CR01

// This is an example of using type hierarchies with a OOP pattern.

// This is not something we want to do in Go. Go does not have the

// concept of sub-typing. All types are their own and the concepts of

// base and derived types do not exist in Go. This pattern does not

// provide a good design principle in a Go program.

package main

import "fmt"

// Animal contains all the base fields for animals.

type Animal struct {

Name string

IsMammal bool

}

// Speak provides generic behavior for all animals and

// how they speak.

func (a \*Animal) Speak() {

fmt.Printf(

"UGH! My name is %s, it is %t I am a mammal\n",

a.Name,

a.IsMammal,

)

}

// Dog contains everything an Animal is but specific

// attributes that only a Dog has.

type Dog struct {

Animal

PackFactor int

}

// Speak knows how to speak like a dog.

func (d \*Dog) Speak() {

fmt.Printf(

"Woof! My name is %s, it is %t I am a mammal with a pack factor of %d.\n",

d.Name,

d.IsMammal,

d.PackFactor,

)

}

// Cat contains everything an Animal is but specific

// attributes that only a Cat has.

type Cat struct {

Animal

ClimbFactor int

}

// Speak knows how to speak like a cat.

func (c \*Cat) Speak() {

fmt.Printf(

"Meow! My name is %s, it is %t I am a mammal with a climb factor of %d.\n",

c.Name,

c.IsMammal,

c.ClimbFactor,

)

}

func main() {

// Create a list of Animals that know how to speak.

animals := []Animal{

// Create a Dog by initializing its Animal parts

// and then its specific Dog attributes.

Dog{

Animal: Animal{

Name: "Fido",

IsMammal: true,

},

PackFactor: 5,

},

// Create a Cat by initializing its Animal parts

// and then its specific Cat attributes.

Cat{

Animal: Animal{

Name: "Milo",

IsMammal: true,

},

ClimbFactor: 4,

},

}

// Have the Animals speak.

for \_, animal := range animals {

animal.Speak()

}

}

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

// NOTES:

// Smells:

// \* The Animal type is providing an abstraction layer of reusable state.

// \* The program never needs to create or solely use a value of type Animal.

// \* The implementation of the Speak method for the Animal type is a generalization.

// \* The Speak method for the Animal type is never going to be called.

CR02

// This is an example of using composition and interfaces. This is

// something we want to do in Go. We will group common types by

// their behavior and not by their state. This pattern does

// provide a good design principle in a Go program.

package main

import "fmt"

// Speaker provide a common behavior for all concrete types

// to follow if they want to be a part of this group. This

// is a contract for these concrete types to follow.

type Speaker interface {

Speak()

}

// Dog contains everything a Dog needs.

type Dog struct {

Name string

IsMammal bool

PackFactor int

}

// Speak knows how to speak like a dog.

// This makes a Dog now part of a group of concrete

// types that know how to speak.

func (d \*Dog) Speak() {

fmt.Printf(

"Woof! My name is %s, it is %t I am a mammal with a pack factor of %d.\n",

d.Name,

d.IsMammal,

d.PackFactor,

)

}

// Cat contains everything a Cat needs.

type Cat struct {

Name string

IsMammal bool

ClimbFactor int

}

// Speak knows how to speak like a cat.

// This makes a Cat now part of a group of concrete

// types that know how to speak.

func (c \*Cat) Speak() {

fmt.Printf(

"Meow! My name is %s, it is %t I am a mammal with a climb factor of %d.\n",

c.Name,

c.IsMammal,

c.ClimbFactor,

)

}

func main() {

// Create a list of Animals that know how to speak.

speakers := []Speaker{

// Create a Dog by initializing its Animal parts

// and then its specific Dog attributes.

&Dog{

Name: "Fido",

IsMammal: true,

PackFactor: 5,

},

// Create a Cat by initializing its Animal parts

// and then its specific Cat attributes.

&Cat{

Name: "Milo",

IsMammal: true,

ClimbFactor: 4,

},

}

// Have the Animals speak.

for \_, spkr := range speakers {

spkr.Speak()

}

}

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

// NOTES:

// Here are some guidelines around declaring types:

// \* Declare types that represent something new or unique.

// \* Validate that a value of any type is created or used on its own.

// \* Embed types to reuse existing behaviors you need to satisfy.

// \* Question types that are an alias or abstraction for an existing type.

// \* Question types whose sole purpose is to share common state.

CR03

// Sample program demonstrating struct composition.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// System wraps Xenia and Pillar together into a single system.

type System struct {

Xenia

Pillar

}

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

// pull knows how to pull bulks of data from Xenia.

func pull(x \*Xenia, data []Data) (int, error) {

for i := range data {

if err := x.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data into Pillar.

func store(p \*Pillar, data []Data) (int, error) {

for i := range data {

if err := p.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from the System.

func Copy(sys \*System, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(&sys.Xenia, data)

if i > 0 {

if \_, err := store(&sys.Pillar, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

sys := System{

Xenia: Xenia{

Host: "localhost:8000",

Timeout: time.Second,

},

Pillar: Pillar{

Host: "localhost:9000",

Timeout: time.Second,

},

}

if err := Copy(&sys, 3); err != io.EOF {

fmt.Println(err)

}

}

CR04

// Sample program demonstrating decoupling with interfaces.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Puller declares behavior for pulling data.

type Puller interface {

Pull(d \*Data) error

}

// Storer declares behavior for storing data.

type Storer interface {

Store(d \*Data) error

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// System wraps Xenia and Pillar together into a single system.

type System struct {

Xenia

Pillar

}

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

// pull knows how to pull bulks of data from any Puller.

func pull(p Puller, data []Data) (int, error) {

for i := range data {

if err := p.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data from any Storer.

func store(s Storer, data []Data) (int, error) {

for i := range data {

if err := s.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from the System.

func Copy(sys \*System, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(&sys.Xenia, data)

if i > 0 {

if \_, err := store(&sys.Pillar, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

sys := System{

Xenia: Xenia{

Host: "localhost:8000",

Timeout: time.Second,

},

Pillar: Pillar{

Host: "localhost:9000",

Timeout: time.Second,

},

}

if err := Copy(&sys, 3); err != io.EOF {

fmt.Println(err)

}

}

CR05

// Sample program demonstrating interface composition.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Puller declares behavior for pulling data.

type Puller interface {

Pull(d \*Data) error

}

// Storer declares behavior for storing data.

type Storer interface {

Store(d \*Data) error

}

// PullStorer declares behavior for both pulling and storing.

type PullStorer interface {

Puller

Storer

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// System wraps Xenia and Pillar together into a single system.

type System struct {

Xenia

Pillar

}

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

// pull knows how to pull bulks of data from any Puller.

func pull(p Puller, data []Data) (int, error) {

for i := range data {

if err := p.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data from any Storer.

func store(s Storer, data []Data) (int, error) {

for i := range data {

if err := s.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from any System.

func Copy(ps PullStorer, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(ps, data)

if i > 0 {

if \_, err := store(ps, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

sys := System{

Xenia: Xenia{

Host: "localhost:8000",

Timeout: time.Second,

},

Pillar: Pillar{

Host: "localhost:9000",

Timeout: time.Second,

},

}

if err := Copy(&sys, 3); err != io.EOF {

fmt.Println(err)

}

}

CR06

// Sample program demonstrating decoupling with interface composition.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Puller declares behavior for pulling data.

type Puller interface {

Pull(d \*Data) error

}

// Storer declares behavior for storing data.

type Storer interface {

Store(d \*Data) error

}

// PullStorer declares behavior for both pulling and storing.

type PullStorer interface {

Puller

Storer

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// System wraps Pullers and Stores together into a single system.

type System struct {

Puller

Storer

}

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

// pull knows how to pull bulks of data from any Puller.

func pull(p Puller, data []Data) (int, error) {

for i := range data {

if err := p.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data from any Storer.

func store(s Storer, data []Data) (int, error) {

for i := range data {

if err := s.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from any System.

func Copy(ps PullStorer, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(ps, data)

if i > 0 {

if \_, err := store(ps, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

sys := System{

Puller: &Xenia{

Host: "localhost:8000",

Timeout: time.Second,

},

Storer: &Pillar{

Host: "localhost:9000",

Timeout: time.Second,

},

}

if err := Copy(&sys, 3); err != io.EOF {

fmt.Println(err)

}

}

CR07

// Sample program demonstrating removing interface pollution.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Puller declares behavior for pulling data.

type Puller interface {

Pull(d \*Data) error

}

// Storer declares behavior for storing data.

type Storer interface {

Store(d \*Data) error

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// System wraps Pullers and Stores together into a single system.

type System struct {

Puller

Storer

}

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

// pull knows how to pull bulks of data from any Puller.

func pull(p Puller, data []Data) (int, error) {

for i := range data {

if err := p.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data from any Storer.

func store(s Storer, data []Data) (int, error) {

for i := range data {

if err := s.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from any System.

func Copy(sys \*System, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(sys, data)

if i > 0 {

if \_, err := store(sys, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

sys := System{

Puller: &Xenia{

Host: "localhost:8000",

Timeout: time.Second,

},

Storer: &Pillar{

Host: "localhost:9000",

Timeout: time.Second,

},

}

if err := Copy(&sys, 3); err != io.EOF {

fmt.Println(err)

}

}

CR08

// Sample program demonstrating being more precise with API design.

package main

import (

"errors"

"fmt"

"io"

"math/rand"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

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

// Data is the structure of the data we are copying.

type Data struct {

Line string

}

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

// Puller declares behavior for pulling data.

type Puller interface {

Pull(d \*Data) error

}

// Storer declares behavior for storing data.

type Storer interface {

Store(d \*Data) error

}

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

// Xenia is a system we need to pull data from.

type Xenia struct {

Host string

Timeout time.Duration

}

// Pull knows how to pull data out of Xenia.

func (\*Xenia) Pull(d \*Data) error {

switch rand.Intn(10) {

case 1, 9:

return io.EOF

case 5:

return errors.New("Error reading data from Xenia")

default:

d.Line = "Data"

fmt.Println("In:", d.Line)

return nil

}

}

// Pillar is a system we need to store data into.

type Pillar struct {

Host string

Timeout time.Duration

}

// Store knows how to store data into Pillar.

func (\*Pillar) Store(d \*Data) error {

fmt.Println("Out:", d.Line)

return nil

}

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

// pull knows how to pull bulks of data from any Puller.

func pull(p Puller, data []Data) (int, error) {

for i := range data {

if err := p.Pull(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// store knows how to store bulks of data from any Storer.

func store(s Storer, data []Data) (int, error) {

for i := range data {

if err := s.Store(&data[i]); err != nil {

return i, err

}

}

return len(data), nil

}

// Copy knows how to pull and store data from any System.

func Copy(p Puller, s Storer, batch int) error {

data := make([]Data, batch)

for {

i, err := pull(p, data)

if i > 0 {

if \_, err := store(s, data[:i]); err != nil {

return err

}

}

if err != nil {

return err

}

}

}

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

func main() {

x := Xenia{

Host: "localhost:8000",

Timeout: time.Second,

}

p := Pillar{

Host: "localhost:9000",

Timeout: time.Second,

}

if err := Copy(&x, &p, 3); err != io.EOF {

fmt.Println(err)

}

}

CR09

// Sample program demonstrating when implicit interface conversions

// are provided by the compiler.

package main

import "fmt"

// Mover provides support for moving things.

type Mover interface {

Move()

}

// Locker provides support for locking and unlocking things.

type Locker interface {

Lock()

Unlock()

}

// MoveLocker provides support for moving and locking things.

type MoveLocker interface {

Mover

Locker

}

// bike represents a concrete type for the example.

type bike struct{}

// Move can change the position of a bike.

func (bike) Move() {

fmt.Println("Moving the bike")

}

// Lock prevents a bike from moving.

func (bike) Lock() {

fmt.Println("Locking the bike")

}

// Unlock allows a bike to be moved.

func (bike) Unlock() {

fmt.Println("Unlocking the bike")

}

func main() {

// Declare variables of the MoveLocker and Mover interfaces set to their

// zero value.

var ml MoveLocker

var m Mover

// Create a value of type bike and assign the value to the MoveLocker

// interface value.

ml = bike{}

// An interface value of type MoveLocker can be implicitly converted into

// a value of type Mover. They both declare a method named move.

m = ml

// prog.go:68: cannot use m (type Mover) as type MoveLocker in assignment:

// Mover does not implement MoveLocker (missing Lock method)

ml = m

// Interface type Mover does not declare methods named lock and unlock.

// Therefore, the compiler can't perform an implicit conversion to assign

// a value of interface type Mover to an interface value of type MoveLocker.

// It is irrelevant that the concrete type value of type bike that is stored

// inside of the Mover interface value implements the MoveLocker interface.

// We can perform a type assertion at runtime to support the assignment.

// Perform a type assertion against the Mover interface value to access

// a COPY of the concrete type value of type bike that was stored inside

// of it. Then assign the COPY of the concrete type to the MoveLocker

// interface.

b := m.(bike)

ml = b

// It's important to note that the type assertion syntax provides a way

// to state what type of value is stored inside the interface. This is

// more powerful from a language and readability standpoint, than using

// a casting syntax, like in other languages.

}

CR10

// Sample program demonstrating that type assertions are a runtime and

// not compile time construct.

package main

import (

"fmt"

"math/rand"

"time"

)

// car represents something you drive.

type car struct{}

// String implements the fmt.Stringer interface.

func (car) String() string {

return "Vroom!"

}

// cloud represents somewhere you store information.

type cloud struct{}

// String implements the fmt.Stringer interface.

func (cloud) String() string {

return "Big Data!"

}

func main() {

// Seed the number random generator.

rand.Seed(time.Now().UnixNano())

// Create a slice of the Stringer interface values.

mvs := []fmt.Stringer{

car{},

cloud{},

}

// Let's run this experiment ten times.

for i := 0; i < 10; i++ {

// Choose a random number from 0 to 1.

rn := rand.Intn(2)

// Perform a type assertion that we have a concrete type

// of cloud in the interface value we randomly chose.

if v, is := mvs[rn].(cloud); is {

fmt.Println("Got Lucky:", v)

continue

}

fmt.Println("Got Unlucky")

}

}

CR11

// Sample program to show how method sets can affect behavior.

package main

import "fmt"

// user defines a user in the system.

type user struct {

name string

email string

}

// String implements the fmt.Stringer interface.

func (u \*user) String() string {

return fmt.Sprintf("My name is %q and my email is %q", u.name, u.email)

}

func main() {

// Create a value of type user.

u := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

// Display the values.

fmt.Println(u)

fmt.Println(&u)

}

CR12

// This is an example that creates interface pollution

// by improperly using an interface when one is not needed.

package main

// Server defines a contract for tcp servers.

type Server interface {

Start() error

Stop() error

Wait() error

}

// server is our Server implementation.

type server struct {

host string

// PRETEND THERE ARE MORE FIELDS.

}

// NewServer returns an interface value of type Server

// with a server implementation.

func NewServer(host string) Server {

// SMELL - Storing an unexported type pointer in the interface.

return &server{host}

}

// Start allows the server to begin to accept requests.

func (s \*server) Start() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

// Stop shuts the server down.

func (s \*server) Stop() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

// Wait prevents the server from accepting new connections.

func (s \*server) Wait() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

func main() {

// Create a new Server.

srv := NewServer("localhost")

// Use the API.

srv.Start()

srv.Stop()

srv.Wait()

}

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

// NOTES:

// Smells:

// \* The package declares an interface that matches the entire API of its own concrete type.

// \* The interface is exported but the concrete type is unexported.

// \* The factory function returns the interface value with the unexported concrete type value inside.

// \* The interface can be removed and nothing changes for the user of the API.

// \* The interface is not decoupling the API from change.

CR13

// This is an example that removes the interface pollution by

// removing the interface and using the concrete type directly.

package main

// Server is our Server implementation.

type Server struct {

host string

// PRETEND THERE ARE MORE FIELDS.

}

// NewServer returns an interface value of type Server

// with a server implementation.

func NewServer(host string) \*Server {

// SMELL - Storing an unexported type pointer in the interface.

return &Server{host}

}

// Start allows the server to begin to accept requests.

func (s \*Server) Start() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

// Stop shuts the server down.

func (s \*Server) Stop() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

// Wait prevents the server from accepting new connections.

func (s \*Server) Wait() error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

func main() {

// Create a new Server.

srv := NewServer("localhost")

// Use the API.

srv.Start()

srv.Stop()

srv.Wait()

}

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

// NOTES:

// Here are some guidelines around interface pollution:

// \* Use an interface:

// \* When users of the API need to provide an implementation detail.

// \* When API’s have multiple implementations that need to be maintained.

// \* When parts of the API that can change have been identified and require decoupling.

// \* Question an interface:

// \* When its only purpose is for writing testable API’s (write usable API’s first).

// \* When it’s not providing support for the API to decouple from change.

// \* When it's not clear how the interface makes the code better.

CR14

// Package pubsub simulates a package that provides publication/subscription

// type services.

package pubsub

// PubSub provides access to a queue system.

type PubSub struct {

host string

// PRETEND THERE ARE MORE FIELDS.

}

// New creates a pubsub value for use.

func New(host string) \*PubSub {

ps := PubSub{

host: host,

}

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return &ps

}

// Publish sends the data for the specified key.

func (ps \*PubSub) Publish(key string, v interface{}) error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

// Subscribe sets up an request to receive messages for the specified key.

func (ps \*PubSub) Subscribe(key string) error {

// PRETEND THERE IS A SPECIFIC IMPLEMENTATION.

return nil

}

CR15

// Sample program to show how you can personally mock concrete types when

// you need to for your own packages or tests.

package main

import (

"github.com/ardanlabs/gotraining/topics/go/design/composition/mocking/example1/pubsub"

)

// publisher is an interface to allow this package to mock the pubsub

// package support.

type publisher interface {

Publish(key string, v interface{}) error

Subscribe(key string) error

}

// mock is a concrete type to help support the mocking of the pubsub package.

type mock struct{}

// Publish implements the publisher interface for the mock.

func (m \*mock) Publish(key string, v interface{}) error {

// ADD YOUR MOCK FOR THE PUBLISH CALL.

return nil

}

// Subscribe implements the publisher interface for the mock.

func (m \*mock) Subscribe(key string) error {

// ADD YOUR MOCK FOR THE SUBSCRIBE CALL.

return nil

}

func main() {

// Create a slice of publisher interface values. Assign

// the address of a pubsub.PubSub value and the address of

// a mock value.

pubs := []publisher{

pubsub.New("localhost"),

&mock{},

}

// Range over the interface value to see how the publisher

// interface provides the level of decoupling the user needs.

// The pubsub package did not need to provide the interface type.

for \_, p := range pubs {

p.Publish("key", "value")

p.Subscribe("key")

}

}