1. Introduction. It is a package to manupulate windows of `Acme`

2. Legal information.

3. Implementation.

```
// Package goacme provides interface to acme programming environment
package goacme
import(
  ⟨Imports 6⟩
)
type(
  ⟨Types 5⟩
)
var(
  ⟨Variables 7⟩
)
⟨Constants 33⟩
```

4. Let's describe a begin of a test for the package. `Acme` will be started for the test.

```
⟨goacme_test.go 4⟩ ≡
package goacme
import(
  "os/exec"
  "9fans.net/go/plan9/client"
  "testing"
  ⟨Test specific imports 13⟩
)
func prepare(t *testing.T){
  _, err := client.MountService("acme")
  if err ≡ nil {
    t.Log("acme started already")
  } else {
    cmd := exec.Command("acme")
    err = cmd.Start()
    if err ≠ nil {
      t.Fatal(err)
    }
    ⟨Sleep a bit 14⟩
  }
}
⟨Test routines 15⟩
```

5. Let's describe *Window* structure. All the fields are unexported. For now *Window* contains *id* of a window, but the structure will be extended.

```
⟨Types 5⟩ ≡
  // Window is a structure to manipulate a particular acme's window.
Window struct{
  id int
  ⟨Window struct members 22⟩
}
```

See also sections 29, 32, 38, 63, 73, 76, 78, and 85.

This code is used in section 3.

6. New.

⟨ Imports 6 ⟩ ≡

```
"9fans.net/go/plan9"
"9fans.net/go/plan9/client"
"sync"
"fmt"
```

See also sections 21, 35, 47, and 88.

This code is used in section 3.

7. At first we have to mount `Acme` namespace

⟨ Variables 7 ⟩ ≡

```
fsys ∗ client.Fsys
once sync.Once
```

See also sections 36, 41, 52, 67, and 79.

This code is used in section 3.

8.

⟨ Mount `Acme` namespace 8 ⟩ ≡

```
{
  var err error
  once.Do(func(){
    fsys, err = client.MountService("acme")
  })
  if err ≠ nil {
    return nil, err
  }
}
```

This code is used in sections 9, 10, 74, and 89.

9.

```
    // New creates a new window and returns ∗Window or error
func New() (∗Window, error){
  ⟨ Mount Acme namespace 8 ⟩
  f, err := fsys.Open("new/ctl", plan9.OREAD)
  if err ≠ nil {
    return nil, err
  }
  defer f.Close()
  var id int
  if _, err := fmt.Fscan(f, &id); err ≠ nil {
    return nil, err
  }
  return Open(id)
}
```

10. Open.

```
// Open opens a window with a specified id and returns *Window or error
func Open(id int) (*Window, error){
  ⟨ Mount Acme namespace 8 ⟩
  if err := fsys.Access(fmt.Sprintf("%d", id), plan9.OREAD); err ≠ nil {
    return nil, err
  }
  this := &Window{id: id}
  ⟨ Init of Window members 23 ⟩
  return this, nil
}
```

11. *Window* **functions.**

12. Close.

```
// Close releases all resources of the window
func (this * Window) Close() error{
  ⟨ Releasing of Window members 24 ⟩
  return nil
}
```

13. Let's test *New* and *Open*

⟨ Test specific imports 13 ⟩ ≡

```
"fmt"
"time"
"9fans.net/go/plan9"
```

See also section 18.

This code is used in section 4.

14.

⟨ Sleep a bit 14 ⟩ ≡

```
time.Sleep(time.Second)
```

This code is used in section 4.

15.

⟨ Test routines 15 ⟩ ≡

```
func TestNewOpen(t *testing.T){
  prepare(t)
  w, err := New()
  if err ≠ nil {
    t.Fatal(err)
  }
  defer w.Close()
  defer w.Del(true)
  if f, err := fsys.Open(fmt.Sprintf("%d", w.id), plan9.OREAD); err ≠ nil {
    t.Fatal(err)
  } else {
    f.Close()
  }
}
```

See also sections 19, 27, 55, 59, 62, 72, 83, and 90.

This code is used in section 4.

16. Read.

```
    // Read reads len(p) bytes from "body" file of the window.
    // Read returns a count of read bytes or error.
func (this * Window) Read(p []byte) (int, error){
  f, err := this.File("body")
  if err ≠ nil {
    return 0, err
  }
  return f.Read(p)
}
```

17. Write.

```
// Write writes len(p) bytes to "body" file of the window.
// Write returns a count of written bytes or error.
func (this * Window) Write(p []byte) (int, error){
  f, err := this.File("body")
  if err ≠ nil {
    return 0, err
  }
  ⟨ Convert f to a wrapper 65 ⟩
  return f.Write(p)
}
```

18. Test of *Read* and *Write* function

```
⟨ Test specific imports 13 ⟩ +≡
  "bytes"
  "errors"
```

19.

```
⟨ Test routines 15 ⟩ +≡
  func TestReadWrite(t * testing.T){
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Close()
    defer w.Del(true)
    b1 := []byte("test")
    _, err = w.Write(b1)
    if err ≠ nil {
      t.Fatal(err)
    }
    w1, err := Open(w.id)
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w1.Close()
    defer w1.Del(true)
    b2 := make([]byte, 10)
    n, err := w1.Read(b2)
    if err ≠ nil {
      t.Fatal(err)
    }
    if bytes.Compare(b1, b2[:n]) ≠ 0 {
      t.Fatal(errors.New("buffers␣don't␣match"))
    }
  }
```

20. Seek.

```
// Seek sets a position for the next Read or Write to offset, interpreted
// according to whence: 0 means relative to the origin of the file, 1 means
// relative to the current offset, and 2 means relative to the end.
// Seek returns the new offset or error
func (this * Window) Seek(offset int64, whence int) (ret int64, err error){
  f, err := this.File("body")
  if err ≠ nil {
    return 0, err
  }
  return f.Seek(offset, whence)
}
```

21. File.

```
⟨ Imports 6 ⟩ +≡
  "io"
```

22. I have decided to store open files in **map**[**string**] ∗ *os.File*.

```
⟨ Window struct members 22 ⟩ ≡
  files map[string] ∗ client.Fid
```

See also sections 49 and 68.

This code is used in section 5.

23.

```
⟨ Init of Window members 23 ⟩ ≡
  this.files = make(map[string] ∗ client.Fid)
```

See also sections 58 and 69.

This code is used in section 10.

24. When *Window* is destroyed, all members of *files* have to be closed.

```
⟨ Releasing of Window members 24 ⟩ ≡
  for _, v := range this.files {
    v.Close()
  }
```

See also section 70.

This code is used in section 12.

25.

```
    // File returns io.ReadWriteSeeker of corresponding file of the windows or error
  func (this ∗ Window) File(file string) (io.ReadWriteSeeker, error){
    fid, ok := this.files[file]
    if ¬ok {
      var err error
      if fid, err = fsys.Open(fmt.Sprintf("%d/%s", this.id, file), plan9.ORDWR); err ≠ nil {
        if fid, err = fsys.Open(fmt.Sprintf("%d/%s", this.id, file), plan9.OREAD); err ≠ nil {
          if fid, err = fsys.Open(fmt.Sprintf("%d/%s", this.id, file), plan9.OWRITE); err ≠ nil {
            return nil, err
          }
        }
      }
      this.files[file] = fid
    }
    var f io.ReadWriteSeeker = fid
    ⟨ Convert f to a wrapper 65 ⟩
    return f, nil
  }
```

26. Del.

```
// Del deletes the window, without a prompt if sure is true.
func (this * Window) Del(sure bool) error{
  f, err := this.File("ctl")
  if err ≠ nil {
    return err
  }
  s := "del"
  if sure {
    s = "delete"
  }
  _, err = f.Write([]byte(s))
  return err
}
```

27. Test of *Del* function.

```
⟨ Test routines 15 ⟩ +≡
func TestDel(t * testing.T){
  w, err := New()
  if err ≠ nil {
    t.Fatal(err)
  }
  w.Del(true)
  w.Close()
  if _, err := Open(w.id); err ≡ nil {
    t.Fatal(errors.New(fmt.Sprintf("window␣%d␣is␣still␣opened", w.id)))
  }
}
```

28. Events processing.

29. At first let's describe *Event* structure. Fiels of *Event* will be specified a bit later.

```
⟨ Types 5 ⟩ +≡
  Event struct{
    ⟨ Fields of Event 34 ⟩
  }
```

30. *readFields* reads properties of an event from r. Some trick is used here: r is supposed not buffered, so it doesn't implement *RuneScanner* interface. When a length of text is parsing in event, a space followed by the length is read by *Fscanf* and we shouldn't read it.

```
func readFields(r io.Reader) (o rune, t rune, b int, e int, f int, s string, err error){
  var l int
  if _, err = fmt.Fscanf(r, "%c%c%d␣%d␣%d␣%d", &o, &t, &b, &e, &f, &l); err ≠ nil {
    return
  }
  if l ≠ 0 {
    rs := make([]rune, l)
    for i := 0; i⟨l; i++ {
      if _, err = fmt.Fscanf(r, "%c", &rs[i]); err ≠ nil {
        return
      }
    }
    s = string(rs)
  }
  var nl [1]byte
  if _, err = r.Read(nl[:]); err ≠ nil {
    return
  }
  return
}
```

31. *readEvent* is unexported function to read *Event* from f.

```
func readEvent(r io.Reader) (*Event, error){
  o, t, b, e, f, s, err := readFields(r)
  if err ≠ nil {
    return nil, err
  }
  var ev Event
  ⟨ Interpret origin 37 ⟩
  ⟨ Interpret action 42 ⟩
  ⟨ Fill addresses 44 ⟩
  ⟨ Interpret flag 46 ⟩
  return & ev, nil
}
```

32. Let's make a type for origin of an action

```
⟨ Types 5 ⟩ +≡
    // ActionOrigin is a origin of the action
  ActionOrigin int
```

33. Here we describe variants of *ActionOrigin*

```
⟨ Constants 33 ⟩ ≡
  const(
    Unknown ActionOrigin = 0
        // Edit is the origin for writes to the body or tag file
    Edit ActionOrigin = 1 << iota
        // File is the origin for through the window's other files
    File
        // Keyboard is the origin for keyboard actions
    Keyboard
        // Mouse is the origin for mouse actions
    Mouse
  )
```

See also sections 39 and 77.

This code is used in section 3.

34.

```
⟨ Fields of Event 34 ⟩ ≡
      // Origin will be an origin of action with type ActionOrigin
  Origin ActionOrigin
```

See also sections 40, 43, and 45.

This code is used in section 29.

35.

```
⟨ Imports 6 ⟩ +≡
  "errors"
```

36.

```
⟨ Variables 7 ⟩ +≡
      // ErrInvalidOrigin will be returned if a case of an unexpected origin of action
  ErrInvalidOrigin = errors.New("invalid origin of action")
```

37.

```
⟨ Interpret origin 37 ⟩ ≡
  switch o {
    case 'E':
      ev.Origin = Edit
    case 'F':
      ev.Origin = File
    case 'K':
      ev.Origin = Keyboard
    case 'M':
      ev.Origin = Mouse
    default:
      ev.Origin = Unknown
  }
```

This code is used in section 31.

38. Let's make a type for type of an action

```
⟨ Types 5 ⟩ +≡
        // ActionType is a type of the action
  ActionType int
```

39. Here we describe variants of *ActionType*

```
⟨ Constants 33 ⟩ +≡
  const(
    Delete ActionType = 1 << iota
    Insert
    Look
    Execute
        // Tag is a flag points out the event has occured in the tag of the window
    Tag
        // TagMask is a mask points out the event should be masked by tag
    TagMask
    AllTypes = Delete | Insert | Look | Execute
  )
```

40.

```
⟨ Fields of Event 34 ⟩ +≡
        // Type will be an type of action with type ActionType
  Type ActionType
```

41.

```
⟨ Variables 7 ⟩ +≡
        // ErrInvalidType will be returned if a case of an unexpected type of action
  ErrInvalidType = errors.New("invalid␣type␣of␣action")
```

42.

```
⟨ Interpret action 42 ⟩ ≡
  switch t {
    case 'D':
      ev.Type = Delete
    case 'd':
      ev.Type = Delete | Tag
    case 'I':
      ev.Type = Insert
    case 'i':
      ev.Type = Insert | Tag
    case 'L':
      ev.Type = Look
    case 'l':
      ev.Type = Look | Tag
    case 'X':
      ev.Type = Execute
    case 'x':
      ev.Type = Execute | Tag
    default:
      return nil, ErrInvalidType
  }
```

This code is used in section 31.

43. *Begin* and *End* are addresses of the action. *begin* and *end* are unexported addresses from an original event - they should be stored, but I decided to hide them to avoid collisions.

```
⟨ Fields of Event 34 ⟩ +≡
  begin int
      // Begin is a start address of a text of the action
  Begin int
  end int
      // End is an end address of the text of the action
  End int
```

44.

```
⟨ Fill addresses 44 ⟩ ≡
  ev.begin = b
  ev.Begin = b
  ev.end = e
  ev.End = e
```

This code is used in section 31.

45. *flag* is an unexported copy of flag from an original event

```
⟨ Fields of Event 34 ⟩ +≡
  flag int
      // IsBuiltin is a flag the action is recognised like an Acme's builtin
  IsBuiltin bool
      // NoLoad is a flag of acme can interpret the action without loading a new file
  NoLoad bool
      // IsFile is a flag the Text is a file or window name
  IsFile bool
      // Text is a text arguments of the action, perhaps with address
  Text string
      // Arg is a text of chorded argument if any
  Arg string
```

46.

```
⟨ Interpret flag 46 ⟩ ≡
  ev.flag = f
  if ev.Type & Execute ≡ Execute {
    ev.IsBuiltin = (ev.flag & 1) ≡ 1
  } else if ev.Type & Look ≡ Look {
    ev.NoLoad = (ev.flag & 1) ≡ 1
    ev.IsFile = (ev.flag & 4) ≡ 4
  }
  ev.Text = s
      // if there is an expansion
  if (ev.flag & 2) ≡ 2 {
    _, _, ev.Begin, ev.End, _, ev.Text, err = readFields(r)
    if err ≠ nil {
      return nil, err
    }
  }
      // if there is a chording
  if (ev.flag & 8) ≡ 8 {
    _, _, _, _, _, ev.Arg, err = readFields(r)
    if err ≠ nil {
      return nil, err
    }
    _, _, _, _, _, _, err = readFields(r)
    if err ≠ nil {
      return nil, err
    }
  }
  ⟨ Check if some arguments are in Text field 48 ⟩
```

This code is used in section 31.

47. If some arguments are in *Text*, then let's add them in the begin of *Arg*

```
⟨ Imports 6 ⟩ +≡
  "strings"
```

48.

⟨ Check if some arguments are in *Text* field 48 ⟩ ≡

```
if len(ev.Text)⟩0 {
    f := strings.Fields(ev.Text)
    if len(f)⟩1 {
        ev.Text = f[0]
        s := ev.Arg
        if len(s)⟩0 {
            s = "␣" + ev.Arg
        }
        ev.Arg = strings.Join(f[1:], "␣") + s
    }
}
```

This code is used in section 46.

49. EventChannel.

⟨ *Window* struct members 22 ⟩ +≡
```
ch chan *Event
```

50.

```
// EventChannel returns a channel of *Event with a buffer size
// from which events can be read or error.
// Only ActionTypes set in tmask are used.
// If TagMask is set in tmask, the event will be masked by tag. Otherwise Tag flag will be ignored.
// First call of EventChannel starts a goroutine to read events from "event" file
// and put them to the channel. Subsequent calls of EventChannel will return the same channel.
func (this *Window) EventChannel(size int, tmask ActionType) (<-chan *Event, error){
  if this.ch != nil {
    return this.ch, nil
  }
  ⟨ Trying to restrict events by type 51 ⟩
  f, err := this.File("event")
  if err != nil {
    return nil, err
  }
  if tmask & TagMask != TagMask {
    tmask |= Tag
  }
  this.ch = make(chan *Event, size)
  go func(){
    for ev, err := readEvent(f); err == nil; ev, err = readEvent(f) {
      if old && ev.Type & tmask != ev.Type {
        if ev.Type & Insert != Insert && ev.Type & Delete != Delete {
          this.UnreadEvent(ev)
        }
        continue
      }
      this.ch <- ev
    }
    close(this.ch)
    this.ch = nil
  }()
  return this.ch, nil
}
```

51. Two kinds of filtiring of events are implemented. If `Acme` has a support of events restriction, *old* is false and we do not check events because of `Acme` does it. Otherwise we check type of events.

```
⟨ Trying to restrict events by type 51 ⟩ ≡
  old := false
  {
    var em string
    if tmask & Delete ≡ Delete {
      em += "D"
    }
    if tmask & Insert ≡ Insert {
      em += "I"
    }
    if tmask & Look ≡ Look {
      em += "L"
    }
    if tmask & Execute ≡ Execute {
      em += "X"
    }
    if tmask & TagMask ≠ TagMask {
      em += strings.ToLower(em)
    }
    if err := this.WriteCtl("events␣%s\n", em); err ≠ nil {
      old = true
    }
  }
```

This code is used in section 50.

52. ReadEvent.

```
⟨ Variables 7 ⟩ +≡
      // ErrChannelAlreadyOpened will be returned
      // if channel of events is opened by call of EventChannel
  ErrChannelAlreadyOpened = errors.New("channel␣of␣events␣is␣already␣opened")
```

53.

```
      // reads an event from "event" file of the window and returns *Event or error
  func (this * Window) ReadEvent() (*Event, error){
    if this.ch ≠ nil {
      return nil, ErrChannelAlreadyOpened
    }
    f, err := this.File("event")
    if err ≠ nil {
      return nil, err
    }
    return readEvent(f)
  }
```

54. UnreadEvent. Only subset of events cat be unread - events with *Mouse* origin and *Look* and *Execute* types. All other events cause errors.

```
    // UnreadEvent writes event ev back to the "event" file,
    // indicating to acme that it should be handled internally.
  func (this * Window) UnreadEvent(ev * Event) error{
    f, err := this.File("event")
    if err ≠ nil {
      return err
    }
    var o rune
    switch ev.Origin {
      case Mouse:
        o = 'M'
      default:
        return ErrInvalidOrigin
    }
    var t rune
    switch ev.Type {
      case Look:
        t = 'L'
      case Look | Tag:
        t = 'l'
      case Execute:
        t = 'X'
      case Execute | Tag:
        t = 'x'
      default:
        return ErrInvalidType
    }
    _, err = fmt.Fprintf(f, "%c%c%d␣%d␣\n", o, t, ev.begin, ev.end)
    return err
  }
```

55. Tests for events

```
⟨ Test routines 15 ⟩ +≡
  func TestEvent(t *testing.T){
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Close()
    defer w.Del(true)
    msg := "Press␣left␣button␣of␣mouse␣on␣"
    test := "Test"
    if _, err := w.Write([]byte(msg + test)); err ≠ nil {
      t.Fatal(err)
    }
    ch, err := w.EventChannel(0, Look | Execute)
    if err ≠ nil {
      t.Fatal(err)
    }
    e, ok :=← ch
    if ¬ok {
      t.Fatal(errors.New("Channel␣is␣closed"))
    }
    if
      e.Origin ≠ Mouse∨e.Type ≠ Look∨e.Begin ≠ len(msg)∨e.End ≠ len(msg)+len(test)∨e.Text ≠ test
      {
      t.Fatal(errors.New(fmt.Sprintf("Something␣wrong␣with␣event:␣%#v", e)))
    }
    if _, err := w.Write([]byte("\nChording␣test:␣select␣'argument',␣press␣middle␣button␣of␣\
        mouse␣on␣'Execute'␣and␣press␣left␣button␣of␣mouse␣without␣releasing␣middle␣butto\
        n")); err ≠ nil
      {
      t.Fatal(err)
    }
    e, ok =← ch
    if ¬ok {
      t.Fatal(errors.New("Channel␣is␣closed"))
    }
    if e.Origin ≠ Mouse ∨ e.Type ≠ (Execute) ∨ e.Text ≠ "Execute" ∨ e.Arg ≠ "argument" {
      t.Fatal(errors.New(fmt.Sprintf("Something␣wrong␣with␣event:␣%#v", e)))
    }
    if err := w.UnreadEvent(e); err ≠ nil {
      t.Fatal(err)
    }
    if _, err := w.Write([]byte("\nPress␣middle␣button␣of␣mouse␣on␣Del␣in␣the␣window's␣tag"));
      err ≠ nil {
      t.Fatal(err)
    }
    e, ok =← ch
    if ¬ok {
      t.Fatal(errors.New("Channel␣is␣closed"))
    }
    if e.Origin ≠ Mouse ∨ e.Type ≠ (Execute | Tag) ∨ e.Text ≠ "Del" {
```

```
    t.Fatal(errors.New(fmt.Sprintf("Something␣wrong␣with␣event:␣%#v", e)))
  }
  if err := w.UnreadEvent(e); err ≠ nil {
    t.Fatal(err)
  }
}
```

56. WriteAddr.

```
    // WriteAddr writes format with args in "addr" file of the window
func (this * Window) WriteAddr(format string, args ...interface{}) error{
    f, err := this.File("addr")
    if err ≠ nil {
        return err
    }
    if len(args)⟩0 {
        format = fmt.Sprintf(format, args...)
    }
    _, err = f.Write([]byte(format))
    return err
}
```

57. ReadAddr.

```
// ReadAddr reads the address of the next read/write operation from "addr" file of the window.
// ReadAddr return begin and end offsets in symbols or error
func (this * Window) ReadAddr() (begin int, end int, err error){
  f, err := this.File("addr")
  if err ≠ nil {
    return
  }
  if _, err = f.Seek(0, 0); err ≠ nil {
    return
  }
  _, err = fmt.Fscanf(f, "%d %d", &begin, &end)
  return
}
```

58. We should have `"addr"` file is opened because `Acme` clears internal address range when `"addr"` is being opened.

```
⟨ Init of Window members 23 ⟩ +≡
  if _, err := this.File("addr"); err ≠ nil {
    return nil, err
  }
```

59. Tests for operations with addresses

```
⟨ Test routines 15 ⟩ +≡
  func TestWriteReadAddr(t * testing.T){
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Close()
    defer w.Del(true)
    if b, e, err := w.ReadAddr(); err ≠ nil {
      t.Fatal(err)
    } else if b ≠ 0 ∨ e ≠ 0 {
      t.Fatal(errors.New(fmt.Sprintf("Something wrong with address: %v, %v", b, e)))
    }
    if _, err := w.Write([]byte("test")); err ≠ nil {
      t.Fatal(err)
    }
    if err := w.WriteAddr("0,$"); err ≠ nil {
      t.Fatal(err)
    }
    if b, e, err := w.ReadAddr(); err ≠ nil {
      t.Fatal(err)
    } else if b ≠ 0 ∨ e ≠ 4 {
      t.Fatal(errors.New(fmt.Sprintf("Something wrong with address: %v, %v", b, e)))
    }
  }
```

60. WriteCtl.

```
// WriteCtl writes format with args in "ctl" file of the window
// In case format is not ended by newline, '\n' will be added to the end of format
func (this * Window) WriteCtl(format string, args ...interface{}) error{
  f, err := this.File("ctl")
  if err ≠ nil {
    return err
  }
  if len(args)⟩0 {
    format = fmt.Sprintf(format, args...)
  }
  if len(format) ≥ 0 ∧ format[len(format) − 1] ≠ '\n' {
    format += "\n"
  }
  if _, err = f.Seek(0, 0); err ≠ nil {
    return err
  }
  _, err = f.Write([]byte(format))
  return err
}
```

61. ReadCtl.

```
// ReadCtl reads the address of the next read/write operation from "ctl" file of the window.
// ReadCtl returns:
// id - the window ID
// tlen - number of characters (runes) in the tag;
// blen - number of characters in the body;
// isdir - true if the window is a directory, false otherwise;
// isdirty - true if the window is modified, false otherwise;
// wwidth - the width of the window in pixels;
// font - the name of the font used in the window;
// twidth - the width of a tab character in pixels;
// error - in case of any error.
func (this * Window) ReadCtl() (id int, tlen int, blen int, isdir bool, isdirty bool, wwidth int, font
  string, twidth int, err error){
  f, err := this.File("ctl")
  if err ≠ nil {
    return
  }
  if _, err = f.(io.Seeker).Seek(0, 0); err ≠ nil {
    return
  }
  var dir, dirty int
  _, err = fmt.Fscanf(f, "%d␣%d␣%d␣%d␣%d␣%d␣%s␣%d", &id, &tlen, &blen, &dir, &dirty, &wwidth,
      &font, &twidth)
  isdir = dir ≡ 1
  isdirty = dirty ≡ 1
  return
}
```

62. Tests for operations with `"ctl"` file

```
⟨ Test routines 15 ⟩ +≡
  func TestWriteReadCtl(t *testing.T){
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Close()
    defer w.Del(true)
    if _, err := w.Write([]byte("test")); err ≠ nil {
      t.Fatal(err)
    }
    if _, _, _, _, d, _, _, _, err := w.ReadCtl(); err ≠ nil {
      t.Fatal(err)
    } else if ¬d {
      t.Fatal(errors.New(fmt.Sprintf("The␣window␣has␣to␣be␣dirty\n")))
    }
    if err := w.WriteCtl("clean"); err ≠ nil {
      t.Fatal(err)
    }
    if _, _, _, _, d, _, _, _, err := w.ReadCtl(); err ≠ nil {
      t.Fatal(err)
    } else if d {
      t.Fatal(errors.New(fmt.Sprintf("The␣window␣has␣to␣be␣clean\n")))
    }
  }
```

63. I found `Acme` panics when a size of message is more that 8168 bytes. So I decided to make a wrapper to replace *Write* method.

```
⟨ Types 5 ⟩ +≡
  wrapper struct{
    f io.ReadWriteSeeker
  }
```

64. *wrapper* has to support *io.ReadWriteSeeker* interface, so here are the interface functions.

```
func (this * wrapper) Read(p []byte) (int, error){
  return this.f.Read(p)
}
func (this * wrapper) Write(p []byte) (int, error){
  if len(p)⟨8168 {
    return this.f.Write(p)
  }
  c := 0
  for i := 0; i⟨len(p); i += 8168 {
    n := i + 8168
    if n⟩len(p) {
      n = len(p)
    }
    n, e := this.f.Write(p[i:n])
    c += n
    if e ≠ nil {
      return c, e
    }
  }
  return c, nil
}
func (this * wrapper) Seek(offset int64, whence int) (ret int64, err error){
  return this.f.Seek(offset, whence)
}
```

65. This is a convertor to *wrapper*

```
⟨ Convert f to a wrapper 65 ⟩ ≡
  f = &wrapper{f: f}
```

This code is used in sections 17 and 25.

66. DeleteAll. *DeleteAll* deletes all windows opened in a session. So all the windows should be stored in a list. Some global variables and *Window* members are needed for this purpose.

67. *fwin* is a pointer to a first *Window* and *lwin* is a pointer to a last *Window*

```
⟨ Variables 7 ⟩ +≡
  fwin * Window
  lwin * Window
```

68. *prev* and *next* are pointer on previous *Window* and next *Window* respectively.

```
⟨ Window struct members 22 ⟩ +≡
  prev * Window
  next * Window
```

69. We need to place the window in the end of list of all windows

```
⟨ Init of Window members 23 ⟩ +≡
  this.prev = lwin
  this.next = nil
  if fwin ≡ nil {
    fwin = this
  }
  if lwin ≠ nil {
    lwin.next = this
  }
  lwin = this
```

70. When *Window* is destroyed, the *Window* has to be excluded from the list of windows

```
⟨ Releasing of Window members 24 ⟩ +≡
  if this.next ≠ nil {
    this.next.prev = this.prev
  }
  if this.prev ≠ nil {
    this.prev.next = this.next
  }
  if fwin ≡ this {
    fwin = this.next
  }
  if lwin ≡ this {
    lwin = this.prev
  }
```

71. Some trick is used to delete all *Window* - when *fwin* is closed, *fwin* is set to *fwin.next*, so to delete all the windows *fwin* will be closed until *fwin* is not null.

```
    // DeleteAll deletes all the windows opened in a session
  func DeleteAll(){
    for fwin ≠ nil {
      fwin.Del(true)
      fwin.Close()
    }
  }
```

72. Test of *DeleteAll* function.

```
⟨ Test routines 15 ⟩ +≡
  func TestDeleteAll(t *testing.T){
    var l [10]int
    for i := 0; i⟨len(l); i++ {
      w, err := New()
      if err ≠ nil {
        t.Fatal(err)
      }
      l[i] = w.id
    }
    DeleteAll()
    for _, v := range l {
      _, err := Open(v)
      if err ≡ nil {
        t.Fatal(errors.New(fmt.Sprintf("window␣%d␣is␣still␣opened", v)))
      }
    }
  }
```

73. Log. Here is function and structures for `Acme`'s log.

```
⟨ Types 5 ⟩ +≡
  Log struct{
    fid  *client.Fid
    ⟨ Log struct members 81 ⟩
  }
```

74. OpenLog.

```
    // OpenLog opens the log and returns *Log or error
func OpenLog() (*Log, error){
    ⟨ Mount Acme namespace 8 ⟩
    f, err := fsys.Open("log", plan9.OREAD)
    if err ≠ nil {
        return nil, err
    }
    return &Log{fid: f}, nil
}
```

75. Close.

```
// Close close the log
func (this *Log) Close() error{
  return this.fid.Close()
}
```

76. Let's make a type of an operation

```
⟨ Types 5 ⟩ +≡
// OperationType is a type of the operation
OperationType int
```

77. Here we describe variants of *OperationType*

```
⟨ Constants 33 ⟩ +≡
const(
  NewWin OperationType = 1 << iota
  Zerox
  Get
  Put
  DelWin
  Focus
)
```

78. Also we need *LogEvent*

```
⟨ Types 5 ⟩ +≡
LogEvent struct{
  Id int
  Type OperationType
  Name string
}
```

79. We need a map to reflect string operatios to *OperationType*

```
⟨ Variables 7 ⟩ +≡
operations = map[string]OperationType{"new": NewWin, "zerox": Zerox, "get": Get, "put":
Put, "del": DelWin, "focus": Focus, }
```

80. Read.

```
// Read reads a log of window operations of the window from the log.
// Read returns LogEvent or error.
func (this *Log) Read() (*LogEvent, error){
  var id int
  var op string
  var n string
  var b [8168]byte
  c, err := this.fid.Read(b[:])
  if err ≠ nil {
    return nil, err
  }
  _, err = fmt.Sscan(string(b[:c]), &id, &op, &n)
  if err ≠ nil {
    _, err = fmt.Sscan(string(b[:c]), &id, &op)
  }
  if err ≠ nil {
    return nil, err
  }
  t, ok := operations[op]
  if ¬ok {
    return nil, errors.New("unexpected␣operation␣code")
  }
  return &LogEvent{Id: id, Type: t, Name: n}, nil
}
```

81. EventChannel.

⟨ *Log* struct members 81 ⟩ ≡
```
ch chan *LogEvent
```
This code is used in section 73.

82.
```
// EventChannel returns a channel of *LogEvent
// from which log events can be read or error.
// Only OperationType set in tmask are used.
// First call of EventChannel starts a goroutine to read events from the log
// and put them to the channel. Subsequent calls of EventChannel will return the same channel.
func (this *Log) EventChannel(tmask OperationType) (<-chan *LogEvent, error){
  if this.ch != nil {
    return this.ch, nil
  }
  this.ch = make(chan *LogEvent)
  go func(){
    for ev, err := this.Read(); err == nil; ev, err = this.Read() {
      if ev.Type & tmask != ev.Type {
        continue
      }
      this.ch <- ev
    }
    close(this.ch)
    this.ch = nil
  }()
  return this.ch, nil
}
```

83.

```
⟨ Test routines 15 ⟩ +≡
  func TestLog(t *testing.T){
    l, err := OpenLog()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer l.Close()
    ch, err := l.EventChannel(NewWin)
    if err ≠ nil {
      t.Fatal(err)
    }
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Del(true)
    defer w.Close()
    ev, ok :=← ch
    if ¬ok {
      t.Fatal(errors.New("cannot␣read␣an␣event␣from␣log"))
    }
    if w.id ≠ ev.Id {
      t.Fatal(errors.New("unexpected␣window␣id"))
    }
  }
```

84. WindowsInfo.

85. Also we need *LogEvent*

```
⟨ Types 5 ⟩ +≡
  Info struct{
    Id int
    TagSize int
    BodySize int
    IsDirectory bool
    IsDirty bool
    Tag []string
  }
  Infos [] ∗ Info
```

86. We need sorted *Infos* slices, so some sort.Interface function have to be implemented

```
  func (this Infos) Len() int{
    return len(this)
  }
  func (this Infos) Less(i, j int) bool{
    return this[i].Id⟨this[j].Id
  }
  func (this Infos) Swap(i, j int){
    this[i], this[j] = this[j], this[i]
  }
```

87. Also *Get* function is implemented to obtain *Info* by *id*

```
    // Get returns Info by id or an error
  func (this Infos) Get(id int) (∗Info, error){
    i := sort.Search(this.Len(), func(i int) bool{
      return this[i].Id ≡ id
    })
    if i⟨this.Len() ∧ this[i].Id ≡ id {
      return this[i], nil
    }
    return nil, errors.New(fmt.Sprintf("window with id=%d has not been found", id))
  }
```

88.

```
⟨ Imports 6 ⟩ +≡
  "bufio"
  "sort"
```

89.

```
// WindowsInfo returns a list of the existing acme windows.
func WindowsInfo() (res Infos, err error){
  ⟨Mount Acme namespace 8⟩
  f, err := fsys.Open("index", plan9.OREAD)
  if err ≠ nil {
    return nil, err
  }
  defer f.Close()
  r := bufio.NewReader(f)
  if r ≡ nil {
    return nil, errors.New("cannot␣create␣reader␣for␣index␣file")
  }
  for s, err := r.ReadString('\n'); err ≡ nil; s, err = r.ReadString('\n') {
    var id, ts, bs, d, m int
    if _, err := fmt.Sscanf(s, "%v␣%v␣%v␣%v␣%v", &id, &ts, &bs, &d, &m); err ≠ nil {
      continue
    }
    res = append(res, &Info{Id: id, TagSize: ts, BodySize: bs, IsDirectory: d ≡ 1, IsDirty:
    m ≡ 1, Tag: strings.Split(s[12 ∗ 5:], "␣")})
  }
  sort.Sort(res)
  return res, nil
}
```

90.

```
⟨ Test routines 15 ⟩ +≡
  func TestWindowsInfo(t *testing.T){
    l1, err := WindowsInfo()
    if err ≠ nil {
      t.Fatal(err)
    }
    w, err := New()
    if err ≠ nil {
      t.Fatal(err)
    }
    defer w.Close()
    l2, err := WindowsInfo()
    if err ≠ nil {
      t.Fatal(err)
    }
    if len(l1) ≡ len(l2) ∨ l2[len(l2) − 1].Id ≠ w.id {
      t.Fatal(errors.New(fmt.Sprintf("something␣wrong␣with␣window␣list:␣%v,␣%v", l1, l2)))
    }
    if _, err := l1.Get(w.id); err ≡ nil {
      t.Fatal(errors.New(fmt.Sprintf(fmt.Sprintf("window␣with␣id=%d␣has␣been␣found", w.id))))
    }
    if i2, err := l2.Get(w.id); err ≠ nil ∨ i2.Id ≠ w.id {
      t.Fatal(errors.New(fmt.Sprintf(fmt.Sprintf("window␣with␣id=%d␣has␣not␣been␣found", w.id))))
    }
    w.Del(true)
    l2, err = WindowsInfo()
    if err ≠ nil {
      t.Fatal(err)
    }
    if len(l1) ≠ len(l2) {
      t.Fatal(errors.New(fmt.Sprintf("sizes␣of␣window␣lists␣mismatched:␣%v,␣%v", l1, l2)))
    }
    if _, err := l2.Get(w.id); err ≡ nil {
      t.Fatal(errors.New(fmt.Sprintf(fmt.Sprintf("window␣with␣id=%d␣has␣been␣found", w.id))))
    }
  }
```

91. Index.

The goacme package for manipulating plumb messages

(version 0.7)