncLOC+UTLOC)	Klassenmethode Loc	20%	19%	40%
Unit Test Code Coverage	TestRunner	80%	78%	90%
open bugs/kLOC	Projekthandbuch	3	5	10
open lint problems/kLOC	SwaLint	5	2	10
max fan out	Moose	1	18	10

Tabelle 1.3: Qualitätsziele und reale Kennwerte aus VER062

dafür entschieden, für jeden Kennwert jeweils eine Grenze für ”zu wenig” und für ”zu viel” Qualität festzulegen.

Während sich jeder etwas unter fehlender Qualität vorstellen kann, erschließt sich nicht gleich, was ”zu viel” Qualität bedeuten soll. Gemeint ist, dass die Anstrengungen, die in die Qualitätsverbesserung gehen, in keinem günstigen Verhältnis mehr zur erreichten Qualitätsverbesserung stehen. In diesem Sinn ist die Obergrenze für Qualität zu verstehen, die wir verwenden. Oberhalb dieser Grenze würden sich nur Qualitätszuwächse mit einem ungünstiges Verhältnis von Aufwand zu Nutzen erzielen lassen.

Unser Qualitätsziel legen wir so fest: der gemessene Kennwert soll sich innerhalb dieser Grenzen bewegen. In der Aufstellung 1.3 sind angegeben:

- der Name der Metrik,
- das Dokument, von dem die Messung abgeleitet wird bzw. das Tool, das zur Messung verwendet wird,
- die Grenze, an der wir ”zu wenig Qualität” ansetzen,
- der tatsächlich gemessene Wert für die Pakete `PVpub-AnalogClock` und `PVpubTests-AnalogClock` und schließlich
- die Grenze, die wir für ”zu viel Qualität” ansetzen.

Die Daten für die genannten Metriken können aufgenommen, für das vorgegebene Qualitätsintervall skaliert und dargestellt werden. Abbildung 1.8 stellt die vorgestellten Kennwerte auf sechs sternförmig angeordneten Achsen dar und zeigt, wo noch ”zu wenig Qualität” herrscht, nämlich dort wo die Datenpunkte im Inneren des Rings liegen. ”Zu viel Qualität”, gemessen an unseren eigenen Standards, haben wir für die Korrektur von SmallLint-Auffälligkeiten aufgewendet. Wo wir außerhalb des festgelegten Qualitätsbereichs liegen haben wir die Datenpunkte schwarz dargestellt, innerhalb weiß. Je näher wir der ”guten Mitte” kommen desto größer haben wir die Datenpunkte dargestellt. Bei der Fehlerdichte kommen wir dem angestrebten Optimum an Qualität schon recht nahe.

Aus der Darstellung lässt sich eine Priorisierung von Projektaufgaben ableiten.. Belange außerhalb des festgelegten Qualitätsbereichs müssen erst in diesen zurückgeführt werden. Erst wenn alle Datenpunkte weiß dargestellt sind sollten weitere Anforderungen umgesetzt werden.

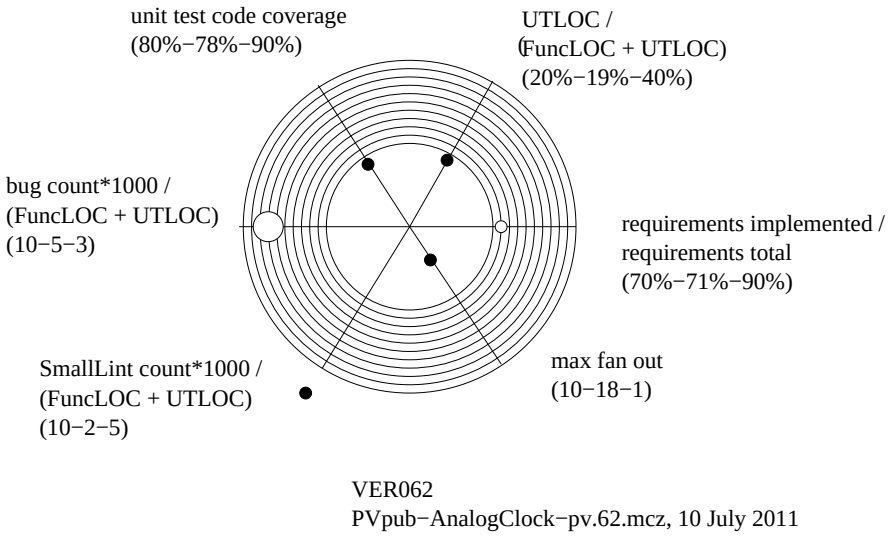

Abbildung 1.8: Qualitätskennzahlen

Dieses Schaubild lässt sich mit einfachen Mitteln aus den gemessenen Kennwerten ermitteln[9]. Eine Schaubilder-Serie zeigt die Entwicklung der Qualitätskennzahlen über die Versionen.

1.8 Stolpern und Weitergehen

Wenn man beim Programmieren

- zu einer Überlegung zurückkommt, die man schon einmal angestellt und danach verworfen hatte,
- immer wieder kleine Änderungen macht, die zwar Teilerfolge bringen aber gleichzeitig auch Nachteile hinsichtlich anderer Programmeigenschaften

dann sollte man wissen, dass dies nichts Unnormales ist und nichts Schlimmes. Ja, man ärgert sich und fühlt sich - je nachdem wie dumm man den eigenen Fehler empfindet - frustriert und vielleicht unfähig.

Ab einer bestimmten Programmgröße ist es normal, dass der Entwickler nur einen Teil der Aufgabenstellung und einen Teil des Programms übersieht.

Zu viel Ärger schadet der guten Laune. Man sollte solche Anzeichen zum Anlass nehmen

[9] Wir verwenden das Projekthandbuch und das Vektorgrafik-Programm XFig.

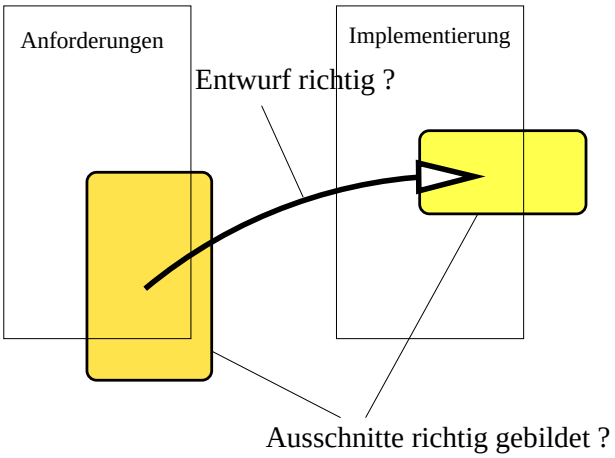

Abbildung 1.9: Nichts Ungewöhnliches: Problembereich und Lösungsbereich werden unvollständig erfasst

- eine Pause einzulegen (die Kreativitätsforschung sagt: möglichst mal etwas ganz Anderes tun),
- ein Fehlverhalten, das man beheben möchte, zu isolieren und genau zu beschreiben. Dabei kommt es darauf an, zunächst nur Fakten und keine Interpretationen festzuhalten. Weiterhin, den allereinfachsten Fall zu erfassen, in dem das Fehlverhalten vorkommt. Und zuletzt, den Fall zu erfassen, in dem das Fehlverhalten so zuverlässig wie möglich reproduziert werden kann.
- zu prüfen ob der Ausschnitt, den man von der Aufgabenstellung überblickt, zu klein ist (oder zu groß und zu oberflächlich)
- zu prüfen, ob der Ausschnitt, den man vom Entwurf überblickt, zu klein ist (oder zu groß und zu oberflächlich)
- zu prüfen, ob der Entwurf, also die Abbildung vom Problembereich auf den Lösungsbereich sich gut erklären läßt,
- darüber nachzudenken, ob es eine alternative Darstellung, vielleicht mit anderen Abstraktionen gibt, mit der man die Komplexität der Aufgabenstellung besser zu fassen bekommt.

Auf welches Ziel sollte man hinarbeiten? Man könnte antworten, auf das Ziel, die tatsächliche Komplexität einer Teil-Aufgabe vollständig zu fassen zu bekommen und sie durch die Entwurfsidee auf einfache Weise zu lösen. Ideal ist, wenn es gelingt, die implementierte Lösung im Code an einer Stelle zu konzentrieren. Auf viele Stellen verteilte Lösungen sind schwieriger zu übersehen und noch schwieriger zu ändern.

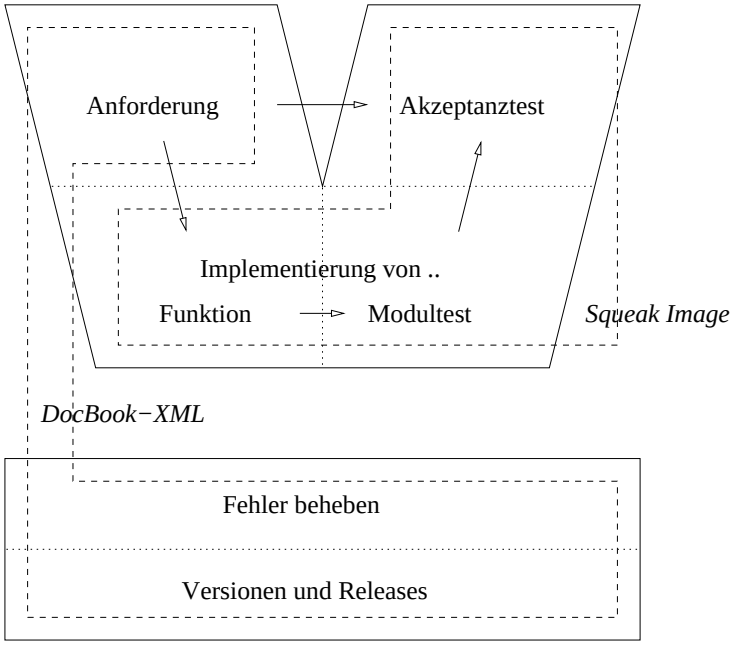

Abbildung 1.10: Dokumentation mit DocBook und im Image

1.9 Dokumentation

Der Ort für Dokumentation in Squeak ist einerseits der Klassenkommentar, andererseits eingefügter Kommentar in Methoden. Für die Dokumentation stehen als Ausdrucksmittel Text und Hypertext-Markierungen zur Verfügung.

Will man eine Kategorie von Klassen oder ein Paket kommentieren, dann gibt es dafür keinen speziellen Ort. Um Platz für die fehlende Paketdokumentation zu schaffen, haben wir eine Klasse `AnalogClockProjectManual` eingeführt, die nur zu Dokumentationszwecken dient. Sie soll für eine Kategorie oder ein Paket als Projekthandbuch dienen. Die Dokumentation findet sich im Klassenkommentar. Wir verfolgen hier das Ziel, die Dokumentation nicht nur innerhalb von Squeak sondern auch außerhalb zu verwenden. Deshalb haben wir das auf XML beruhende DocBook-Format gewählt, das sich gut für medienneutrale und plattformunabhängige Veröffentlichung eignet.

Die Aufteilung der Dokumentation auf zwei Medien ist in Abbildung 1.10 dargestellt.

Wir legen den einfachen Entwicklungsprozess zu Grunde, der in 1.1 beschrieben ist. Eine Implementierung beruht auf Anforderungen[10]. Die Anforderungen werden im Projekthandbuch chronologisch aufgelistet. Sie erhalten einen Index der Art `RQFxxx` oder `RQNxxx`, je nachdem ob es sich um funktionale oder nicht-funktionale Anforderungen handelt.

Sobald eine Softwareversion auf SqueakSource abgelegt wird, wird sie dort nummeriert. Wir verwenden diese Nummer, um das gesamte Paket mit einem Index der Art `VERxxx`

[10]Ohne Anforderungen ließe sich nie beurteilen, ob die Implementierung ihr Ziel erreicht hat.

zu kennzeichnen.

Bei der Nutzung oder während der Entwicklung werden Fehler an der Software entdeckt. Diese werden mit einem Index der Art `BUGxxx` bezeichnet und ebenfalls chronologisch gelistet.

Anforderungen, Versionsinformationen und Fehler können weiter mit Zusatzinformationen versehen und aufeinander bezogen werden.

1.9.1 Anforderungen

Anforderungen erhalten als Attribute

- zu welcher Softwareversion aufgenommen
- in welcher Version umgesetzt

Zur einfachen Betrachtung des Standes der Umsetzung wird eine Marke eingefügt oberhalb derer alle Anforderungen umgesetzt wurden.

1.9.2 Versionen

Versionen erhalten als Attribute

- Zeitpunkt der Ablage auf dem SqueakSource-Server
- durchgeführte Qualitätssicherungsmaßnahmen und Metriken (siehe 1.7.7)
- enthaltene neue Anforderungen und Bugfixes

1.9.3 Fehler

Fehler (BUGS) erhalten als Attribute

- zu welcher Version festgestellt
- in welcher Version behoben

Wie bei den Anforderungen haben wir auch bei den Fehlern eine Marke eingefügt oberhalb derer alle Fehler behoben wurden.

1.9.4 PDF erstellen

Um ein PDF-Dokument zu erstellen, kopiert man die XML-Quellen im Klassenkommentar von `AnalogClockProjectManual` in eine Datei. Wir gehen davon aus, die Datei heiße `AnalogClockProjectManual.xml`. Weiterhin benötigen wir DocBook-Stylesheets, sie seien - auf einer Linux-Maschine - unter /usr/share/xml/docbook/stylesheet/nwalsh/1.75.2 abgelegt.

Dann führen die nächsten beiden Schritte zum PDF-Dokument:

- `xsltproc --xinclude --output AnalogClockProjectManual.fo -stringparam index.on.type 1 /usr/share/xml/docbook/stylesheet/nwalsh/1.75.2/fo/docbook.xsl AnalogClockProjectMa`
- `fop AnalogClockProjectManual.fo AnalogClockProjectManual.pdf.`

Wer eine einfache Option zum Editieren und Validieren von DocBook-Dokumenten sucht, kann `emacs` den `nXML-mode` beibringen. Die Quellen dazu finden sich auf docbook-5-support.

Literaturverzeichnis

[1] Allen-Conn, B.; Rose, Kim: Fundamentale Ideen im Unterricht. Mit Squeak Mathematik und Naturwissenschaften verstehen. Viewpoints Research Institute, 2003; dt. Übersetzung 2008

[2] Alpert, Sherman; Brown, Kyle; Woolf, Bobby: The Design Patterns Smalltalk Companion, Addison-Wesley, 1998

[3] Arnoldi, Massimo; Beck, Kent; Bieri, Markus; Lange, Manfred: A Pattern Language for Values That Change. 1999.

[4] Astels, David R.: sSpec. Astels, 2006

[5] Aytar, Onur: A Guide to Work with Squeak Morph Classes. Northeastern University, 2002

[6] Beck, Kent: Simple Smalltalk Testing: With Patterns. Paper

[7] Beck, Kent: Smalltalk Best Practice Patterns. Prentice Hall, 1997

[8] Beck, Kent: Kent Beck's Guide to better Smalltalk. ??, ??

[9] Bellin, David; Simone, Susan Suchman: The CRC Card Book. Addison Wesley, 1997

[10] Bergel, Alexandre et al.: Smalltalk Exercises. 2005.

[11] Bergel, Alexandre; Denker, Marcus; Ducasse, Stéphane: Squeak Tools

[12] Bergel, Alexandre; Ducasse, Stephane; Putney, Colin; Wuyts, Roel: The Omnibrowser Reference.

[13] Black, Andrew P.: Squeak Smalltalk - Quick Reference. 2001

[14] Black, Andrew P.: Squeak Smalltalk - Language Reference. 2001 .

[15] Black, Andrew P.: Squeak, an Open Source Smalltalk for the 21st century. ECOOP 2001.

[16] Black, Andrew; Ducasse, Stéphane; Nierstrasz, Oscar; Pollet, Damien: Squeak by Example. Version of 2008-03-10

[17] Brauer, Johannes: Grundkurs Smalltalk - Objektorientierung von Anfang an. Eine Einführung in die Programmierung. 3. Auflage. Vieweg + Teubner, 2009

[18] Briffaut, Xavier; Ducasse, Stéphane: Squeak Programmation. Eyrolles, 2001

[19] Bühler, Thomas: MooseGager, a Software Metrics Tool based on Moose. 2003, Institut für Informatik und angewandte Mathematik, Universität Bern.

[20] Burkert, Chris: Smalltalk FAQ.

[21] Cassou, Damien: SmallWiki 2: Design overview.

[22] Conway, Matthew J.: Alice: Easy-to-Learn 3D Scripting for Novices. Dissertation, University of Virginia, 1997

[23] Dann, Wanda P.; Cooper, Stephen; Pausch, Randy: Learning to Program with Alice. Prentice Hall, 2006

[24] Demeyer, Serge; Ducasse, Stéphane: Metrics, Do they really help ? LMO 1999.

[25] Demeyer, Serge; Ducasse, Stéphane, Nierstrasz, Oscar: Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2003

[26] Denker, Marcus; Ducasse, Stéphane: Software Evolution from the Field: An Experience Report from the Squeak Maintainers.

[27] Denker, Marcus: Metaprogramming and Reflection. Refactoring. WS 2006/2007

[28] Ducasse, Stéphane: OO Design with Smalltalk. A pure object-oriented language and environment.

[29] Ducasse, Stéphane; Lanza, Michele: Towards a Methodology for the Understanding of Object-Oriented Systems. Technique et sciences informatiques. Volume X - n° X/2001, pages 1 à Y.

[30] Ducasse, Stéphane; Besset Didier: Learning Programming Concepts with Squeak. Draft, 2001

[31] Ducasse, Stéphane: From Logo to OO: Learning how to program in Squeak. Draft, 2002

[32] Ducasse, Stéphane: Squeak. Learn Programming with Robots. Apress, 2005

[33] Ducasse, Stéphane: Brainstorming Draft, 2006.

[34] Ducasse, Stéphane: The Squeak Object Model

[35] Ducasse, Stéphane; Lienhard, Adrian; Renggli Lukas: Seaside: A Flexible Environment for Building Dynamic Web Applications. In: IEEE Software, September/October 2007, S. 56-63

[36] Feathers, Michael C.: Working Effectively with Legacy Code. Prentice Hall, 2005

[37] Fowler, Martin: Refactoring. Improving the Design of Existing Code. Addison-Wesley, 2000

[38] Fowler, Martin: Is Design dead? In: 080817 Design http://www.martinfowler.com/articles/designDead.html, 2004

[39] Gallenbacher, Jens: Abenteuer Informatik. IT zum Anfassen von Routenplaner bis Online-Banking. Elsevier, 2007

[40] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[41] Gälli, Markus; Denker, Marcus: Von kleinen und großen Erfindern. Squeak: Lernumgebung und Smalltalk-System für Kinder und Erwachsene. In: c't, Heft 7, S. 216-221, 2004

[42] Gälli, Markus; Nierstrasz, Oscar; Ducasse, Stéphane: One-Method Commands: Linking Methods and Their Tests. Für: OOPSLA, 2004

[43] Goldberg, Adele; Robson, David: Smalltalk-80. The Language and its Implementation. Addison-Wesley, 1983

[44] Gómez Deck, Diego et al.: SQUEAK: un mundo para aprender. ISBN: 84-932888-9-6

[45] Gómez Deck, Diego: Programando con Smalltalk.

[46] Goos, Gerhard; Zimmermann, Wolf: Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen. Springer, 2006

[47] Greenberg, Andrew C.: Extending the Squeak Virutal Machine.

[48] Greevy, Orla: Enriching Reverse Engineering with Feature Analysis. Dissertation SCG des IAM, Uni Bern, 2007

[49] Guzdial, Mark: Squeak: Object-Oriented Design with Multimedia Applications. Draft as of December 8, 1999. College of Computing. Georgia Institute of Technology.

[50] Herbert, Charles: An Introduction to Programming using Alice. Thompson Course Technology, 2007

[51] Ingalls, Dan; Kaehler, Ted; Maloney, John; Wallace, Scott; Kay, Alan: Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself. In Proceedings OOPSLA 1997, ACM SIGPLAN Notices, pages 318-326. ACM Press, November 1997.

[52] Johnson, Ralph: CS497REJ objektorientierte Programmierung und objektorientierter Entwurf. Deutsche Übersetzung: Georg Heeg.

[53] Karwal, Arpan; Moloney, Cecilia: Squeak and Children: Programming their way to learning new concepts in Math and Science. 2006

[54] Kay, Allan: Children Learning by Doing. Squeak Etoys on the OLPC XO. Rough Draft. Viewpoints Research Institute, April 2007

[55] Langr, Jeff: Essential Java Style: Patterns for Implementation

[56] Maloney, John: An Introduction to Morphic: The Squeak User Interface Framework. Walt Disney Imagineering

[57] Mens, Kim; González, Sebastián; Ordólez, Diego: Practical Session Notes. In: Software Engineering: Measures and Maintenance INGI2252 (2005-2006)

[58] Meyer, Bertrand: Object Oriented Software Construction. Prentice Hall, 1997.

[59] Müller, Frank: Mehr als ein Spiel. Croquet: Verteilte Benutzerwelten mit Smalltalk. In: iX 10/2005, S. 136-139

[60] Müller, Frank: Partytime. Smalltalk - Zum aktuellen Stand der ersten objektorientierten Programmiersprache. In: iX 11/200, S. 116-120

[61] Pierce, Jeff: Alice in a Squeak Wonderland.

[62] Prasse, Michael: Objektorientierte Software-Entwicklung mit Smalltalk. Tomcat Computer GmbH, 171 Seiten

[63] Rappin, Noel: Squeak for Non-Native Speakers

[64] Reichhart, Stefan: Rule-based Assessment of Test Quality. In: Journal of Object Technology, Special Issue TOOLS Europe 2007, 2007.

[65] Reichhart, Stefan: Assessing Test Quality. Masterarbeit am Institut für Informatik und angewandte Mathematik, Bern 2007

[66] Reißing, Ralf: Bewertung der Qualität objektorientierter Entwürfe. Fakultät Informatik der Universität Stuttgart, 2002.

[67] Renggli, Lukas: Magritte. Meta-Described Web Application Development. Masterarbeit Universität Bern, 2006.

[68] Renggli, Lukas: Pier - The Meta-Described Content Management System. Software Composition Group, University of Bern, Switzerland. 2007

[69] Reenskaug, Trygve: BabySRE. Squeak Reverse Engineering.

[70] Sharp, Alec: Smalltalk by Example, 1997

[71] Steiner, Max: Proseminar Programmiersprachen: Smalltalk. März 2005

[72] Tomek, Ivan: Introduction to VisualWorks Smalltalk. Cincom, 2003

[73] Tuchel, Joachim: Lebendes Objekt. Smalltalk: ein aktueller Klassiker. In: c't, 2003, Heft 2, S. 188-193

[74] Vuletich, Juan: Morphic 3: The Future of GUIs. Vortrag auf "Smalltalks 2007" in Buenos Aires, Argentinien

[75] Walenta, Kathrin: Smalltalk. Seminar Programmiersprachen.

Tabellenverzeichnis

Abbildungsverzeichnis

Inhaltsverzeichnis