Interactive programming

Daniel Szmulewicz

September 18, 2015

Introduction

A little quiz for starters

Pieter is looking at Mieke, and Mieke is looking at Jan. Pieter is married, but Jan is not. Is a married person looking at an unmarried person?

- A. Yes
- B. No
- C. There is not enough information to derive the correct answer.

Evolution

The human brain is wired to accept the first seemingly fit answer that comes to mind when considering a problem. It makes sense from an evolutionary point of view: heuristics speed up the process of finding a solution. Speed is important for survival.

Reasoning and Logic

> Psychologists concur with philosophers in the irrelevance of logic to everyday reasoning. Decades of work on Wason's selection task seem to confirm this. — Pascal Engels

Thinking, Fast and Slow

Following Daniel Kahneman's terminology, the brain works in two modes, system 1 and system 2. System 1 is the thinking that is fast, intuitive, and unaware of its workings. System 2 is the slow thinking able to correct system 1's mistakes. The problem is that we often fail to recognize when we need to switch gear and slow down.

So, programming are we?

Question: When we program, do we use system 1 or system 2? We use both. When we learn or face a difficult part of the program, we use system 2, but most of the time we'll try to maximize system 1 and proceed fast to use what we've learnt, what we already know.

Acceptance

> Civilization advances by extending the number of important operations which we can perform without thinking about them — Alfred North Whitehead

Epistemological pluralism

We need to be humble in the light of our shortcomings, and embrace any strategy that helps us exercice our capabilities. Constructionism. Seminal paper: Epistemological Pluralism and the Revaluation of the Concrete, by Seymour Papert and Sherry Turkle

Methodology

Desirable properties such as:

- testability
- reactivity
- incrementality
- immediacy

Why interactive programming?

- bottom-up programming
- domain exploration
- bricolage and prototyping
- Bret Victor
- ideal fit for innovation, learning and teaching

A definition

Interactive programming is the procedure of writing parts of a program while it is already active.

From within the environment provided by the REPL, you can define and redefine program elements such as variables, functions, classes, and methods; evaluate any Lisp expression; load files containing Lisp source code or compiled code; compile whole files or individual functions; enter the debugger; step through code; and inspect the state of individual Lisp objects.

How do you build a story for interactive programming?

- What properties must a programming language have to build a good story for interactive programming?

Consult a paper (from 1978): Programming in an Interactive Environment: the "Lisp" Experience by Erik Sandewall

- Bootstrapping
- Incrementality
- Procedure orientation
- Internal representation of programs
- Full checking capability
- Declaration-free kernel
- Data structures and database
- Defined I/O for data structure
- Handles and interactive control

The kernel of the programming system must contain the following programs:

- a parser
- a program-printer
- an interpreter and/or
- a compiler

The REPL

```
(loop (print (eval (read))))
```

Functionality of a Lisp REPL

- History of inputs and outputs.
- Variables for last result, last error (*1, *e).
- Help and documentation for commands. (`doc`, `source` in clojure.repl namespace)
- Variables to control the reader. (`*data-readers*`, `*default-data-reader-fn*`)
- Variables to control the printer. (`*print-length*`, `*print-level*`)

How is a Lisp REPL different from any other REPL?

- SO: Is Lisp the only language with REPL?
- Clearly not: `http://repl.it/`
- Yet, Lisp's story for interactive programming is unparalleled.

A REPL is not enough

> But for the true Lisp programming experience, you need an environment, such as SLIME, that lets you interact with Lisp both via the REPL and while editing source files. — **Peter Seibel**, Practical Common Lisp

> For instance, you don't want to have to cut and paste a function definition from a source file to the REPL or have to load a whole file just because you changed one function; your Lisp environment should let us evaluate or compile both individual expressions and whole files directly from your editor. — **Peter Seibel**, Practical Common Lisp

The environment

Types of environments

Lisp	Non-Lisp
SLIME (CL)	COLT (Actionscript)
Cider (Clojure	SuperCollider (C++)
Emacs (Lisp)	Squeak (Smalltalk)
Geiser (Scheme)	
Genera (Lisp Machine)	

Clojure

Henceforth, the environment will be known as Cider.

Restarting the runtime

You don't want to do that. Clojure needs to bootstrap itself inside the JVM each and every time.

Restarting the application

Ah. Let's see.

Reloading the namespace

Cojure built-in facilities: `(require ...  :reload)` and `(require ...  :reload-all)` Some problems.

- If you modify two namespaces which depend on each other, you must remember to reload them in the correct order to avoid compilation errors.
- If you remove definitions from a source file and then reload it, those definitions are still available in memory. If other code depends on those definitions, it will continue to work but will break the next time you restart the JVM.
- If the reloaded namespace contains defmulti, you must also reload all of the associated defmethod expressions.
- If the reloaded namespace contains defprotocol, you must also reload any records or types implementing that protocol and replace any existing instances of those records/types with new instances.

- If the reloaded namespace contains macros, you must also reload any namespaces which use those macros.
- If the running program contains functions which close over values in the reloaded namespace, those closed-over values are not updated. (This is common in web applications which construct the "handler stack" as a composition of functions.)

tools.namespace

Solved by tools.namespace single API call `refresh`. The `refresh` function will scan all the directories on the classpath for Clojure source files, read their ns declarations, build a graph of their dependencies, and load them in dependency order. (You can change the directories it scans with set-refresh-dirs.) But first, it will unload (remove) the namespaces that changed to clear out any old definitions.

```
user=> (require '[clojure.tools.namespace.repl :refer [refresh]])
user=> (refresh)
user=> (def my-app (start-my-app))

user=> (stop-my-app my-app)
user=> (refresh)
user=> (def my-app (start-my-app))
```

So, is that it?

- No global state. `refresh` will destroy Vars when it reloads the namespace (even `defonce`)
- Acquiring and releasing resources (sockets, files, database connections)

The Reloaded pattern

`tools.namespace` + Lifecycle protocol (Stuart Sierra) `http://thinkrelevance.com/blog/2013/06/04/clojure-workflow-reloaded`

component

`component` is a tiny Clojure framework for managing the lifecycle of software components which have runtime state.

system

A set of readymade components. The usual suspects.

Jetty	Datomic
HTTP kit	H2
Aleph	Monger
Sente	nREPL
Neo4j	Langohr
ElasticSearch	PostgreSQL
Immutant	Etsy

Leiningen

JVM instances proliferation. Examples: the REPL, the `cljs` build, the hot-reloading process (figwheel), preprocessors, watchers. . . .

In shell a:

```
$ lein repl :headless
```

In shell b:

```
$ lein trampoline cljsbuild repl-listen
```

Or:

```
$ lein figwheel
```

Boot

Single JVM instance.

```
(deftask dev
  "Run a restartable system in the Repl"
  []
  (comp
   (watch :verbose true)
   (reload) ; figwheel
   (cljs :source-map true) ; cljsbuild
   (repl :server true))) ; REPL
```

Anatomy of a system

```
(defsystem dev-system
  [:db (new-h2-database DEFAULT-MEM-SPEC)
   :web (new-web-server (Integer. (env :http-port)) app)])
```

Boot-system

```
(deftask dev
  "Run a restartable system in the Repl"
  []
  (comp
   (environ :env {:http-port "3025"
  :imdb-key "xxxxx-xxxx-xxxxx-xxxxx"})
   (watch :verbose true)
   (system :sys #'dev-system :auto-start true :hot-reload true :files ["handler.clj"])
   (reload) ; equivalent to figwheel
   (cljs :source-map true) ; equivalent to cljsbuild
   (repl :server true)))
```

`https://github.com/danielsz/system#boot-system`

CQFD

Your application updates itself automatically when editing your source code.

- `(require ... :reload)`
- `(refresh}` (via `(reloaded.repl/reset)`)

Reference

A history of interactive programming

`http://ecx.images-amazon.com/images/I/51qjJtCFxiL.jpg`

GUIs and interactive environments

`http://upload.wikimedia.org/wikipedia/en/d/d2/Symbolics-document-examiner.png` Garnet - a graphical toolkit for Lisp

Live coding

Live coding performance in Impromptu by Andrew Sorensen.

A REPL Everywhere

Full Clojure programming environment in Microsoft Excel `https://github.com/whamtet/Excel-REPL`

The ultimate interactive session