

Universidad de Buenos Aires

Facultad de Ingeniería

Presentación Trabajo Final:

Clojure IDE

**Tutora:** Rosita Wachenchauzer

**Alumno:** Juan Martín Muñoz Facorro

**Padrón N°**: 846722

Contenido

[Introducción 3](#_Toc331983481)

[¿Qué es Clojure? 3](#_Toc331983482)

[¿Por qué Clojure? 3](#_Toc331983483)

[¿Qué es un IDE? 3](#_Toc331983484)

[¿Por qué un IDE para Clojure? 4](#_Toc331983485)

[Desafíos 5](#_Toc331983486)

[Funcionalidades 6](#_Toc331983487)

[Bibliografía 8](#_Toc331983488)

# Introducción

Por medio del siguiente informe se pretende presentar una propuesta para un **Trabajo Final** de la carrera de **Ingeniería Informática** de la **Universidad de Buenos Aires**. Dicho **Trabajo Final** consiste en implementar un **IDE** (*Integrated Development Environment*) pensado principalmente para trabajar con el lenguaje **Clojure** (aunque extensible a otros lenguaje), que ofrezca una **interfaz** (UI) y **experiencia** **de usuario** (UX) accesible y que fomente la productividad en un entorno gráfico moderno y familiar.

# ¿Qué es Clojure?

**Clojure** es un dialecto de **Lisp**, el cual es a su vez un lenguaje de programación diseñado por John McCarthy en 1958 (McCarthy, 1996). Desde su creación en ese año han surgido un gran número de implementaciones de dicho lenguaje, siendo las más conocidas al día de hoy **Common Lisp** y **Scheme** (Seibel, 2009). Una de las razones por la cual existe un gran número de implementaciones de este lenguaje, es la simplicidad de su sintaxis, la cual se basa en la utilización de paréntesis y notación prefija (Steele & Gabriel, 1993) como elemento de expresión.

Lisp is not the right language for any particular problem. Rather, Lisp encourages one to attack a new problem by implementing new languages tailored to that problem.

—“Lisp: A Language for Stratified Design” (Abelson 1988)

**Lisp**, a pesar de ser un lenguaje con gran poder de expresión, por muchos años fue considerado como un lenguaje de uso académico y relacionado con la rama de investigación en Inteligencia Artificial (Steele & Gabriel, 1993). No obstante también se lo utilizó para el desarrollo de algunas aplicaciones comerciales como **Viaweb**, uno de los primeros sitios en ofrecer una plataforma web de e-commerce y que luego fue comprado por **Yahoo** (Graham, 2001).

**Clojure** fue creado por Rick Hickey y lanzado en 2007 (Hickey, 2008), fue diseñado para correr sobre la **JVM** (Java Virutal Machine) y soportar interoperabilidad con las librerías de existentes Java (Hickey, 2008), pero la comunidad open-source extendió las plataformas donde se puede utilizar este lenguaje creando soporte para el **CLR** (*Common Language Runtime*) de **Microsoft.NET** y un compilador a **JavaScript** llamado **ClojureScript**.

# ¿Por qué Clojure?

Desde su aparición en 2007, **Clojure** fue ganando adeptos en algunas comunidades de programación, tanto por su poder de expresión como por la posibilidad de reutilización de librerías existentes de Java.

La aparición de nuevas plataformas en donde se hizo posible la utilización de **Clojure** también aportó a la popularización de este lenguaje. En particular, la creación de **ClojureScript** abrió las puertas a la experimentación de **Clojure** en aplicaciones web del lado del cliente.

# ¿Qué es un IDE?

Un **IDE** es una aplicación que ofrece un conjunto de herramientas que facilita el desarrollo de aplicaciones informáticas de distintos tipos, en uno o varios lenguajes determinados. Existen hoy en día **IDE**s tanto de carácter propietario como open-source, siendo los más conocidos el **Microsoft Visual Studio**, utilizado para desarrolar en la plataforma **.NET**, y el **Eclipse** orientado principalmente a desarrollos en **Java**.

El objetivo de todo **IDE** es facilitar la tarea de crear software, de forma tal que se logre aumentar la productividad del usuario y minimizar las operaciones rutinarias o que no aportan valor al producto que se quiere generar.

# ¿Por qué un IDE para Clojure?

La documentación oficial de **Clojure** (Clojure, 2012) lista una seria de opciones para trabajar con este lenguaje. Estas son en su gran mayoría complementos para **IDE**s existentes y otras son editores de texto de consola típicamente utilizados por la comunidad de programación:

* **Counter-Clockwise**: complemento para **Eclipse**.
* **La Clojure**: complemento para **IntelliJ**.
* **Enclojure**: complemento para **Netbeans**.
* **Clooj**: editor básico de Clojure.
* **Emacs**.
* **Vim**.

Históricamente la comunidad de **Lisp** recomienda para programar en este lenguaje o cualquiera de sus dialectos (como lo es **Clojure**), alguna aplicación que pertenezca a la familia **Emacs**. En particular se recomienda **GNU Emacs**, la cual ofrece una alta flexibilidad por su capacidad de extender sus comandos utilizando **Emacs Lisp**, una variante de **Lisp**.

**Emacs**, a pesar de ser una herramienta muy poderosa, sufre la desventaja de ser muy poco amigable, lo que tiene como consecuencia una curva de aprendizaje muy lenta. Entre algunas de las cosas que producen esto, se puede mencionar el hecho que no ofrece una interfaz de usuario familiar, dado que muestra al usuario ventanas de puro texto sin incluir al mouse como un elemento de interacción. Sumado a esto, para poder manejar **Emacs** con naturalidad, se deben incorporar algunos conceptos especiales que esta aplicación utiliza para trabajar, como son los **buffers**, mientras que también se deben aprender algunas definiciones y nombres nuevos, como las operaciones de texto **kill** y **yank**, entre otras (Carper, 2010).

A pesar de apuntar a todos los **IDE**s en general, la siguiente imagen trata de captar la naturaleza de lo expuesto anteriormente (Stanislav, 2012).



**Chas Cemerick**, el autor de **Clojure Programming**, realizó por medio de su página una encuesta para la comunidad de **Clojure** en los últimos dos años (Emerick, Results from the State of Clojure, Summer 2010 Survey | cemerick, 2010) (Emerick, Results of the 2011 State of Clojure survey, 2011). Entre las preguntas existentes se encuentra la siguiente: ¿Qué ambientes de desarrollo usa para trabajar con Clojure? A pesar de que el ambiente de desarrollo con mayor porcentaje de utilización es **emacs**, el autor plantea lo siguiente:

I continue to maintain that broad acceptance and usage of Clojure will require that there be top-notch development environments for it that mere mortals can use and not be intimidated by…and IMO, while emacs is hugely capable, I think it falls down badly on a number of counts related to usability, community/ecosystem, and interoperability.

I’ll repeat again: I’d love to pay for a super-polished clojure dev env. I’ve got $500 (every two years, say) for whoever can make it happen. Seriously. There are comments in the raw survey results that indicate that others are ready to make a similar purchase. Pay attention if you’re a developer tools company, or aspire to have/be one.

Chas Emerick

Por lo expuesto anteriormente se considera un aporte de gran valor y a la vez un desafío, el desarrollo de un **IDE**, concebido desde el origen para ser utilizado con el lenguaje **Clojure** y extensible a otros lenguajes de programación; utilizando para su construcción el mismo lenguaje **Clojure**.

# Desafíos

**UI/UX**

Como se mencionó en una sección anterior, el principal objetivo de un **IDE** es darle al usuario una herramienta que facilite su trabajo y pueda concentrar sus esfuerzos en aquellas tareas que generan valor. Es por esto que la interfaz del usuario y la experiencia de éste con la aplicación son factores clave para el éxito del presente proyecto.

**Extensibilidad**

Si bien el lenguaje principal para el cual se piensa utilizar el **IDE** propuesto es **Clojure**, se cree que la posibilidad de extender o adaptar la herramienta a diferentes necesidades del usuario, es una característica deseable dado que sin ésta la aplicación tendría un carácter rígido que no se condice con la naturaleza del ámbito tecnológico. Esto también permite darle al usuario que lo desee una herramienta para la customización de su ambiente y herramientas de trabajo.

**Integración con Java**

**Uno de los más grandes atractivos de Clojure es que permite reutilizar librerías de Java, lo cual facilita la implementación de muchas soluciones, mediante la combinación del poder del lenguaje Clojure y la robustez de código probado y estable existente. Dada la gran interacción que se presenta entre estos dos lenguajes, la aplicación debe ser capaz de facilitar la integración del uso de Java dentro de Clojure, para que la utilización de uno en el otro sea lo más natural posible. Este es un objetivo difícil de lograr dado que para ello la aplicación no sólo debe ser capaz de manejar código de Clojure, sino también poseer la infraestructura para manipular clases y librerías de Java, con todo lo que ello implica.**

# Funcionalidades

A continuación se presentan dos listas de funcionalidades que tendrá el IDE propuesto. La primera consta de aquellas que no pueden faltar, dado que hacen a la esencia de una aplicación de este estilo y a los requerimientos para que el entorno de desarrollo sea efectivo. La segunda está compuesta por funcionalidades deseables, las cuales podrían llegar a incluirse como extensiones en el caso que se deseen.

Gran parte de los requerimientos de la siguiente lista fue tomada de la especificación que el autor antes mencionado (Emerick, The ideal Clojure development environment, 2010) hace en un artículo sobre las características principales que él piensa debería tener un **IDE** para **Clojure**.

**Requeridas**

* **Edición de Código**
  + Resaltado de código según sintaxis.
    - Palabras claves, delimitadores, comentarios, literales, construcciones especiales del lenguaje.
  + Seleccionar, copiar, pegar, cortar, buscar, etc.
  + Equilibrio de paréntesis, corchetes y llaves.
  + Equivalencia con ParEdit (comandos que facilitan la edición de Expresiones-S).
  + Paréntesis en modo Arco Iris.
  + Navegación de Expresiones-S.
  + Expansión de macros en contexto.
  + Auto-Completar código.
    - Visibilidad de las variables definidas localmente, las librerías externas referenciadas y las librerías del núcleo del lenguaje (clojure.core).
    - Java (interoperabilidad): utilizar los *types hints* y hacer lo posible para determinar los demás tipos.
    - Ofrecer documentación asociada con cada entidad mostrada.
* **Administración Proyectos y Dependencias**
  + Organización de archivos.
  + Compilación del código.
  + Configuración de ambiente de desarrollo.
    - Existencia de una configuración inicial básica.
    - Inclusión de algunas librerías estándar al trabajar desde una sesión REPL.
  + Manejo de dependencias (maven/leiningen).
* **Interfaz de Usuario**
  + Simplicidad.
  + Descubrimiento progresivo de las funcionalidades de la aplicación.
* **REPL (Read-Eval-Print-Loop)**
  + Soporte para múltiples REPL.
    - Procesos distintos de la aplicación principal.
    - REPL remotos.
  + Historial de ejecución en el REPL accesible.
    - Buscar.
    - Ejecutar nuevamente.
  + Funcionalidad de edición completa dentro del REPL (ver punto anterior).
  + Navegador de namespaces que se encuentran cargados en el REPL en tiempo de ejecución.
  + Verificar el estado o matar un REPL determinado.
  + Separación (opcional) de los streams de entrada, salida (\*out\*) y error (\*err\*).
  + Generación y configuración automática del *classpath* del REPL cargado, incluyendo:
    - El *path* a los archivos fuente del proyecto actual.
    - El *path* a los archivos fuente de los proyectos referenciados.
    - Todas las dependencias indicadas en cada proyecto.

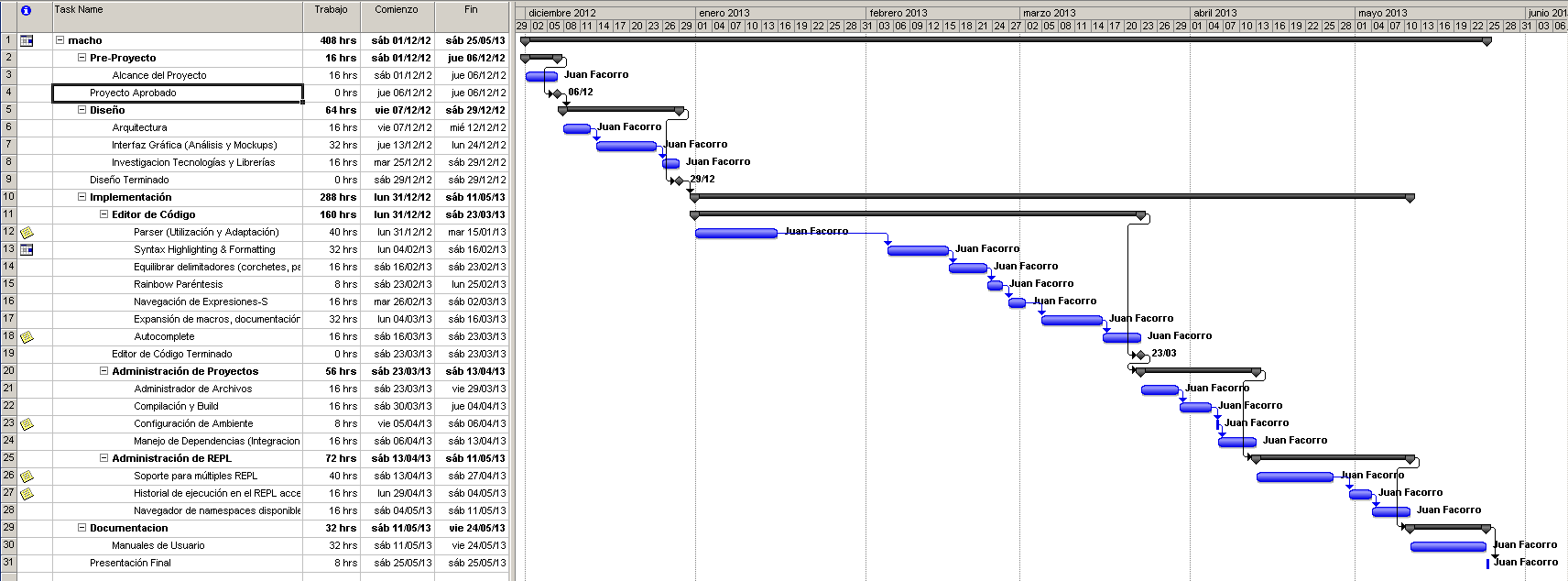
**Deseables**

* **Edición de Código**
  + Formato Automático Sintáctico: por ejemplo no aplicar las mismas reglas de formato para un *doc-string* que para el resto del código.
* **Soporte Integral de Desarrollo en Java**
  + Debugger.
* **Generación de Código**
  + deftype/defrecord/extend-type/gen-class/proxy
* **REPL**
  + Formato configurable de la salida.
  + Posibilidad de “imprimir” entidades además de texto, al estilo Mathematica u otras aplicaciones y lenguajes existentes.
* **Análisis Estático de Código**
  + Navegación de símbolos (por ej. “Ir a la declaración”).
  + Encontrar usos de un símbolo.
  + Estructura del archivo abierto.
  + Navegador de Namespaces con capacidad de búsqueda.

# Plan de Trabajo

A continuación se presenta un cronograma de las tareas a realizar junto con una estimación de las horas invertidas en cada una de las tareas.

El calendario se encuentra confeccionado teniendo en cuenta que se emplearán en promedio 18 horas semanales al desarrollo del proyecto.



# Bibliografía

Carper, B. (Junio de 2010). *Emacs isn't for everyone*. Obtenido de briancarpet.net: http://briancarper.net/blog/534/emacs-isnt-for-everyone

Clojure. (Mayo de 2012). *Getting Started*. Obtenido de Clojure Documentation: http://dev.clojure.org/display/doc/Getting+Started

Emerick, C. (2010). *Results from the State of Clojure, Summer 2010 Survey | cemerick*. Obtenido de cemerick: http://cemerick.com/2010/06/07/results-from-the-state-of-clojure-summer-2010-survey/

Emerick, C. (2010). *The ideal Clojure development environment*. Obtenido de cemerick: http://cemerick.com/ideal-clojure-development-environment/

Emerick, C. (2011). *Results of the 2011 State of Clojure survey*. Obtenido de cemerick2: http://cemerick.com/2011/07/11/results-of-the-2011-state-of-clojure-survey/

Graham, P. (Abril de 2001). *Beating the Averages*. Obtenido de Paul Graham: http://www.paulgraham.com/avg.html

Hickey, R. (10 de 2008). *Clojure*. Obtenido de Clojure.org: http://clojure.org/page/diff/home/42336501

McCarthy, J. (1996). *The Implementation of LISP*. Obtenido de John McCarthy website: http://www-formal.stanford.edu/jmc/history/lisp/node3.html

Seibel, P. (2009). 1. Introduction: Why Lisp? En P. Seibel, *Practical Common Lisp.*

Stanislav. (March de 2012). *Loper OS: Programmer's Editors, Illustrated.* Obtenido de Loper OS.

Steele, G., & Gabriel, R. P. (1993). *The Evolution of Lisp.*