// Using the template, declare a set of concrete types that implement the set

// of predefined interface types. Then create values of these types and use

// them to complete a set of predefined tasks.

package main

// Add import(s).

import "fmt"

// administrator represents a person or other entity capable of administering

// hardware and software infrastructure.

type administrator interface {

administrate(system string)

}

// developer represents a person or other entity capable of writing software.

type developer interface {

develop(system string)

}

// =============================================================================

// adminlist represents a group of administrators.

type adminlist struct {

list []administrator

}

// Enqueue adds an administrator to the adminlist.

func (l \*adminlist) Enqueue(a administrator) {

l.list = append(l.list, a)

}

// Dequeue removes an administrator from the adminlist.

func (l \*adminlist) Dequeue() administrator {

a := l.list[0]

l.list = l.list[1:]

return a

}

// =============================================================================

// devlist represents a group of developers.

type devlist struct {

list []developer

}

// Enqueue adds a developer to the devlist.

func (l \*devlist) Enqueue(d developer) {

l.list = append(l.list, d)

}

// Dequeue removes a developer from the devlist.

func (l \*devlist) Dequeue() developer {

d := l.list[0]

l.list = l.list[1:]

return d

}

// =============================================================================

// Declare a concrete type named sysadmin with a name field of type string.

type sysadmin struct {

name string

}

// Declare a method named administrate for the sysadmin type, implementing the

// administrator interface. administrate should print out the name of the

// sysadmin, as well as the system they are administering.

func (s \*sysadmin) administrate(system string) {

fmt.Println(s.name, "is administering", system)

}

// Declare a concrete type named programmer with a name field of type string.

type programmer struct {

name string

}

// Declare a method named develop for the programmer type, implementing the

// developer interface. develop should print out the name of the

// programmer, as well as the system they are coding.

func (p \*programmer) develop(system string) {

fmt.Println(p.name, "is developing", system)

}

// Declare a concrete type named company. Declare it as the composition of

// the administrator and developer interface types.

type company struct {

administrator

developer

}

// =============================================================================

func main() {

// Create a variable named admins of type adminlist.

var admins adminlist

// Create a variable named devs of type devlist.

var devs devlist

// Enqueue a new sysadmin onto admins.

admins.Enqueue(&sysadmin{"John"})

// Enqueue two new programmers onto devs.

devs.Enqueue(&programmer{"Mary"})

devs.Enqueue(&programmer{"Steve"})

// Create a variable named cmp of type company, and initialize it by

// hiring (dequeuing) an administrator from admins and a developer from devs.

cmp := company{

administrator: admins.Dequeue(),

developer: devs.Dequeue(),

}

// Enqueue the company value on both lists since the company implements

// each interface.

admins.Enqueue(&cmp)

devs.Enqueue(&cmp)

// A set of tasks for administrators and developers to perform.

tasks := []struct {

needsAdmin bool

system string

}{

{needsAdmin: false, system: "xenia"},

{needsAdmin: true, system: "pillar"},

{needsAdmin: false, system: "omega"},

}

// Iterate over tasks.

for \_, task := range tasks {

// Check if the task needs an administrator else use a developer.

if task.needsAdmin {

// Dequeue an administrator value from the admins list and

// call the administrate method.

adm := admins.Dequeue()

adm.administrate(task.system)

continue

}

// Dequeue a developer value from the devs list and

// call the develop method.

dev := devs.Dequeue()

dev.develop(task.system)

}

}