Orion, ein Windowmanager in Scheme

David Frese

10. April 2003

Zusammenfassung

Orion ist ein X-Windows-Windowmanager der erstmals die beiden Grundkonzepte der Fensterverwaltung „Seiten und Kacheln“ und „Stapeln und Verschieben“ verbindet. In dieser Arbeit wird gezeigt, wie die Verbindung dieser beiden Konzepte möglich ist. Danach wird die Implementierung von Orion in der Programmiersprache Scheme vorgestellt.

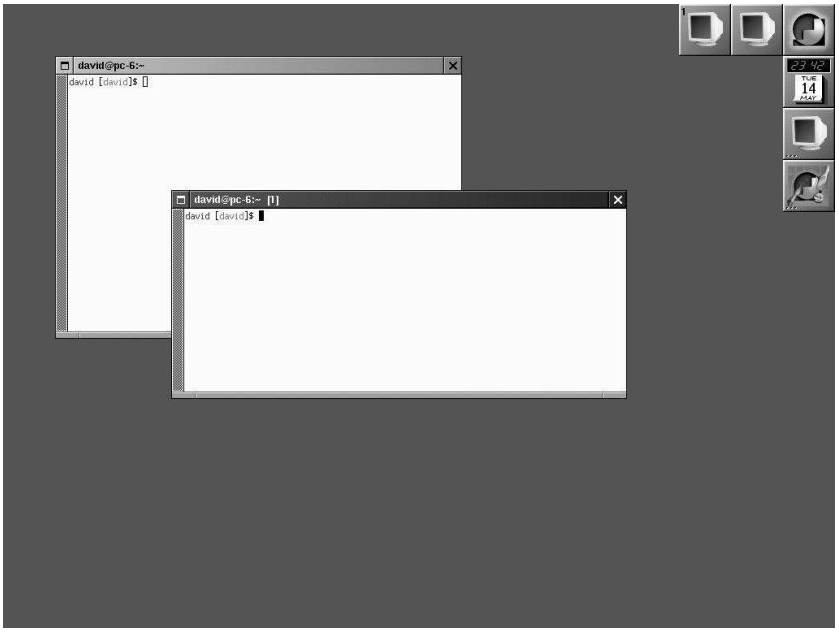

Abbildung 1: WindowMaker Screenshot

1 Einleitung

Orion vereinigt zwei Grundkonzepte des Window-Managements. „Stapeln und Verschieben“ und „Seiten und Kacheln“.

Stapeln und Verschieben Vielleicht durch das Fenstersystem von X-Windows inspiriert, ist der allergrößte Teil der Windowmanager nach dem Konzept „Stapeln und Verschieben“ aufgebaut. Der Benutzer kann die Fenster dabei an beliebiger Position auf dem Bildschirm platzieren, und deren Größe frei verändern. Da sich die Fenster dabei überlappen können, ist zudem noch eine Ordnung der Fenster nötig, die für ein Fenster bestimmt, „über“ welchen anderen Fenstern es liegt. Eine einfache Implementierung dieses Systems ist der Windowmanager WindowMaker [Koj97] (siehe Abbildung 1).

Seiten und Kacheln Das andere Konzept „Seiten und Kacheln“ beruht darauf, daß Fenster sich nicht überlappen. Der Benutzer teilt den Bildschirm in mehrere Kacheln auf, die den Bildschirm ganz abdecken. Jede der Kacheln kann dann mehrere Programmfenster enthalten, von denen der Windowmanager jeweils eines anzeigt. Jedes Programmfenster füllt außerdem immer die gesamte Kachel aus, wodurch die Programmfenster in einer Kachel wie ein Stapel Papier wirken. Implementierungen dieses Konzepts sind zum Beispiel der ursprüngliche Windowmanager des „Project Oberon“ von Professor Wirth [WG92] und „Ion“ von Tuomo Valkonen [Val99] (siehe Abbildung 2).

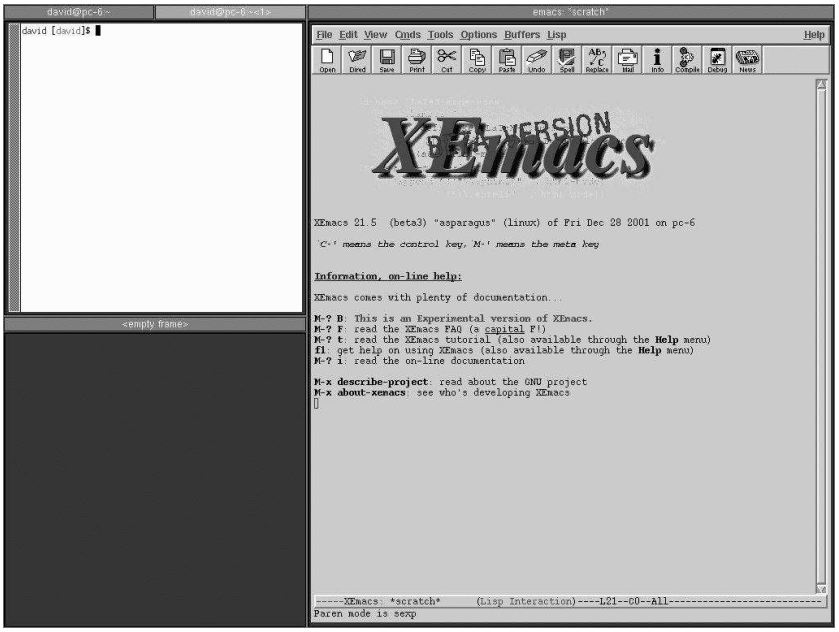

Abbildung 2: Ion Screenshot

Das Konzept von Orion Beide Konzepte des Window-Mangements haben ihre Vor- und Nachteile. Beim Konzept „Stapeln und Verschieben“ kann der Benutzer solche Programme leichter bedienen, die aus mehreren Fenstern bestehen, oder verschieden große Dialogfenster verwenden. Außerdem gibt es Programme, die Fenster einer bestimmten Größe benötigen, was bei diesem Konzept ohne weiteres möglich ist. Das Konzept „Seiten und Kacheln“ hat den Vorteil, daß durch die klare Gliederung, der Benutzer sehr einfach und schnell damit arbeiten kann, vor allem wenn es, wie zum Beispiel bei Text-basierten Programmen, nicht auf eine exakte Größe der Fenster ankommt. Ein weiterer Vorteil ist die unkomplizierte Navigation mit der Tastatur, die nur aus dem Auswählen der Kachel (zum Beispiel mit den Pfeiltasten) und dem Durchblättern der Seiten besteht.

Beide Konzepte sind also in unterschiedlichen Situationen sinnvoll. Deshalb ist die grundlegende Idee hinter Orion, dem Benutzer beide Konzepte zur Verfügung zu stellen, und zwar nicht als sich gegenseitig ausschließende Alternativen, sondern parallel. Der Benutzer kann bei Orion den Bildschirm, genauso wie bei Ion, in Kacheln aufteilen. Jede Kachel erhält dann durch den Benutzer einen eigenen, internen Windowmanager zugewiesen, der den Inhalt der Kachel nach einem der beiden Konzepte verwaltet (siehe Abbildung 3). Der interne Windowmanager, der nach dem Konzept des „Stapeln und Verschieben“ funktioniert, wird im folgenden als *Move-wm*, der „Seiten und Kacheln“-Windowmanager als *Switch-wm* bezeichnet.

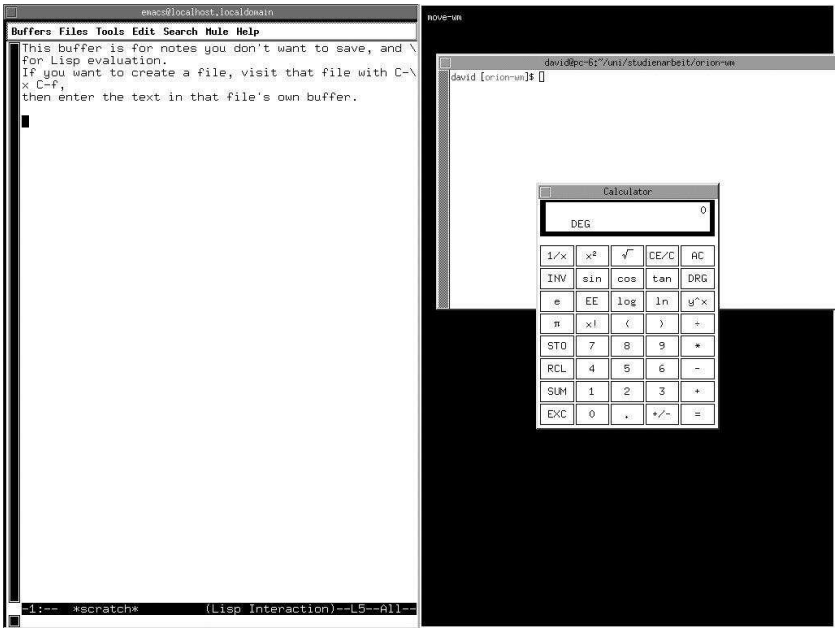

Abbildung 3: Orion Screenshot

2 Grundlagen des Windowmanagements

Dieser Abschnitt gibt einen kurzen Überblick über das X-Window-System und die Xlib-Bibliothek. Desweiteren wird der Begriff des Windowmanagers erklärt, sowie die Aufgaben eines Windowmanagers anhand des *Inter-Client Communications Conventions Manual* [Ros88] dargestellt.

2.1 X-Windows

Das X-Window-System wurde Mitte der 80er Jahre am Massachusetts Institute of Technology, mit Unterstützung der Digital Equipement Corporation, entwickelt, um die Darstellung grafischer Informationen auf UNIX-Systemem zu ermöglichen. Dazu wurde ein Client-Server-Modell entwickelt, bei dem der Client bestimmt, was darzustellen ist und der Server dafür zuständig ist, wie es darzustellen ist. Das Protokoll, über das Client und Server kommunizieren, ist das X-Protokoll [GKM90]. Die meistverbreitetste Implementierung des Servers ist das XFree86-Paket, welches in fast allen Unix-Distributionen enthalten ist. Für die Implementierung des Protokolls auf der Seite des Clients exisitiert die *Xlib*, programmiert in der Sprache C. Diese Bibliothek wird seit 1985 von „The Open Group“ entwickelt und es gibt diverse Dokumentationen darüber, zum Beispiel [Nye90].

2.2 Programmieren unter X

Der Kern einer X-Client-Anwendung besteht aus dem Display-Objekt. Es ist die Repräsentation des Kommunikations-Kanals mit dem X-Server und wird daher für fast alle Funktionsaufrufe der Xlib benötigt. Ein

Programm muß sich zuerst bei einem X-Server mit einem Aufruf der Funktion `XOpenDisplay` anmelden. Danach kann der Client den Server anweisen, bestimmte Ressourcen für ihn anzulegen, von denen manche auf dem Bildschirm sichtbar werden können, zum Beispiel Fenster, und manche unsichtbar bleiben, wie zum Beispiel Farbtabellen.

Die Idee hinter dem Fenstersystem von X besteht darin, daß ein Fenster (ein Rechteck aus Bildpunkten) eine beliebige Anzahl von Unterfenstern, *Child-Windows*, enthalten kann. Diese Child-Windows überdecken einen Teil des Fensters, können aber nicht darüber hinausragen. Sie können sich aber mit anderen Child-Windows desselben Fensters überlappen. Daher existiert eine bestimmte Reihenfolge unter den Child-Windows, die *Stacking-Order*, die angibt, welches Child-Window den Inhalt eines anderen überdeckt. Der X-Server stellt außerdem ein *Root-Window* zur Verfügung, dessen Inhalt dem Bildschirm entspricht, wie ihn der Benutzer sieht. Ein Programm, das grafische Inhalte anzeigen will, erstellt ein Child-Window dieses Root-Windows und kann so einen Teil des Bildschirms für seine Ausgabe nutzen.

X-Windows benutzt für die Aktualisierung des Inhalts dieser Fenster das *Damage-Control-Modell*. Das bedeutet, daß der X-Server dem Erzeuger eines Fensters, im folgenden auch der *Besitzer* genannt, eine Nachricht schickt, sobald der Besitzer einen Teil dieses Fenster neu mit Inhalt füllen muß (eine so genannte *Expire*-Nachricht). Der Inhalt muß zum Beispiel neu erzeugt werden, wenn das Fenster bislang von einem anderen Fenster überdeckt war, und jetzt sichtbar geworden ist.

Ein vollständiges Graphical-User-Interface besteht nicht nur aus der Ausgabe eines Programms, sondern auch aus den Eingaben des Benutzers. Dazu verwaltet der X-Server eine Maus, auch Pointer gennant, und die Tastatur. Mausereignisse, wie Bewegungen und Klicks, sendet der X-Server an den Besitzer des Fensters, in dem sich die Maus gerade befindet. Tastaturereignisse werden in der Regel an den Besitzer des Fensters mit dem *Input-Focus* gesendet.

Da der Benutzer nicht nur ein Programm mit grafischer Ausgabe gleichzeitig verwenden will, muß er den Input-Focus und die Sichtbarkeit der Fenster verändern können. Dies ermöglicht dem Benutzer der *Windowmanager*.

2.3 Windowmanager

Windowmanager sind im X-Windows-System als einfache Clients implementiert. Was den Windowmanager gegenüber dem X-Server von anderen Clients unterscheidet ist, daß er den X-Server anweist, ihm die Nachrichten der sogenannten *Substructure-Redirect-Gruppe* für das Root-Window zu schicken. Diese Gruppe besteht aus Nachrichten der Typen *Map-Request*, *Configure-Request* und *Circulate-Request*. Nachrichten des Typs Map-Request betreffen das Sichtbarmachen, *Mapping* genannt, von Unterfenstern des Root-Windows. Die Idee dahinter ist, daß wie bereits erwähnt, ein Programm ein Fenster stets als Unterfenster des Root-Windows erzeugt. Danach weist das Programm den X-Server an, dieses Fenster sichtbar zu machen. Statt dies zu tun, schickt der X-Server nun aber eine Map-

Request-Nachricht an den Windowmanager und ermöglicht es dem Windowmanager so zum Beispiel, das Fenster erst an eine andere Stelle in der Fensterhierarchie zu hängen und es erst dann sichtbar zu machen. Der Windowmanager fängt also, mit Hilfe des X-Servers, das Sichtbarmachen eines Programmfensters ab. Genauso verhält es sich mit den Nachrichten des Typs Configure-Request, die die Position und Größe des Programmfensters bestreffen und den Nachrichten des Typs Circulate-Request, die eine Veränderung der Stacking-Order der Fenster bestreffen. Der X-Server überlässt also jeweils die Ausführung dieser Änderungen dem Windowmanager.

Darüber hinaus gibt es aber noch eine Reihe anderer Aufgaben, die ein Windowmanager erledigen muß. Diese Aufgaben und Konventionen, die sowohl andere Programme, als auch der Windowmanager einhalten müssen, beschreibt das „Inter-Client Communication Conventions Manual“ [Ros88]. Die wichtigsten Punkte sind:

- Der Windowmanager kann ein Programmfenster, ein so genanntes *Top-Level-Window*, an eine beliebige Stelle in der Fensterhierarchie hängen.
- Der Windowmanager kann die Größe eines Programmfensters frei verändern.
- Der Windowmanager verwaltet den Input-Focus, das heißt, er bestimmt, in der Regel als Reaktion auf eine Aktion des Benutzers, welches Programm die Tastaturereignisse erhält.
- Der Windowmanager stellt sicher, daß die Farben des aktiven Programmfensters richtig dargestellt werden, indem er die Farbtabellen dieses Fensters installiert.
- Der Windowmanager informiert ein Programm darüber, daß der Benutzer ein Programmfenster schließen möchte.
- Der Windowmanager sorgt für die richtige Darstellung von Dialogfenster, so genannte *Transients*. Der Windowmanager muß dazu ein Transient immer anstelle des Hauptfensters aktivieren und anzeigen.

3 Xlib-Anbindung

Orion baut auf der Scheme-Implementierung Scsh auf. Diese bietet eine Anbindung an die Xlib namens *Scx*. In diesem Abschnitt werden Scsh und Scx kurz vorgestellt.

3.1 Scsh

Das Projekt *the SCheme SHell* [Shi94] wurde 1994 von Olin Shivers begonnen und ist eine Scheme-Implementierung basierend auf *Scheme48* [s48]. Die Erweiterungen bestehen aus Bibliotheken für Unix-Systemaufrufe, zum Beispiel für I/O, Dateisystem und Signal-Behandlung.

3.2 Scx

Um einen Windowmanager für X-Windows zu programmieren, ist ein Zugang zur Kommunkation mit dem X-Server nötig. Die C-Bibliothek Xlib bietet zu diesem Zweck Datenstrukturen und Funktionen an. Ausgangspunkt für die Programmierung eines Windowmanagers in Scheme ist daher eine Anbindung an diese Bibliothek. Eine Anbindung an die Scheme-Implemenierung Scsh, namens Scx, haben Norbert Freudemann und der Autor im zweiten Halbjahr 2001 programmiert.

Neben der reinen Kapselung der Xlib-Funktionen in Scheme-Funktionen, bietet Scx bereits weitere nützliche Funktionalität. So kapselt Scx X-Ressourcen wie Fenster, Schriften und Farbtabellen in Scheme-Datenstrukturen und, sofern das Programm diese Ressourcen auch erzeugt hat, gibt Scx sie automatisch wieder frei. Außerdem sind für das Warten auf Ereignisse besondere Funktionen implementiert. Diese blockieren nicht den gesamten Interpreter, sondern nur den aktuellen Thread. Dadurch ist es mit Scx weiterhin möglich, nebenläufig zu programmieren.

3.3 Erweiterung von Scx

Zum Programmieren eines größeren Programms wie Orion waren noch weitere Abstraktionen von den einfachen Xlib Funktionen nötig. Dazu gehört erstens die Ereignisbehandlung und zweitens das Abfangen von Tastatureingaben zur Bedienung des Windowmanagers.

3.3.1 Ereignisbehandlung

Die von der Xlib, beziehungsweise Scx, bereitgestellte Schnittstelle zur Ereignisbehandlung besteht aus der Funktion `display-select-input` zur Auswahl bestimmter Arten von Ereignissen für ein Fenster und der Funktion `wait-event` zum Abrufen des nächsten Ereignisses:

```
(display-select-input window event-mask)
(wait-event display) -> event
```

Der X-Server speichert für jedes Fenster eine Ereignismaske, die bestimmt, welche Ereignisse des Fensters er an das Programm schickt. `Display-select-input` setzt diese Maske neu.

Wenn für ein Fenster ein Ereignis eintritt, für das sich das Programm interessiert, dann schickt der X-Server dieses Ereignis an das Programm. Mit der Funktion `wait-event` muß das Programm alle Ereignisse zentral abholen, was auch die globale Behandlung aller Ereignisse in einer einzigen Schleife nahelegt. Das erschwert allerdings eine übersichtliche Programmierung. Daher wurde für Orion ein Modell mit *Callbacks* implementiert, das beide Schritte der Ereignisbehandlung, Auswahl und Abholen, verbindet. Die Schnittstelle besteht aus drei Funktionen:

```
(request-events! handler window mask)
(stop-events! handler window mask)
(start-event-loop dpy)
```

`request-events!` registriert die Callback-Funktion `handler` zur Behandlung aller Ereignisse für das Fenster `window`, die durch `mask` spezifiziert sind. Diese *Requests* werden global gespeichert. Wenn mehrere Requests für das selbe Fenster registriert sind, vereinigt `request-events!` die Ereignismasken, und ruft `display-select-input` mit dieser gemeinsamen Ereignismaske auf. `stop-events!` meldet ensprechend Callback-Funktionen für bestimmte Ereignisse wieder ab.

Die Funktion `start-event-loop` ist für das Abholen und Verteilen der Ereignisse zuständig. Sie holt das nächste Ereignis vom X-Server ab und ruft dann alle Callback-Funktionen auf, die sich für dieses Ereignis registriert haben. `start-event-loop` wiederholt dies, bis einer der Handler signalisiert Orion zu beenden indem er `#f` zurückgibt.

3.3.2 Tastaturereignisse

Der X-Server schickt normalerweise alle Tastaturereignisse an das Fenster, das den Eingabe-Focus hat. Programme haben aber die Möglichkeit, bestimmte Tasteneingaben abzufangen. Dieses sogenannte *Grabbing* funktioniert so, daß ein Programm über die Funktion `grab-key` eine Tastenkombination und ein Fenster beim X-Server registriert. Wenn nun dieses Fenster oder eines der Fenster, die in der Fensterhierarchie unterhalb von ihm sind, den Eingabe-Focus hat, und diese Tastenkombination vom Benutzer gedrückt wird, schickt der X-Server ein entsprechendes Tastaturereignis an das angegebene Fenster. Der Windowmanager kann also mit einem Aufruf von `grab-key`, mit dem Root-Window als Parameter, bestimmte Tastenkombinationen für sich reservieren.

Für Orion ist es aber nötig, Tastaturereignisse nicht von allen Fenstern, sondern nur von den Fenstern anderer Programme abzufangen und dann innerhalb von Orion, an das in der Hierarchie am weistesten unten liegende Fenster weiterzugeben. Dazu verarbeitet Orion Tastaturereignisse in einer globalen Ereignisbehandlungsroutine. Zum Registrieren einer Tastatenkombination gibt es die Funktion `register-action!`:

```
(register-action! window binding thunk . override-subwindows?)
```

`register-action!` fängt also für das Fenster `window` die Tastenkombination `binding` ab und ruft die Funktion `thunk` ohne Argumente auf, sobald diese Tastenkombination vom Benutzer eingegeben wurde. Der optionale Paramter `override-subwindows?` gibt dabei an, ob dieses Fenster vor untergeordneten Fenstern von Orion, die dieselbe Tastenkombination registrieren, Vorrang hat.

4 Abstrakter Windowmanager

Für die Implementierung der beiden Windowmanager von Orion, wurde zunächst ein abstrakter Windowmanager programmiert. Dieser beinhaltet zum Beispiel Code für die Verwaltung einer Liste der Fenster, die ein Windowmanager verwaltet oder die Behandlung der für einen Windowmanager wichtigen Ereignisse.

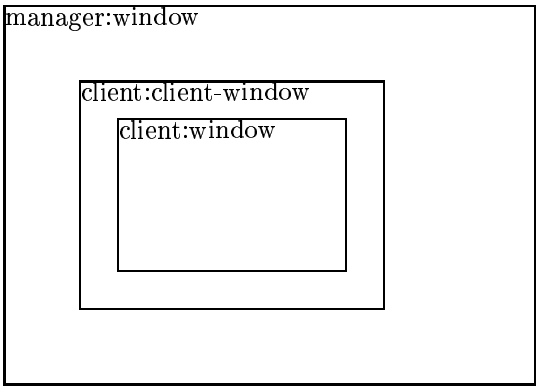

Abbildung 4: Bezeichnung der Fenster eines Windowmanagers

4.1 make-wm-constructor

Die Abstraktion geschieht über eine Funktion namens `make-wm-constructor`, die den Konstruktor für einen konkreten Windowmanager zurückgibt. Die relativ lange Definition von `make-wm-constructor` soll im folgenden in mehreren Schritten erklärt werden.

Die Parameter dieser Funktion sind allesamt Funktionen, die die besonderen Eigenschaften der konkreten Windowmanager ausmachen:

```
(define (make-wm-constructor init-manager deinit-manager
                             init-client deinit-client
                             fit-windows
                             draw-main-window
                             focus-changed
                             fit-client
                             fit-client-window
                             draw-client-window
                             update-client-state)
```

`init-manager` und `deinit-manager` dienen zur Erzeugung und Zerstörung von zusätzlichen Fenstern eines Windowmanagers und anderer spezifischer Daten. `init-client` soll einen neuen Client des Windowmanagers initialisieren, und `deinit-client` eventuell erzeugte Fenster wieder zerstören. In `fit-windows` kann der Windowmanager seine eigenen Fenster wieder an die Größe des Windowmanager-Fensters anpassen. `draw-main-window` und `draw-client-window` sollen dazu dienen, den Inhalt des Windowmanagerfensters, beziehungsweise eines Fensters, das ein Programmfenster enthält, neu zu zeichnen. Die verschiedenen Fenster und deren Bezeichnung sind in Abbildung 4 skizziert. Wenn das Windowmanagerfenster den Eingabe-Fokus erhält oder verliert, ruft der Code `focus-changed` auf. `fit-client` und `fit-client-window` dienen jeweils zur Anpassung der Größe der Fenster, nachdem sich die Größe des jeweils anderen geändert hat. Die letzte Funktion, `update-client-state`, wird aufgerufen, wenn sich der Name eines Programmfensters ändert oder es den Eingabe-Fokus erhält oder verliert.

Die Konstruktor-Funktion für einen Windowmanager hat die Parameter `parent`, ein Fenster, `options`, eine Liste von Konfigurationsoptionen, und eine beliebige Anzahl von Fenstern, die der Windowmanager anfänglich verwalten soll. Zuerst erzeugt der Konstruktor das Hauptfenster des Windowmanagers als Unterfenster von `parent`, mit derselben Höhe und Breite wie `parent` und einem schwarzen Hintergrund:

```
(lambda (parent options . children)
  (let* ((main-window (create-simple-window
                       parent
                       (window-width parent)
                       (window-height parent)
                       (make-set-window-attribute-alist
                        (background-pixel
                         (black-pixel (window-display parent))))))
```

Danach erzeugt der Code eine Datenstruktur zur Repräsentation des Windowmanagers. Diese Datenstruktur enthält das soeben erstellte Fenster, eine Liste der Clients des Windowmanagers, den aktuell aktiven Client (mit `#f` initialisiert) und ein Feld, in dem der Windowmanager spezifische Daten speichern kann (ebenfalls mit `#f` initialisiert). Die weiteren Felder der Datenstruktur bestehen aus den Funktionen, die an anderer Stelle des abstrakten Windowmanagers noch benötigt werden.

```
         (wm (make-wm-data main-window '() #f
                           #f
                           deinit-manager
                           init-client deinit-client
                           fit-client
                           fit-client-window
                           draw-client-window
                           update-client-state)))
```

Jetzt wird die Initialisierungsfunktion des Windowmanagers aufgerufen. In dieser kann der Windowmanager zum Beispiel noch weitere Fenster erzeugen oder das Feld für die privaten Daten initialisieren.

```
    (init-manager wm options)
```

Die Ereignisbehandlungsroutine für das Windowmanager-Fenster ruft im wesentlichen nur die entsprechenden Funktionen des konkreten Windowmanagers auf:

```
    (let ((event-handler
           (lambda (event)
             (let ((type (any-event-type event)))
               (cond
                ((eq? (event-type configure-notify) type)
                 (fit-windows wm))
                ((eq? (event-type expose) type)
                 (draw-main-window wm))
                ((eq? (event-type focus-in) type)
                 (if (window-focused? main-window)
```

```
                    (focus-changed wm)))
                ((eq? (event-type focus-out) type)
                 (if (not (window-contains-focus? main-window))
                    (focus-changed wm)))
                ((eq? (event-type enter-notify) type)
                 (if (not (window-contains-focus? main-window))
                    (focus-window main-window))))))))
      (request-events! event-handler main-window
                       (event-mask structure-notify
                                   enter-window
                                   focus-change
                                   exposure)))
```

Der Aufruf der in Abschnitt 3.3.1 beschriebene Funktion `request-events!` registriert diese Ereignisbehandlungsroutine für das Fenster `main-window` und die angegebenen Ereignis-Typen.

Zuletzt gibt der Code noch die übergebenen Fenster an die Funktion `wm-manage-window` weiter, die diese Fenster dann entsprechend in den Windowmanager integriert.

```
    (for-each (lambda (window)
                (wm-manage-window wm window))
              children)
    wm)))
```

Die bereits erwähnte Datenstruktur zur Speicherung der spezifischen Daten eines Windowmanagers sieht folgendermaßen aus:

```
(define-record-type wm-data :wm-data
  (make-wm-data window clients current-client
                private-data
                deinit-manager
                init-client deinit-client
                fit-client
                fit-client-window
                draw-client-window
                update-client-state)
  wm?
  (window wm:window)
  (clients wm:clients set-wm:clients!)
  (current-client wm:current-client set-wm:current-client!)
  (private-data wm:private-data set-wm:private-data!)
  (deinit-manager wm:deinit-manager)
  (init-client wm:init-client)
  (deinit-client wm:deinit-client)
  (fit-client wm:fit-client)
  (fit-client-window wm:fit-client-window)
  (draw-client-window wm:draw-client-window)
  (update-client-state wm:update-client-state))
```

Zum Beenden eines Windowmanagers gibt es ebenfalls eine allgemeine Funktion:

```
(define (destroy-wm wm)
  ((wm:deinit-manager wm) wm)
  (destroy-window (wm:window wm)))
```

4.2 Client

Der abstrakte Windowmanager implementiert auch die Verwaltung der Clients auf abstrakter Ebene. Dazu gehören die beiden Funktionen `wm-manage-window` und `wm-unmanage-window`.

`wm-manage-window` startet die Verwaltung eines Fensters durch den Windowmanager. Dazu ruft der Code zunächst `create-client` auf, um eine Repräsentation dieses Clients zu erzeugen (Details von `create-client` folgen). Danach fügt der Code dieses Objekt in die Liste der Clients ein und `init-client` führt die konkrete Initialisierung durch. Die letzten Schritte sind dann noch, das Fenster mit `fit-client` auf die richtige Größe zu bringen und mit einem Aufruf von `map-window` das Fenster sichtbar zu machen:

```
(define (wm-manage-window wm window . maybe-rect)
  (let ((client (create-client wm window)))
    (set-wm:clients! wm (cons client (wm:clients wm)))
    (apply (wm:init-client wm) wm client maybe-rect)
    ((wm:fit-client wm) wm client)
    (map-window window)))
```

Die Funktion `wm-unmanage-window` dient zum Entfernen eines Clients aus einem Windowmanager. Sie wird aufgerufen, wenn der Benutzer ein Fenster in einen anderen Windowmanager übertragen will. Dazu ruft der Code die Funktion `reparent-window` auf, um das Fenster wieder als Unterfenster des Root-Windows einzufügen. Der Code ruft dann die Funktion `wm-deinit-client` auf, die im folgenden beschrieben wird:

```
(define (wm-unmanage-window wm window)
  (let ((client (find (lambda (c) (eq? window (client:window c)))
                      (wm:clients wm))))
    (if client
        (begin
          (reparent-window window (window-root window) 0 0)
          (wm-deinit-client wm client))
        #f)))
```

`wm-deinit-client` entfernt den Client aus der Liste, korrigiert das `current-client` Feld des Windowmanagers, ruft die `deinit-client` Funktion des konkreten Windowmanagers auf und zerstört schließlich das `client-window` des Clients.

```
(define (wm-deinit-client wm client)
  (set-wm:clients! wm (filter (lambda (c) (not (eq? c client)))
                              (wm:clients wm)))
  (if (eq? (wm:current-client wm) client)
      (set-wm:current-client! wm #f))
  ((wm:deinit-client wm) wm client)
  (destroy-window (client:client-window client)))
```

Die Datenstruktur für einen Client enthält das Programmfenster des Clients, das *client-window*, welches dieses Programmfenster enthält und ein Feld in dem der jeweilige Windowmanager spezielle Daten speichern kann:

```
(define-record-type client :client
  (make-client private-data window client-window)
  client?
  (private-data client:private-data set-client:private-data!)
  (window client:window)
  (client-window client:client-window))
```

Die Funktion `create-client` erzeugt eine solche Datenstruktur, das dazugehörige *client-window*, sowie zwei Ereignisbehandlungsroutinen für die beiden Fenster:

```
(define (create-client wm window)
  (let ((fit-client (wm:fit-client wm))
        (fit-client-window (wm:fit-client-window wm))
        (draw-client-window (wm:draw-client-window wm))
        (update-client-state (wm:update-client-state wm)))
    (let* ((client-window
            (create-simple-window
              (wm:window wm)
              (window-width window)
              (window-height window)
              (make-set-window-attribute-alist
               (background-pixel
                (black-pixel (window-display window))))))
           (client (make-client #f window client-window)))
```

Die Funktion `reparent-window` führt das Programmfenster `window` als Unterfenster des `client-window` ein:

```
      (reparent-window window client-window 0 0)
      (letrec
          ((client-window-mask (event-mask exposure
                                           structure-notify
                                           focus-change))
```

Die Ereignisbehandlungsroutine für das `client-window` ruft bei den Ereignissen `expose` (Neuzeichnen), `configure-notify` (Fenstergröße geändert) und `circualte-notify` (Reihenfolge geändert) einfach die entsprechenden Funktionen des Windowmanagers auf:

```
           (event-handler
            (lambda (event)
              (let ((type (any-event-type event)))
                (cond
                 ((eq? (event-type expose) type)
                  (draw-client-window wm client))

                 ((eq? (event-type configure-notify) event)
                  (fit-client wm client))
```

```
((eq? (event-type circulate-notify) type)
 (update-client-state wm client))
```

Erhält das `client-window` den Eingabefokus, so gibt die Behandlungsroutine ihn an das Programmfenster weiter. In jedem Fall wird der Windowmanager darüber informiert, daß sich der Fokus geändert hat:

```
((or (eq? (event-type focus-in) type)
     (eq? (event-type focus-out) type))
 (if (window-focused? client-window)
     (take-focus window))
 (update-client-state wm client))
```

Wenn das `client-window` zerstört worden ist, dann schickt der X-Server eine `destroy-notify` Nachricht. Die Ereignisbehandlungsroutine beendet in diesem Fall die Behandlung der Ereignisse mit einem Aufruf der Funktion `stop-events!`

```
((eq? (event-type destroy-notify) type)
 (stop-events! event-handler client-window
               client-window-mask))

))))

(client-mask (event-mask property-change
                         structure-notify))
```

Die Ereignisbehandlungsroutine für das Programmfenster informiert den Windowmanager über eine Änderung der Größe des Programmfensters und eine Änderung des Programmtitels, der in der `WM_NAME` Eigenschaft des Fensters gespeichert ist:

```
(client-handler
 (lambda (event)
   (let ((type (any-event-type event)))
     (cond
      ((eq? (event-type property-notify) type)
       (let ((name (atom-name (property-event-display event)
                              (property-event-atom event))))
         (cond
          ((equal? "WM_NAME" name)
           (update-client-state wm client))
          )))

      ((eq? (event-type configure-notify) type)
       (fit-client-window wm client))
```

Die Behandlung einer `destroy-notify` Nachricht besteht darin, die Behandlung der Ereignisse zu beenden und die oben beschriebene Funktion `wm-deinit-client` aufzurufen.

```
((eq? (event-type destroy-notify) type)
 (stop-events! client-handler window client-mask)
 (wm-deinit-client wm client))
```

```
            (else #t)))))
        )
      (request-events! event-handler client-window
                       client-window-mask)
      (request-events! client-handler window client-mask))

    client)))
```

5 Switch-wm

Die Implementierung des „Seiten und Kacheln“ Konzepts ist in zwei unabhängige Teile aufgeteilt. Die Kacheln sind durch einen in Abschnitt 7 beschriebenen *Splitter* realisiert. Die Seiten werden durch den *Switch-wm-Windowmanager* implementiert.

Für die Implementierung des *Switch-wm-Windowmanagers* müssen die Eigenheiten dieses Windowmanagers durch die Argumente von `make-wm-constructor` ausgedrückt werden. Im folgenden sollen die wichtigsten Funktionen kurz vorgestellt werden.

Die Funktion `init-manager` dient zur Initialisierung eines neu erstellten Managers. Beim Switch-wm besteht dies aus einer *draw-data* Datenstruktur, die Informationen über Farben und Schriftarten enthält, einer *Key-bindings-Datenstruktur*, die die Tastenkürzel des Windowmanagers enthält, sowie einem Fenster für die Titelleisten der Programme und einer Titelleiste *empty-title-bar* die angezeigt wird, wenn der Windowmanager keine Programmfenster enthält.

```
(define (init-manager wm options)
  (let ((main-window (wm:window wm)))
    (let* ((draw-data (create-draw-data main-window options))
           (key-bindings
            (create-key-bindings (window-display main-window)
                                                 options))
           (title-bars-window (create-title-bars-window wm))
           (data (make-data draw-data key-bindings
                            title-bars-window
                            #f)))

      (set-wm:private-data! wm data)
      (set-data:empty-title-bar! data
                        (create-empty-title-bar wm options))
```

Des weiteren bringt `init-manager` das Fenster für die Titelleisten in die richtige Größe, fügt die *empty-title-bar* hinzu und macht es sichtbar. Zuletzt werden die Tastenkürzel mit den dazugehörigen Aktionen durch die Funktion `register-key-actions` registriert.

```
      (fit-title-bars-window wm)
      (add-title-bar! wm (data:empty-title-bar data))
      (map-window (data:title-bars-window data))

      (register-key-actions wm))))
```

Die Funktion `init-client` initialisiert ein neu erstelltes Client-Objekt. Sie ruft zunächst die Funktion `fit-client-window` auf, die das Fenster auf die maximale Größe, das heißt die Größe des Windowmanagerfensters ohne den Platz für die Titelleisten, bringt. Dann ruft der Code `map-window` auf, um den Client sichtbar zu machen und erzeugt eine Titelleiste für diesen Client. Diese Titelleiste wird im `private-data` Feld des Client-Objekts gespeichert und mit `add-title-bar!` sichtbar gemacht. Zuletzt wählt `switch-to-client` den neuen Client aus.

```
(define (init-client wm client . maybe-rect)
  (fit-client-window wm client)
  (map-window (client:client-window client))
  (let ((tb (create-client-title-bar wm client)))
    (set-client:private-data! client tb)
    (add-title-bar! wm tb)
    (switch-to-client wm client)))
```

`switch-to-client` bringt das Fenster eines Clients in der Reihenfolge der Fenster nach oben und macht es damit sichtbar. Außerdem stellt die Funktion fest, ob andere Clients in diesem Windowmanager Dialogfenster, sogenannte *Transients*, für den zu selektierenden Client sind. Wenn ja, werden diese Clients ausgewählt.

```
(define (switch-to-client wm client)
  (if client
      (begin
        (show-client client)
        (set-wm:current-client! wm client)
        (let ((transients
               (filter (lambda (c)
                         (let ((w (client:window c)))
                           (and (transient-for? w)
                                (eq? (get-transient-for-window w)
                                     (client:window client)))))
                       (wm:clients wm))))
          (if (null? transients)
              (focus-client client)
              (for-each (lambda (c) (switch-to-client wm c))
                        transients))))
      (set-wm:current-client! wm #f)))
```

6 Move-wm

Ein *Move-wm* Windowmanager realisiert das Konzept des „Stapeln und Verschieben“ innerhalb einer Kachel. Er muß es also dem Benutzer ermöglichen die Programmfenster beliebig zu positionieren und in der Größe zu verändern, sowie deren Reihenfolge zu verändern.

Im Gegensatz zum *Switch-wm* benötigt ein *Move-wm* Windowmanager keine zusätzlichen Fenster. Daher ist die `init-manager` Funktion etwas kürzer. Sie erstellt nur spezifische Daten, die zum Zeichnen der Fenster

benötigt werden, und erzeugt und registriert die Tastenkombinationen, die zur Steuerung des Move-wm benutzt werden.

```
(define (init-manager wm options)
  (let* ((main-window (wm:window wm))
         (draw-data (create-draw-data main-window options))
         (key-bindings
          (create-key-bindings (window-display main-window)
                                                options))
         (data (make-data draw-data key-bindings)))

    (set-wm:private-data! wm data)

    (register-key-actions wm)))
```

`init-client` ruft zunächst die Funktion `initial-client-rect` auf, die berechnet, an welche Position und in welcher Größe das *Client-window* dargestellt werden muß. Danach erzeugt `init-client` eine Titelleiste, die sich im Gegensatz zum Switch-wm innerhalb des *client-windows* befindet. Weitere zusätzliche Fenster sind die *Resize-bars*, die sich am Rand des *Client-windows* befinden, und es dem Benutzer ermöglichen, die Größe des Fensters mit der Maus zu verändern.

```
(define (init-client wm client . maybe-rect)
  (let ((r (initial-client-rect wm (client:window client)
                                maybe-rect)))
    (move/resize-window (client:window client) r)
    (let* ((title-bar (create-client-title-bar wm client))
           (resize-bars (create-resize-bars wm client))
           (data (make-client-data client
                                   title-bar
                                   resize-bars)))
      (set-client:private-data! client data)

      (fit-client-windows wm client)

      (map-window (title-bar-window title-bar))
      (for-each map-window resize-bars)))

  (map-window (client:client-window client))
  (select-client wm client))
```

Als letzte Funktion wird hier `select-client` vorgestellt. Das eigentliche Auswählen eines Clients besteht beim Move-wm darin, das entsprechende *Client-window* mit `raise-window` in der Fenster-Hierarchie nach oben zu bringen. `select-client` bringt aber danach auch alle Transients dieses Clients nach oben und stellt damit sicher, daß diese vor ihm zu sehen sind. Als letztes fokussiert der Code das oberste dieser Fenster.

```
(define (select-client wm client)
  (let ((clients (cons client (transient-clients wm client))))
    (for-each (lambda (client)
```

```
            (raise-window (client:client-window client)))
          clients)
    (focus-client (last clients))))
```

7 Splitter

Ein *Splitter* realisiert die Kacheln des „Seiten und Kacheln“-Konzepts. Er bietet die Möglichkeit, an der Stelle eines einzigen Fensters oder Windowmanagers, zwei zu plazieren. Damit ist ein Splitter ebenfalls eine Art Windowmanager, der genau zwei Fenster verwalten kann. Dazu erzeugt ein Splitter zwei Fenster `window-1` und `window-2` die dann jeweils einen Client enthalten können. Die `manage-window`-Funktion des Splitters sieht daher folgendermaßen aus:

```
(define (split-manage-window split window . maybe-title-bar-rect)
  (cond
   ;; use as first client
   ((not (split:client-1 split))
    (reparent-window window (split:window-1 split) 0 0)
    (fit-client split window)
    (set-split:client-1! split window)
    (map-window window))
   ;; use as second client
   ((not (split:client-2 split))
    (reparent-window window (split:window-2 split) 0 0)
    (fit-client split window)
    (set-split:client-2! split window)
    (map-window window))

   (else
    (debug-message 1 "cannot add third window % to splitted window %"
   window (split:main-window split))
    #f)))
```

Eine besondere Programmierung ist beim Erzeugen und Beenden eines Splitters erforderlich, worauf der folgende Abschnitt genauer eingeht.

8 Kontrollinstanz

Einige Aufgaben des Fenstermanangements können die einzelnen Windowmanager nicht, oder nur sehr schwer durchführen, da diese möglichst autonom arbeiten sollten. Es bedarf also eines „Überbaus“ über die hierarchische Struktur, der folgende Aufgaben erfüllen muß:

- Erzeugen einer neuen Windowmanager-Instanz
- Beenden einer Instanz
- Abspeichern und Wiederherstellen der Hierarchie
- Abspeichern und Wiederherstellen der Konfiguration einzelner Instanzen

- Delegieren von neuen Programmfenstern an bestimmte Windowmanager
- Austauschen von Fenstern zwischen verschiedenen Windowmanagern

Die Realisierung dieser Aufgaben wird in den folgenden Unterabschnitten kurz erläutert.

8.1 Erzeugen einer neuen Windowmanager-Instanz

Die meisten Windowmanager für X-Windows Systeme bieten dem Benutzer die Möglichkeit, sogenannte *Workspaces* zu benutzen. Das sind mehrere unabhängige „Seiten“ des Bildschirms, auf denen der Benutzer verschiedene Fenster platzieren kann und zwischen denen er hin- und herschalten kann. Diese Funktionalität läßt sich in Orion einfach durch einen Switch-Windowmanager ohne Titelleisten realisieren. Diese Instanz ist auch die Anfangskonfiguration beim ersten Start von Orion. Darauf aufbauend ist das Erzeugen weiterer Windowmanager für den Benutzer sehr einfach: Mit den Funktionstasten `F5` oder `F6` erzeugt der Benutzer eine neue Instanz eines Switch-Windowmanagers, beziehungsweise Move-Windowmanagers, die die `wm-manage-window` Funktion in den momentan fokussierten Windowmanager einfügt.

Das Erzeugen eines neuen Splitters gestaltet sich ähnlich einfach: Hier muß sich der Benutzer nur zwischen einem vertikalen, und einem horizontalen Splitter entscheiden – `M-s` oder `M-h`. Schwierig ist nur, zu entscheiden welches Fenster geteilt wird. Da es für den Benutzer nur in seltensten Fällen Sinn macht, ein Programmfenster direkt in einem Teilfenster des Splitters zu haben, ersetzt Orion den aktuell ausgewählten Windowmanager durch den Splitter, und platziert diesen Windowmanager in das erste Teilfenster des Splitter.

8.2 Beenden einer Instanz

Problematisch bei der Beendigung einer Windowmanager-Instanz wäre es, einen Windowmanager zu beenden, der noch Programmfenster, oder andere Windowmanager enthält. Daher wird diese Möglichkeit von vorneherein ausgeschlossen. Das Schließen eines Programmfensters und das Beenden eines Windowmanager geschieht über dieselbe Tastenkombination. Ein Windowmanager wird dadurch erst beendet, wenn er keine Unterfenster mehr enthält.

Eine Ausnahme bildet wieder der Splitter: Hier ist es am sinnvollsten, den Splitter zu beenden, sobald eines der beiden Teilfenster leer geworden ist. An seine Stelle tritt dann der Inhalt des anderen Teilfensters.

8.3 Abspeichern und Wiederherstellen der Konfiguration

Die Hierarchie der Windowmanager abzuspeichern gestaltet sich relativ einfach, da sie global verwaltet wird und daher jederzeit bekannt ist. In einer Konfigurationsdatei wird einfach eine Liste mit folgendem Format

abgespeichert:

<manager> → (<type> <options-name> <options> . <manager>*)
<type> → move-wm | switch-wm | split

In dieser Liste ist die Hierarchie als Baumstruktur implizit enthalten.

Die beiden Elemente <options-name> und <options> dienen zum Speichern der Konfiguration der einzelnen Windowmanager-Instanzen. <options> ist eine Assoziationsliste, die bestimmte Bezeichner auf Scheme-Literale abbildet. Diese Liste enthält Optionen wie Farben, Schriftarten und zum Beispiel das Größenverhältnis der beiden Teilfenster eines Splitters. Der Eintrag <options-name> kann entweder `#f` oder ein Bezeichner sein. Dieser Bezeichner verweist auf eine weitere Liste mit Optionen, die zu den dynamisch gespeicherten Optionen hinzugefügt werden. Diese Bezeichner werden in einer separaten Datei definiert, und bilden damit soetwas wie *Vorlagen* für das Aussehen aller Windowmanager. Dies ist aber bisher noch nicht implementiert.

Zum Wiederherstellen der Konfiguration, erzeugt Orion dann einfach Windowmanager des angegebenen Typs, mit den gespeicherten Optionen.

8.4 Delegieren von neuen Programmfenstern an bestimmte Windowmanager

Eine weitere Aufgabe der Kontrollinstanz ist, ein neu erzeugtes Programmfenster in die Hierarchie der Windowmanager einzubinden. Ein Programm erzeugt ein neues Fenster als Unterfenster des *Root-Windows*. Wie bereits erwähnt unterbindet der X-Server daraufhin das Sichtbarmachen, das *Mapping*, dieses Fensters und sendet Orion stattdessen eine *MapRequest*-Nachricht. Die Kontrollinstanz behandelt dieses Ereignis dann, indem sie die `manage-window` Funktion des momentan *aktiven Windowmanagers* aufruft. Der Windowmanager selbst führt dann alle weiteren notwendigen Schritte durch. Der momentan aktive Windowmanager ist dadurch bestimmt, daß die Kontrollinstanz die Veränderungen des Input-Fokus beobachtet, das heißt sobald ein bestimmter Windowmanager den Fokus erhält, wird er zum aktiven Windowmanager. Diese Methode ist notwendig, da auch kein Fenster den Eingabe-Fokus haben kann (z.B. mit zwei Bildschirmen), aber trotzdem ein Windowmanager neue Fenster übernehmen muß.

8.5 Austauschen von Fenstern zwischen verschiedenen Windowmanagern

Um den Bedienkomfort zu erhöhen, muß Orion dem Benutzer auch die Möglichkeit bieten, ein Fenster aus einem Windowmanager herauszunehmen und zu einem anderen Windowmanager hinzuzufügen. Eine einfache Methode, die Auswahl eines Fensters zu ermöglichen ist, den Titel der Fenster zu verwenden. Der Benutzer fokussiert den Windowmanager zu dem er ein Fenster hinzufügen will, drückt eine Tastenkombination (`M-a`) und kann dann den Titel des gewünschten Fensters aus einer Liste auswählen.

9 Zusammenfassung

Zunächst wurden in dieser Arbeit die beiden Grundkonzepten der Fensterverwaltung, „Seiten und Kacheln“ und „Stapeln und Verschieben“, anhand der Beispiele Ion [Val99] beziehungsweise WindowMaker [Koj97] vorgestellt. Danach wurde gezeigt wie Orion diese beiden Konzepte in einem einzigen Windowmanager verbindet und daß eine solche Verbindung vorteilhaft ist.

Nach einer kurzen Einführung in das X-Window-System [GKM90] und der Konventionen für Windowmanager aus dem ICCCM [Ros88] wurde die Implementierung des Windowmanagers in Scheme vorgestellt. Der Windowmanager konnte im Rahmen dieser Studienarbeit allerdings nicht vollständig implementiert werden. Zum einen fehlen einige Teile wie zum Beispiel die Installation von Farbtabellen für das momentan aktive Programmfenster oder die Realisierung von benutzerspezifischen Konfigurationsdateien. Zum anderen muß die Geschwindigkeit und Stabilität der Implementierung noch stark verbessert werden.

Literatur

[GKM90] Jim Gettys, Phil Karlton, and Scott McGregor, *The X window system, version 11*, Software Practice and Experience **20** (1990), no. S2.

[Koj97] Alfredo Kojima, *Windowmaker* `http://www.windowmaker.org/`, 1997.

[Nye90] Adrian Nye, *The definitive guides to the X window system - vol. 1 xlib programming manual (v11)*, O'Reilly & Associates, Inc., Apr 1990.

[Ros88] David Rosenthal, *Inter-Client Communication Conventions Manual*, 1988.

[s48] *Scheme48* `http://www.s48.org/`.

[Shi94] Olin Shivers, *A Scheme Shell*, Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1994.

[Val99] Tuomo Valkonen, *Ion* `http://modeemi.fi/~tuomov/ion/`, 1999.

[WG92] Niklaus Wirth and Jürg Gutknecht, *Project Oberon, The Design of an Operating System and Compiler*, Addison Wesley, 1992.