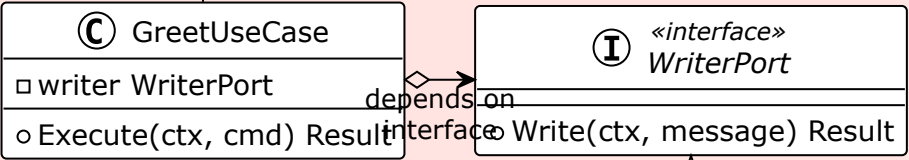


# Static vs Dynamic Dispatch

## Dependency Injection in Go with Generics

### Dynamic Dispatch (Runtime Polymorphism)



**Runtime Cost:**

- Vtable lookup each call
- No inlining possible
- Interface value boxing

**Benefits:**

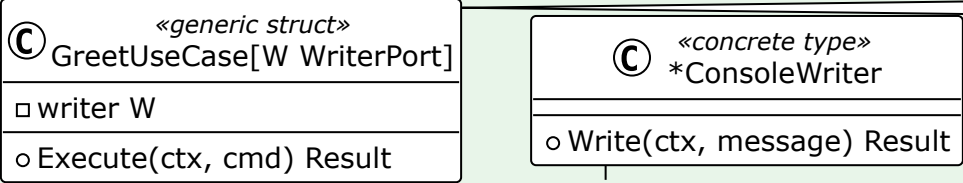
- Runtime flexibility
- Can swap at runtime

#### Traditional Go Pattern (NOT used)

```
```go
// Interface-based injection
type GreetUseCase struct {
    writer WriterPort // interface type
}

// Runtime dispatch via vtable
func (uc *GreetUseCase) Execute(...) {
    uc.writer.Write(...) // vtable lookup
}
```
```

### Static Dispatch (Compile-Time Polymorphism)



**Compile-Time Cost:**

- Zero overhead
- Direct call (inlinable)
- No vtable lookup

**Trade-off:**

- Fixed at compile time
- Larger binary (monomorphization)

#### This Project Uses Static Dispatch

```
```go
// Bootstrap instantiates with concrete types
writer := adapter.NewConsoleWriter()
uc := usecase.NewGreetUseCase[*adapter.ConsoleWriter](writer)
cmd := command.NewGreetCommand[*usecase.GreetUseCase[*adapter.ConsoleWriter]](uc)

// Compiler knows exact types
// All calls are statically dispatched
```
```

#### Why Static Dispatch?

**Performance:**

- Zero runtime overhead
- Method calls can be inlined
- No interface value allocation

**Type Safety:**

- Full type checking at compile time
- Errors caught before runtime

**Matching Ada Pattern:**

- Ada uses generics for static dispatch
- Go generics provide equivalent pattern
- Same architecture, same performance

#### Go Generics (1.18+)

```
```go
type GreetUseCase[W WriterPort] struct {
    writer W // Concrete type, not interface
}

// W is known at instantiation:
// NewGreetUseCase[*ConsoleWriter](w)
// Compiler generates code for *ConsoleWriter
```
```