Exercises in Programming Style

Workshop

The book: <https://www.amazon.com/Exercises-Programming-Style-Cristina-Videira/dp/1482227371>  
On GitHub: <https://github.com/crista/exercises-in-programming-style>

# Historical

## Good old times / Early 50s style

* Very small amount of primary memory, typically orders of magnitude smaller than the data that needs to be processed/generated. (The example sets the limit to 1024 cells)
* No labels -- i.e. no variable names or tagged memory addresses. All we have is memory that is addressable with numbers.

## Go-Forth / Stack machine

* Existence of an all-important data stack. All operations (conditionals, arithmetic, etc.) are done over data on the stack
* Existence of a heap for storing data that's needed for later operations. The heap data can be associated with names (i.e. variables). As said above, all operations are done over data on the stack, so any heap data that needs to be operated upon needs to be moved first to the stack and eventually back to the heap
* Abstraction in the form of user-defined "procedures" (i.e. names bound to a set of instructions), which may be called something else entirely

# Basic Styles

## Monolith / Labyrinth / Brain dump

* No abstractions
* No use of library functions

## Cookbook / Procedural

* Larger problem decomposed in procedural abstractions
* Larger problem solved as a sequence of commands, each corresponding to a procedure

## Candy factory / Functional / Pipeline

* Larger problem decomposed in functional abstractions. Functions, according to Mathematics, are relations from inputs to outputs.
* Larger problem solved as a pipeline of function applications

## Code golf

* As few lines of code as possible

# Function Composition

## Infinite mirror / Inductive / Recursive

* All, or a significant part, of the problem is modelled by induction. That is, specify the base case (n\_0) and then the n+1 rule

## Continuation-passing style

Variation of the candy factory style, with the following additional constraints:

* Each function takes an additional parameter, usually the last, which is another function
* That function parameter is applied at the end of the current function
* That function parameter is given as input what would be the output of the current function
* Larger problem is solved as a pipeline of functions, but where the next function to be applied is given as parameter to the current function

## Monadic Identity / Imperative functional style

* Existence of an abstraction to which values can be converted.
* This abstraction provides operations to (1) wrap around values, so that they become the abstraction; (2) bind itself to functions, so to establish sequences of functions; and (3) unwrap the value, so to examine the final result.
* Larger problem is solved as a pipeline of functions bound together, with unwrapping happening at the end.
* Particularly for The One style, the bind operation simply calls the given function, giving it the value that it holds, and holds on to the returned value.

# Object Oriented

## Things / Object-oriented style / The Kingdom of Nouns

* The larger problem is decomposed into 'things' that make sense for the problem domain
* Each 'thing' is a capsule of data that exposes procedures to the rest of the world
* Data is never accessed directly, only through these procedures
* Capsules can reappropriate procedures defined in other capsules

## Letterbox / Messaging style

* The larger problem is decomposed into 'things' that make sense for the problem domain
* Each 'thing' is a capsule of data that exposes one single procedure, namely the ability to receive and dispatch messages that are sent to it
* Message dispatch can result in sending the message to another capsule

## Closed Maps / Prototypes

* The larger problem is decomposed into 'things' that make sense for the problem domain
* Each 'thing' is a map from keys to values. Some values are procedures/functions.

## Abstract data types

* The larger problem is decomposed into 'abstract things' that make sense for the problem domain
* Each abstract thing is described by what operations the things of that abstraction can eventually do
* Concrete things are then bound, somehow, to the abstractions; mechanisms for doing that vary
* The rest of the application uses the things not by what they are but by they do in abstract

## Inversion of control / Callback heaven/hell

* Larger problem is decomposed into entities using some form of abstraction (objects, modules or similar)
* The entities are never called on directly for actions
* The entities provide interfaces for other entities to be able to register callbacks
* At certain points of the computation, the entities call on the other entities that have registered for callbacks

## Publish-Subscribe

* Larger problem is decomposed into entities using some form of abstraction (objects, modules or similar)
* The entities are never called on directly for actions
* Existence of an infrastructure for publishing and subscribing to events, aka the bulletin board
* Entities post event subscriptions (aka 'wanted') to the bulletin board and publish events (aka 'offered') to the bulletin board. the bulletin board does all the event management and distribution

# Reflection and Metaprogramming

## Introspective

* The problem is decomposed using some form of abstraction (procedures, functions, objects, etc.)
* The abstractions have access to information about themselves, although they cannot modify that information

## Reflective / Metaprogramming style

* The program has access to information about itself, i.e. introspection
* The program can modify itself -- adding more abstractions, variables, etc. at run-time

## Aspect-oriented style

* The problem is decomposed using some form of abstraction (procedures, functions, objects, etc.)
* Aspects of the problem are added to the main program without any edits to the source code of the abstractions. These side functions latch on the main abstractions by naming them, as in "I'm an aspect of foo (even though foo may not know it!)"

## Plugins

* The problem is decomposed using some form of abstraction (procedures, functions, objects, etc.)
* All or some of those abstractions are physically encapsulated into their own, usually pre-compiled, packages. Main program and each of the packages are compiled independently. These packages are loaded dynamically by the main program, usually in the beginning (but not necessarily).
* Main program uses functions/objects from the dynamically-loaded packages, without knowing which exact implementations will be used. New implementations can be used without having to adapt or recompile the main program.
* External specification of which packages to load. This can be done by a configuration file, path conventions, user input or other mechanisms for external specification of code to be linked at run time.

# Adversity

## Constructive / Defensive / Hopeful

* Every single procedure and function checks the sanity of its arguments and either returns something sensible when the arguments are unreasonable or assigns them reasonable values
* All code blocks check for possible errors and escape the block when things go wrong, setting the state to something reasonable

## Tantrum / Design by contract

* Every single procedure and function checks the sanity of its arguments and refuses to continue when the arguments are unreasonable
* All code blocks check for all possible errors, possibly print out context-specific messages when errors occur, and pass the errors up the function call chain

## Passive aggressive / Exception

* Every single procedure and function checks the sanity of its arguments and refuses to continue when the arguments are unreasonable, jumping out of the function
* When calling out other functions, program functions only check for errors if they are in a position to react meaningully
* Error handling occurs at higher levels of function call chains, wherever it is meaningul to do so

## Declared intentions / "You've been warned!"

* Existence of a run-time typechecker
* Procedures and functions declare what types of arguments they expect
* If callers send arguments of types that are't expected, the procedures/functions are not executed

## Quarantine / Monadic IO

* Core program functions have no side effects of any kind, including IO
* All IO actions must be contained in computation sequences that are clearly separated from the pure functions
* All sequences that have IO must be called from the main program

# Data-Centric

## Tabular / Relational

* The input data of the problem is modeled as entities with relations between them
* The data is placed in tables, with columns potentially cross-referencing data in other tables
* Existence of a relational query engine
* The problem is solved by issuing queries over the tabular data

## Spreadsheet / Dataflow / Active data

* The problem is modeled like a spreadsheet, with columns of data and formulas
* Some data depends on other data according to formulas. When data changes, the dependent data also changes automatically.

## Data streams / Dataflow / Data generators

* Data comes to functions in streams, rather than as a complete whole all at at once
* Functions are filters / transformers from one kind of data stream to another

# Concurrency

## Free agents / Active letterbox / Actors

Similar to the letterbox style, but where the 'things' have independent threads of execution. Constraints:

* The larger problem is decomposed into 'things' that make sense for the problem domain
* Each 'thing' has a queue meant for other \textit{things} to place messages in it
* Each 'thing' is a capsule of data that exposes only its ability to receive messages via the queue
* Each 'thing' has its own thread of execution independent of the others.

## Dataspaces

* Existence of one or more units that execute concurrently
* Existence of one or more data spaces where concurrent units store and retrieve data
* No direct data exchanges between the concurrent units, other than via the data spaces

## Map-reduce / Inverse multiplexer

* Input data is divided in chunks, similar to what an inverse multiplexer does to input signals
* A map function applies a given worker function to each chunk of data, potentially in parallel
* A reduce function takes the results of the many worker functions and recombines them into a coherent output

## Double Map-reduce / Hadoop style / Double inverse multiplexer

Very similar to style #30, but with an additional twist. Constraints:

* Input data is divided in chunks, similar to what an inverse multiplexer does to input signals
* A map function applies a given worker function to each chunk of data, potentially in parallel
* The results of the many worker functions are reshuffled in a way that allows for the reduce step to be also parallelized
* The reshuffled chunks of data are given as input to a second map function that takes a reducible function as input

# Interactivce

## Model/View/Controller

* The application is divided into three components: the model, the view and the controller. The model represents the application's data; the view represents a specific rendition of the data; the controller provides for input/output, for populating/updating the model and for invoking the right view.

## RESTful / Stateless Ping-Pong

REST = REpresentational State Transfer ([http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm))

REST is a style for network-based interactive applications that underlies the Web. The example here doesn't go over the network, but preserves the main contraints of REST, which are:

* Interactive: end-to-end between an active agent (e.g. a person) and a backend
* Separation between Client (user interface) and Server (data storage)
* Statelessness, as in client--stateless-server: every request from client to server must contain all the information necessary for the server to serve the request. The server cannot store context of the interaction. Session state is on the client.
* Uniform interface: resources that are created and retrieved, resource identifiers and hypermedia representation that is the engine of application state