Design Pattern [Wikipedia]:
[A] design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

Example: In an OO Language like Java or C#, operations on data are tied to classes. Thus:
- Cannot (directly) add a new operation on data without changing all involved classes.
- The code for an operation gets spread out across all involved classes.

Solution: The Visitor pattern (or double dispatch):
- Allows operations to be defined separately from data classes and in one place.
- Allows operations to be defined by simple “pattern matching” (case analysis).
- Not entirely trivial: takes a lecture to explain. See:
  http://en.wikipedia.org/wiki/Visitor_pattern

Functional languages provides separation between operations and data, and typically pattern matching too, “for free”.
However, handling effects in a pure language requires work because, by definition, there are no implicit effects in a pure language.
This lecture: A design pattern for effects.
A Blessing and a Curse

• The **BIG** advantage of pure functional programming is
  “everything is explicit;”
i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
• The **BIG** problem with pure functional programming is
  “everything is explicit.”
Can really add a lot of clutter, especially in large programs.

Example: LTXL Identification (1)

`enterVar` inserts a variable at the given scope level and of the given type into an environment.

• Check that no variable with same name has been defined at the same scope level.
• If not, the new variable is entered, and the **resulting environment** is returned.
• Otherwise an **error message** is returned.

```
enterVar :: Id -> Int -> Type -> Env
          -> Either Env ErrorMsg
```

Example: LTXL Identification (2)

Goals of LTXL identification phase:

• Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.
  I.e., map unannotated AST `Exp ()` to annotated AST `Exp Attr`.
• **Report** conflicting variable definitions and undefined variables.

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) = 
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
      case enterVar i l t env of
        Left env' -> (env', [])
        Right m -> (env, [m])
    (ds'', env'', ms3) =
      identDefs l env' ds
```

Example: LTXL Identification (3)
Example: LTXL Identification (4)

Error checking and collection of error messages arguably added a lot of clutter. The core of the algorithm is this:

```haskell
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
where
  e' = identAux l env e
  env' = enterVar i l t env
  (ds', env'') = identDefs l env' ds
```

Example: A Simple Evaluator

```haskell
data Exp = Lit Integer
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp

eval :: Exp -> Int
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 \( \text{`div'} \) eval e2
```

Making the evaluator safe (1)

```haskell
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 + n2)
```
Making the evaluator safe (3)

\[
\text{safeEval} (\text{Mul } e1 \ e2) = \\
\quad \text{case safeEval } e1 \text{ of} \\
\quad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \quad \text{Just } n1 \rightarrow \\
\quad \quad \quad \text{case safeEval } e2 \text{ of} \\
\quad \quad \quad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \quad \quad \quad \text{Just } n2 \rightarrow \text{Just } (n1 * n2)
\]

Making the evaluator safe (4)

\[
\text{safeEval} (\text{Div } e1 \ e2) = \\
\quad \text{case safeEval } e1 \text{ of} \\
\quad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \quad \text{Just } n1 \rightarrow \\
\quad \quad \quad \text{case safeEval } e2 \text{ of} \\
\quad \quad \quad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \quad \quad \quad \text{Just } n2 \rightarrow \\
\quad \quad \quad \quad \quad \text{if } n2 == 0 \\
\quad \quad \quad \quad \quad \text{then Nothing} \\
\quad \quad \quad \quad \quad \text{else Just } (n1 \div n2)
\]

Any common pattern?

Clearly a lot of code duplication! Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
- If one evaluation fail, fail overall.
- Otherwise, make result available to following evaluations.

Example: Numbering trees

\[
\text{data Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a)
\]

\[
\text{numberTree :: Tree } a \rightarrow \text{Tree } \text{Int}
\]

\[
\text{numberTree } t = \text{fst } (\text{ntAux } t \ 0)
\]

\[
\quad \text{where}
\]

\[
\text{ntAux } (\text{Leaf } _) \quad n = (\text{Leaf } n, n+1)
\]

\[
\text{ntAux } (\text{Node } t1 \ t2) n = \\
\quad \text{let } (t1', n') = \text{ntAux } t1 \ n \\
\quad \quad \text{in let } (t2', n'') = \text{ntAux } t2 \ n' \\
\quad \quad \quad \text{in } (\text{Node } t1' \ t2', n'')
\]
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

Sequencing evaluations (1)

**Sequencing** is common to both examples, with the outcome of a computation affecting subsequent computations.

```haskell
evalSeq :: Maybe Integer
        -> (Integer -> Maybe Integer)
        -> Maybe Integer

evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Sequencing evaluations (2)

```haskell
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)

evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Sequencing evaluations (3)

```haskell
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
        safeEval e2 `evalSeq` (\n2 ->
            Just (n1 + n2))
    safeEval (Sub e1 e2) =
        safeEval e1 `evalSeq` (\n1 ->
            safeEval e2 `evalSeq` (\n2 ->
                Just (n1 - n2)))
```
Sequencing evaluations (4)

- safeEval (Mul e1 e2) =
  - safeEval e1 'evalSeq' (\n1 ->
  - safeEval e2 'evalSeq' (\n2 ->
    Just (n1 - n2))
- safeEval (Div e1 e2) =
  - safeEval e1 'evalSeq' (\n1 ->
  - safeEval e2 'evalSeq' (\n2 ->
    if n2 == 0 then Nothing
    else Just (n1 'div' n2))

Aside: Scope rules of $\lambda$-abstractions

The scope rules of $\lambda$-abstractions are such that parentheses can be omitted:

- safeEval :: Exp -> Maybe Integer...
- safeEval (Add e1 e2) =
  - safeEval e1 'evalSeq' \n1 ->
  - safeEval e2 'evalSeq' \n2 ->
    Just (n1 + n2)

Exercise 1: Inline evalSeq (1)

- safeEval (Add e1 e2) =
  - safeEval e1 'evalSeq' \n1 ->
  - safeEval e2 'evalSeq' \n2 ->
    Just (n1 + n2)

Exercise 1: Inline evalSeq (2)

- safeEval (Add e1 e2) =
  - case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> safeEval e2 'evalSeq' (\n2 -> ...)
Exercise 1: Inline `evalSeq (3)`

```haskell
= safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just n2 -> (Just n1 + n2)
```

Maybe viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a `computation` of a value of type `a` that may fail.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. *failure is an effect*, implicitly affecting subsequent computations.
- Let's adopt names reflecting our intentions.

Maybe viewed as a computation (2)

Successful computation of a value:

```haskell
mbReturn :: a -> Maybe a
mbReturn = Just
```

Sequencing of possibly failing computations:

```haskell
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a -> f a
```

Maybe viewed as a computation (3)

Failing computation:

```haskell
mbFail :: Maybe a
mbFail = Nothing
```
The safe evaluator revisited

```haskell
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
  safeEval e1 'mbSeq' \n1 ->
  safeEval e2 'mbSeq' \n2 ->
  mbReturn (n1 + n2)
...
```

SafeEval (Div e1 e2) =
  safeEval e1 'mbSeq' \n1 ->
  safeEval e2 'mbSeq' \n2 ->
  if n2 == 0 then mbFail
  else mbReturn (n1 'div' n2))

Stateful Computations (1)

- A stateful computation consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:
  ```haskell
type S a = Int -> (a, Int)
  (Only Int state for the sake of simplicity.)
```
- A value (function) of type `S a` can now be viewed as denoting a stateful computation computing a value of type `a`.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. state updating is an effect, implicitly affecting subsequent computations. (As we would expect.)

Stateful Computations (3)

Computation of a value without changing the state:
```haskell
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:
```haskell
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
  let (a, n') = sa n
  in f a n'
```
Stateful Computations (4)

Reading and incrementing the state:

```haskell
sInc :: S Int
sInc = \n -> (n, n + 1)
```

Numbering trees revisited

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
where
  ntAux (Leaf _) =
    sInc 'sSeq' \n -> sReturn (Leaf n)
  ntAux (Node t1 t2) =
    ntAux t1 'sSeq' \t1' ->
    ntAux t2 'sSeq' \t2' ->
    sReturn (Node t1' t2')
```

Observations

- The “plumbing” has been captured by the abstractions.
- In particular, there is no longer any risk of “passing on” the wrong version of the state!

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations
- In fact, both examples are instances of the general notion of a **MONAD.**
Monads in Functional Programming

A monad is represented by:

- A type constructor
  \[ M : \ast \rightarrow \ast \]
  \[ M \mathbf{T} \] represents computations of a value of type \( \mathbf{T} \).
- A polymorphic function
  \[ \text{return} :: \mathbf{a} \rightarrow M \mathbf{a} \]
  for lifting a value to a computation.
- A polymorphic function
  \[ (\gg\gg) :: M \mathbf{a} \rightarrow (\mathbf{a} \rightarrow M \mathbf{b}) \rightarrow M \mathbf{b} \]
  for sequencing computations.

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a Type Class:

```haskell
class Monad m where
    return :: \mathbf{a} \rightarrow m \mathbf{a}
    (\gg\gg) :: m \mathbf{a} \rightarrow (\mathbf{a} \rightarrow m \mathbf{b}) \rightarrow m \mathbf{b}
```

This allows the names of the common functions to be overloaded, and the sharing of derived definitions.

Monads in Haskell (2)

The Haskell monad class have two further methods with default instances:

```haskell
(\gg\gg) :: m \mathbf{a} \rightarrow m \mathbf{b} \rightarrow m \mathbf{b}
\text{m } \gg \text{k } = \text{m } \gg\gg \_ \rightarrow \text{k}

\text{fail } :: \text{String } \rightarrow m \mathbf{a}
\text{fail } s = \text{error } s
```

The Maybe monad in Haskell

```haskell
instance Monad Maybe where
    -- return :: \mathbf{a} \rightarrow \text{Maybe } \mathbf{a}
    return = \text{Just}
    -- (\gg\gg) :: \text{Maybe } \mathbf{a} \rightarrow (\mathbf{a} \rightarrow \text{Maybe } \mathbf{b})
    \rightarrow \text{Maybe } \mathbf{b}
    \text{Nothing } \gg\gg \_ = \text{Nothing}
    (\text{Just } x) \gg\gg f = f x
```
Monad-specific operations

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```haskell
fail :: String -> Maybe a
catch :: Maybe a -> Maybe a -> Maybe a
    m1 'catch' m2 =
        case m1 of
            Just _ -> m1
            Nothing -> m2
```

The do-notation (1)

Haskell provides convenient syntax for programming with monads:

```haskell
do
    a <- exp₁
    b <- exp₂
    return exp₃
```

is syntactic sugar for

```haskell
exp₁ >>= \a ->
exp₂ >>= \b ->
return exp₃
```

The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

```haskell
do
    exp₁
    exp₂
    return exp₃
```

is syntactic sugar for

```haskell
exp₁ >>= \_ ->
exp₂ >>= \_ ->
return exp₃
```

The HMTC Diagnostics Monad

```haskell
D :: * -> * -- Instances: Monad.
emitInfoD :: SrcPos -> String -> D ()
emitWngD :: SrcPos -> String -> D ()
emitErrD :: SrcPos -> String -> D ()
failD :: SrcPos -> String -> D a
failNoReasonD :: D a
failIfErrorsD :: D ()
stopD :: D a
runD :: D a -> (Maybe a, [DMsg])
```

(Roughly: The actual HMTC impl. is more refined.)
Identification Revisited (1)

Recall:

\[
\text{enterVar} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \\
\to \text{Either Env String}
\]

Let’s define a version using the Diagnostics monad:

\[
\text{enterVarD} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \to \text{D Env}
\]

\[
\text{enterVarD} \ i \ l \ t \ \text{env} = \\
\begin{cases}
\text{Left env’} & \to \text{return env’} \\
\text{Right m} & \to \text{do} \\
\hspace{1em} \text{emitErrD NoSrcPos m} \\
\hspace{1em} \text{return env}
\end{cases}
\]

Identification Revisited (2)

Now we can define a monadic version of \(\text{identDefs}\):

\[
\text{identDefs} :: \text{Int} \to \text{Env} \to \{(\text{Id},\text{Type},\text{Exp}())\} \\
\to \text{D}\{(\text{Id},\text{Type},\text{Exp Attr})\}, \text{Env}
\]

\[
\text{identDefs} \ l \ \text{env} \ [] = \text{return} \ (\ [], \ \text{env})
\]

\[
\text{identDefs} \ l \ \text{env} \ (\ (i,t,e) : \ \text{ds}) = \text{do} \\
\hspace{1em} e’ & \leftarrow \text{identAux} \ l \ \text{env} \ e \\
\hspace{1em} \text{env’} & \leftarrow \text{enterVarD} \ i \ l \ t \ \text{env} \\
\hspace{1em} (\text{ds’}, \ \text{env’}) & \leftarrow \text{identDefs} \ l \ \text{env’} \ \text{ds} \\
\hspace{1em} \text{return} \ (\ (i,t,e’) : \ \text{ds’}, \ \text{env’})
\]

Identification Revisited (3)

Compare with the “core” identified earlier!

\[
\text{identDefs} \ l \ \text{env} \ [ ] = \ (\ [], \ \text{env})
\]

\[
\text{identDefs} \ l \ \text{env} \ ((i,t,e) : \ \text{ds}) = \\
\hspace{1em} ((i,t,e’) : \ \text{ds’}, \ \text{env’})
\]

where

\[
\begin{align*}
\text{e’} &= \text{identAux} \ l \ \text{env} \ e \\
\text{env’} &= \text{enterVar} \ i \ l \ t \ \text{env} \\
(\text{ds’}, \ \text{env’}) &= \text{identDefs} \ l \ \text{env’} \ \text{ds}
\end{align*}
\]

The monadic version is very close to ideal, without sacrificing functionality, clarity, or purity!

Further Reading

  [http://www.cs.nott.ac.uk/~gmh/monads](http://www.cs.nott.ac.uk/~gmh/monads)