

# Functional Design Patterns

Franz Thoma

Summer BOBKonf 2019, Berlin, 2019-08-21



# Are There Design Patterns in Functional Programming?

# How do you implement common design patterns like Repository, Factory, IoC in Haskell?

— [Question on Reddit \(paraphrased\)](#)

↑

19

↓

Article on OOP design pattern alternatives in Haskell

I was wondering if there are some more coherent articles that discuss some common design patterns like *repository*, *factory*, *IoC*, etc... but from Haskell perspective?


# How does one design a large scale application with functional programming?

## — Question on Reddit (paraphrased)

↑

77


↓


Posted by 


**Haskell Design Patterns?**


I come from OOP and as I learn Haskell what I find particularly hard is to understand the design strategy that one uses in functional programming to create a large application. In OOP one has to identify those elements of the application that make sense to be represented as objects, their relationships, their behaviour and then create classes to express them and encapsulate their data and operations (methods). For example, when one wants to write an application which deals with geometrical entities he can represent them in classes like Triangle, Tetrahedron etc and handle them through some base class like Shape in a generic manner. How does one design a large scale application (not simple examples) with functional programming?


I think that this kind of knowledge and examples are very important for any programming language to become popular and although one can find a lot of material for OOP there is a profound lack of such information and design tutorials for functional programming except for syntax and abstract mathematical ideas when a developer needs more practical information and design patterns to learn and adapt to his needs.

 63 Comments

 Share

 Save

 Hide

 Report

93% Upvoted

# Functional programming has fewer “design patterns” and more “libraries”

## — One of the answers

↑  
↓ Functional programming has fewer "design patterns" and more "libraries" -- we're a bit better about abstracting out the repetitive patterns in approaches to problems. That's not to say that we don't have our own design patterns (which might someday influence the design of future languages such that we can eliminate them), but it's harder to recognise them in general at the moment.

Here is a [talk](#) I highly recommend, given by Simon Peyton Jones, discussing one of the major approaches to functional programming which I would consider a "design pattern" of sorts -- embedded domain-specific languages.

At present, it's hard to imagine taking the entirety of that approach and turning it into a reusable library to kill the pattern entirely, though there is no shortage of libraries which can help with various aspects of it.

# Types types types. Everything starts from the types.

## — Another answer

↑  
↓ Types types types. Everything starts from the types. Figure out what data types your application needs. Then figure out what operations you need to do on those types. Try to make them pure functions as much as possible. This is where it all starts. It's very similar to OO classes actually, minus the inheritance.

There are plenty of other things to talk about. But that would take a lot longer, and this is really the core to it all. This talk by Conal Elliott does a great job at showing this in action:

<https://www.youtube.com/watch?v=bmKYiUOEo2A>

# The functor design pattern

— Blog post by Gabriel Gonzalez

Saturday, September 15, 2012

## The functor design pattern

This post builds on my previous post on the [category design pattern](#) and this time I will discuss the functor design pattern. If you are an intermediate Haskell programmer and you think you already understand functors, then think again, because I promise you this post will turn most of your preconceptions about functors on their head and show you how functors are much more powerful and generally applicable than you might realize.

# How do you mock things in Haskell?

— A colleague.



# How do you do Dependency Injection in Haskell?

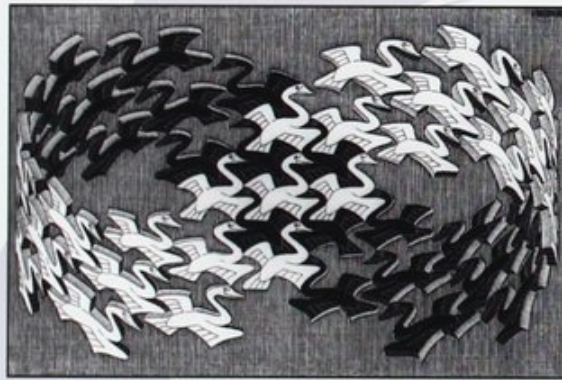
— A colleague.

# What Is A Design Pattern, Anyway?

# Design Patterns

## Elements of Reusable Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Wikipedia Definition

*In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.*

# Is **Iterator** a design pattern in Java?

- ▶ It's listed in GoF...
- ▶ But it's a library interface you can implement.

```
interface Iterable<T> {  
    Iterator<T> iterator();  
}  
  
interface Iterator<T> {  
    T next();  
    boolean hasNext();  
}
```

- ▶ Even comes with syntactic sugar!

```
Iterable<Item> iterable = ...  
  
for (Item item: iterable) {  
    ...  
}
```

# Working Definition

*A Design Pattern is a reusable solution for a recurring problem that can be given in terms of an easy-to-follow recipe, but not as a reusable piece of code.*

# GoF

*If we assumed procedural languages, we might have included design patterns called “Inheritance,” “Encapsulation,” and “Polymorphism.” Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 366).*

# The *Monad* design pattern in Java...

```
class Optional<T> {  
    public <R> Optional<R> flatMap(Function<T, Optional<R>> f) { ... }  
}  
  
class Future<T> {  
    public <R> Future<R> flatMap(Function<T, Future<R>> f) { ... }  
}
```

It's a pattern!



# ...and in Haskell

```
instance Monad Maybe where ...  
instance Monad Future where ...
```

It's a (standard) library interface, not a pattern!

# OOP design patterns that do not work in functional programming

| OOP               | FP                     |
|-------------------|------------------------|
| Strategy Pattern  | Higher-Order Functions |
| Visitor Pattern   | Pattern Matching       |
| Singleton Pattern | Top-Level Constant     |

# OOP concepts that can be emulated as a pattern in functional programming

| OOP               | FP   |
|-------------------|--|
| Classes/Instances | Service Pattern  |
| Subtyping         | N.N. (See e.g. <a href="#">optics</a> package: <code>class Is a b where ...</code> ) |

# Some Functional Design Patterns

# The *Transformer Stack* Pattern

Managing and combining monadic side effects

# The *Transformer Stack* Pattern

```
newtype AppT s m a = App (StateT s (ReaderT Config m) a)
    deriving (Functor, Applicative, Monad)

instance MonadTrans (AppT s) where
    lift = AppT . lift . lift

runAppT :: AppT s m a -> s -> Config -> m (a, s)
runAppT (AppT action) initialState config = runStateT (runReaderT action config)
initialState

type App s a = AppT s Identity a

runApp :: App s a -> s -> Config -> (a, s)
runApp action initialState config = runIdentity (runAppT action initialState config)
```

```
config :: AppT s m Config
config = AppT (lift ask)

appState :: AppT s m s
appState = AppT get
```

# The *Transformer Stack* Pattern

```
handleEvent :: Event      → App      AppState ()  
renderApp   :: AppT       AppState IO      ()  
renderButton :: AppT      ButtonState IO      ()
```

# The *Service* Pattern

Exchangable implementations of components

Inversion of Control by emulating OOP interfaces/objects with methods.



# The *Service* Pattern

```
-- The service »interface«
data Logger = Logger { log :: Level → String → IO () }

-- The »implementations«
withFileLogger :: FilePath → Level → (Logger → IO a) → IO a
withFileLogger logFile threshold action = do
  handle ← openFile logFile AppendMode
  action Logger { log = \level msg → guard (level >= threshold) (hPutStrLn handle msg) }
  hclose handle

withConsoleLogger :: Level → (Logger → IO a) → IO a
withConsoleLogger threshold action =
  action Logger { log = \level msg → guard (level >= threshold) (putStrLn msg) }

withNoOpLogger :: (Logger → IO a) → IO a
withNoOpLogger action = action Logger { log = \_ _ → pure () }

-- Convenience
logError , logWarn, logInfo, logDebug :: MonadReader Logger m ⇒ String → m ()
logError msg = ask >>= \logger → log logger Error msg
logWarn msg = ask >>= \logger → log logger Warn msg
logInfo msg = ask >>= \logger → log logger Info msg
logDebug msg = ask >>= \logger → log logger Debug msg
```

# The *Service* Pattern

```
launchWithCountdown :: (MonadIO m, MonadReader Logger m) => m ()
launchWithCountdown = do
  logInfo "3"
  logInfo "2"
  logInfo "1"
  liftIO launchMissiles
  logInfo "Missiles launched."
```

```
-- A »mock implementation« ...
withCollectingLogger :: (Logger -> IO a) -> IO (a, [(Level, String)])
withCollectingLogger action = do
  messagesRef <- newIORef []
  result <- action Logger { log = \level msg -> modifyIORef' ((level, msg):) messagesRef }
  messages <- readIORef messagesRef
  pure (result, messages)

-- ... can be used for testing
withCollectingLogger launchWithCountdown
```

# The *Service* Pattern

```
-- »dependency injection«  
main      = runManaged $ do  
  logger ← managed (withConsoleLogger Info )  
  database ← managed (withMySQLDatabase logger) -- inject Logger service into Database service  
  ...
```

## Sources:

- ▶ <https://www.schoolofhaskell.com/user/meiersi/the-service-pattern>
- ▶ <https://jaspervdj.be/posts/2018-03-08-handle-pattern.html>

# The *Config Monoid* Pattern (aka *Partial Options Monoid*)

Merging config options from multiple sources with a given precedence

# The *Config Monoid* Pattern (aka *Partial Options Monoid*)

```
data Options = Options
  { verbose :: Bool
  , cacheTimeout :: Maybe Long
  }

data PartialOptions = PartialOptions
  { pVerbose :: Last Bool
  , pCacheTimeout :: Last Long
  }
  deriving (Monoid)
```

```
buildOptions :: PartialOptions → Validation String Options
buildOptions PartialOptions {..} = Options
  <$> maybe (Failure ["No verbosity given"]) Success (getLast pVerbose)
  <*> getLast pCacheTimeout
```

# The *Config Monoid* Pattern (aka *Partial Options Monoid*)

```
loadOptions :: FilePath → IO      Options
loadOptions configFile = do
  configFileOptions ← parseConfigFile configFile
  cmdLineOptions ← parseCmdLineOptions
  case buildOptions (defaultOptions <> configFileOptions <> cmdLineOptions) of
    Success  opts → pure opts
    Failure  errs → error ("Configuration errors: " ++ intercalate ", "      errs)
```

Sources:

- <https://medium.com/@jonathangfischhoff/the-partial-options-monoid-pattern-31914a71fc67>

Thank you!

Questions?



# Thank you!

Slides on Github: [github.com/fmthoma/functional-design-patterns-slides](https://github.com/fmthoma/functional-design-patterns-slides)

[fmthoma](#) on Github

[fmthoma](#) on keybase.io

[franz.thoma@tngtech.com](mailto:franz.thoma@tngtech.com)