Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Departamento de Computación - FCEyN - UBA

Lenguajes y Paradigmas de programación
Buenas Prácticas de Programación

Algoritmos y programas

- Aprendieron a especificar problemas
- El objetivo es ahora escribir un algoritmo que cumpla esa especificación
  - Secuencia de pasos que pueden llevarse a cabo mecánicamente
- Puede haber varios algoritmos que cumplan una misma especificación
- Una vez que se tiene el algoritmo, se escribe el programa
  - Expresión formal de un algoritmo
  - Lenguajes de programación
    - Sintaxis definida
    - Semántica definida
    - Qué hace una computadora cuando recibe ese programa
    - Qué especificaciones cumple
    - Ejemplos: Haskell, Python, C, C++, C#, Java, Smalltalk, Prolog, etc.

Lenguajes de programación

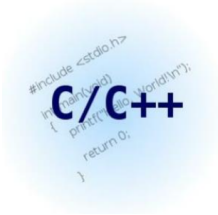
- En palabras simples, es el conjunto de instrucciones a través del cual los humanos interactúan con las computadoras.
- Permiten escribir programas que son ejecutados por computadoras.

Lenguajes de programación

No es tema de la materia... pero demos algún contexto por si se ponen a googlear...

- **Lenguaje Máquina**: son lenguajes que están expresados en lenguajes directamente inteligibles por la máquina, siendo sus instrucciones cadenas de 0 y 1.
- **Lenguaje de Bajo Nivel**: son lenguajes que dependen de una máquina (procesador) en particular (el más famoso probablemente sea Assembler)
- **Lenguaje de Alto Nivel** (en la materia usaremos algunos de estos): fueron diseñados para que las personas puedan escribir y entender más fácilmente los programas que escriben.

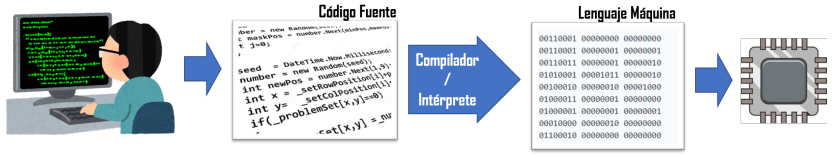
```
Push  a       ; a → pila
Push  b       ; b → pila
Load  (c),R1  ; c → R1
Mult  (S),R1  ; b*c → R1
Store R1,R2   ; R1 → R2
Add   (S),R1  ; a+b*c → R1
Store R1,(x)  ; R1 → x
Add   #3,R2   ; 3+b*c → R2
Store R2,(y)  ; R2 → y
```



Código fuente, compiladores, intérpretes...

No es tema de la materia... pero demos algún contexto por si se ponen a googlear...

- **Código Fuente**: es el programa escrito en un lenguaje de programación según sus reglas sintácticas y semánticas.
- **Compiladores e Intérpretes**: son programas *traductores* que toman un código fuente y generan otro programa en otro lenguaje, por lo general, lenguaje de máquina



IDE (Integrated Development Environment)

Para escribir y ejecutar un programa alcanza con tener:

- Un editor de texto para escribir programas (Ejemplos: notepad, notepad++, gedit, etc)
- Un compilador o intérprete (según el lenguaje a utilizar), para procesar y ejecutar el programa

Pero un mundo mejor es posible...

IDE (Integrated Development Environment)

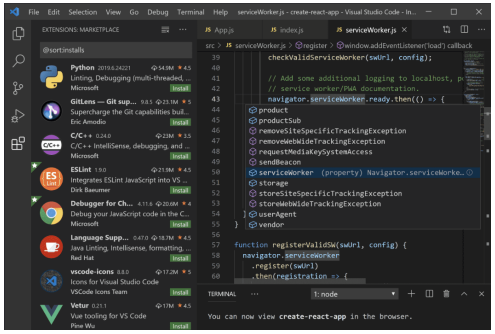
Ventajas de utilizar algún IDE

- Un editor está orientado a editar archivos mientras que un IDE está orientado a trabajar con proyectos, que tienen un conjunto de archivos.
- Integran un editor con otras herramientas útiles para los desarrolladores:
  - Los IDEs son capaces de verificar la sintaxis de los programas escritos (*linters*)
  - Generar vistas previas (*previews*) de cierto tipo de archivos (ej, archivos HTML para desarrollos web)
  - Suelen tener herramientas integradas (por ejemplo, el Android Studio tiene emuladores integrados)
  - Se pueden especializar según cada lenguaje particular
  - Permiten hacer depuración o *debugging*!

IDE (Integrated Development Environment)

Algunos IDEs:

- Visual Studio (https://visualstudio.microsoft.com/es/)
- Eclipse (https://www.eclipse.org/)
- IntelliJ IDEA (https://www.jetbrains.com/es-es/idea/)
- Visual Code o Visual Studio Code (https://code.visualstudio.com/)
  - Es un editor que se "convierte" en IDE mediante *extensions*.
  - Lo utilizaremos para programar en Haskell y Python.



Paradigmas

Existen diversos paradigmas de programación. Comunmente se los divide en dos grandes grupos:

- Programación Declarativa
  - Son lenguajes donde el programador le indicará a la máquina lo que quiere hacer y el resultado que desea, pero no necesariamente el cómo hacerlo.
- Programación Imperativa
  - Son lenguajes en los que el programador debe precisarle a la máquina de forma exacta el proceso que quiere realizar.

Paradigmas

Cada grupo, se especializa según diferentes características

- Programación Declarativa: describe un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.
  - Paradigma Lógico: los programas están construídos únicamente por expresiones lógicas (es decir, son Verdaderas o Falsas).
  - Paradigma Funcional: está basado en el modelo matemático de composición funcional. El resultado de un cálculo es la entrada del siguente, y así sucesivamente hasta que una composición produce el valor deseado.

Paradigmas

Cada grupo, se especializa según diferentes características

- Programación Imperativa: describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa.
  - Paradigma Estructurado: los programas se dividen en bloques (procedimientos y funciones), que pueden o no comunicarse entre sí. Existen estructuras de control, que dirigen el flujo de ejecución: IF, GO TO, Ciclos, etc.
  - Paradigma Orientado a Objetos: se basa en la idea de encapsular estado y comportamiento en objetos. Los objetos son entidades que se comunican entre sí por medio de mensajes.

Paradigmas

Cada grupo, se especializa según diferentes características

- Programación Declarativa
  - Paradigma Lógico: PROLOG
  - Paradigma Funcional: LISP, GOFER, HASKELL.
- Programación Imperativa
  - Paradigma Estructurado: PASCAL, C, FORTRAN, FOX, COBOL
  - Paradigma Orientado a Objetos: SMALLTALK
- Lenguajes multiparadigma: lenguajes que soportan más de un paradigma de programación.
  - JAVA, PYTHON, .NET, PHP

Paradigmas

Dado dos números, determinar si el segundo es el doble que el primero...

- Prolog:

```
% La siguiente regla es verdadera si X es el doble que Y
es_doble(X, Y) :-
    X is 2*Y.
```

- Haskell:

```
esDoble :: Integer -> Integer -> Bool
esDoble x y = x == 2*y -- Verificamos si x es igual al doble de y
```

- Python:

```
def esDoble(x: int, y: int) -> bool:
    if(x == 2*y):
        return True
    else:
        return False
```


Paradigmas

En la materia resolveremos (programaremos) problemas utilizando estos dos paradigmas:

- Paradigma Funcional
  - Utilizaremos Haskell
- Paradigma Imperativo
  - Utilizaremos Python

Buenas prácticas de programación

El código nunca desaparecerá, ya que representa los detalles de los requisitos. En algún nivel, dichos detalles no se pueden ignorar ni abstraer; deben especificarse, y para especificar requisitos de forma que un equipo pueda ejecutarlos se necesita la programación. Dicha especificación es el código.

(...) el código es básicamente el lenguaje en el que expresamos los requisitos en última instancia. Podemos crear lenguajes que se asemejen a dichos requisitos. Podemos crear herramientas que nos permitan analizar y combinar dichos requisitos en estructuras formales, pero nunca eliminaremos la precisión necesaria; por ello, siempre habrá código.

Robert Martin, 2009.

Buenas prácticas de programación

Sé de una empresa que, a finales de la década de 1980, creó una magnífica aplicación, muy popular y que muchos profesionales compraron y utilizaron. Pero los ciclos de publicación empezaron a distanciarse. No se corrigieron los errores entre una versión y la siguiente. Crecieron los tiempos de carga y aumentaron los fallos. Todavía recuerdo el día en que apagué el producto y nunca más lo volví a usar. Poco después, la empresa desapareció.

Dos décadas después conocí a uno de los empleados de la empresa y le pregunté sobre lo que había pasado. La respuesta confirmó mis temores. Habían comercializado el producto antes de tiempo con graves fallos en el código. Al añadir nuevas funciones, el código empeoró hasta que ya no pudieron controlarlo. El código incorrecto fue el motivo del fin de la empresa.

Robert Martin, 2009.

Utilizar nombres declarativos

- Usar nombres que revelen la intención de los elementos nombrados. El nombre de una variable/función debería decir todo lo que hay que saber sobre la variable/función!

  1. Los nombres deben referirse a conceptos del dominio del problema.
  2. Una excepción suelen ser las variables con scopes pequeños. Es habitual usar **i**, **j** y **k** para las variables de control de los ciclos.
  3. Si es complicado decidirse por un nombre o un nombre no parece natural, quizás es porque esa variable o función no representa un concepto claro del problema a resolver.

- Evitar la desinformación!

  1. Llamar **hp** a la "hipotenusa" es una mala idea.
  2. Tener variables llamadas **cantidadDeElementosEnElEjeXSinMarcar** y **cantidadDeElementosEnElEjeYSinMarcar** no es buena idea.

Utilizar nombres declarativos

- Usar nombres pronunciables! No es buena idea tener una variable llamada **cdcptdc** para representar la "cantidad de cuentas por tipo de cliente".
- Se debe tener un nombre por concepto. No tener funciones llamadas **grabar**, **guardar** y **registrar**.
- Los nombres de las funciones deben representar el concepto calculado, o la acción realizada en el caso de las funciones **que no devuelvan un resultado**.
- No hacer chistes!

No hacer chistes

- Why does `man` print "gimme gimme gimme" at 00:30?
- *We've noticed that some of our automatic tests fail when they run at 00:30 but work fine the rest of the day. They fail with the message "gimme gimme gimme" in stderr, which wasn't expected. Why are we getting this output?*
- Source: `https://unix.stackexchange.com/questions/405783/why-does-man-print-gimme-gimme-gimme-at-0030`

```
src/man.c-1167- if (first_arg == argc) {
src/man.c-1168-   /*
http://twitter.com/#!/marnanel/status/13228055719011942 4 */
src/man.c-1169-   time_t now = time (NULL);
src/man.c-1170-   struct tm *localnow = localtime (&now);
src/man.c-1171-   if (localnow &&
src/man.c-1172-       localnow->tm_hour == 0 && localnow->tm_min == 30)
src/man.c-1173:     fprintf (stderr, "gimme gimme gimme\n");
```

I have contacted RHEL support about this issue.

The string comes from well known ABBA song Gimme! Gimme! Gimme! (A Man After Midnight).

The developer of the man-db, Colin Watson, decided that there was enough fun and the story won't get forgotten and removed the easter egg completely.

Utilizar nombres declarativos

Ambos programas son el mismo... ¿Cuál se lee más claro?

```
int x = 0;
vector<double> y;
...
for(int i=0;i<=4;i=i+1) {
 x = x + y[i];
}

int totalAdeudado = 0;
vector<double> deudas;
...
for(int i=0;i<=conceptos;i=i+1) {
 totalAdeudado = totalAdeudado + deudas[i];
}
```

Indentación (sangrado)

En algunos lenguajes esto NO SÓLO es estético... sino obligatorio

```
int main () {
int i, j;
for (i = 0; i ≤ 10; i++){
for (j = 0; j ≤ 10; j++){
cout ≪ i ≪ " x  " ≪  j ≪ " = " ≪ i*j;
}
}
return 0;
}
```

```
int main () {
  int i, j;
  for (i = 0; i ≤ 10; i++){
    for (j = 0; j ≤ 10; j++){
      cout ≪ i ≪ " x  " ≪  j ≪ " = " ≪ i*j;
    }
  }
  return 0;
}
```


Code styles

Python, Haskell, Java, etc: cada lenguaje tiene sus propias guías de estilo

- Existen distintos estilos de código para C++
- Ejemplos:
  - Google
  - GNU
  - Stroustup
  - LLVM
  - ...etc
- Se puede configurar la IDE para que re-formatee respetando el estilo elegido.
- En el caso de equipos con más de 1 desarrollador, es importante que todos estén alineados y utilizando el mismo estilo!

Ejemplo: GNU C++ Code Style

```
int sumIndexes (int n)
{
  int s = 0;
  int i = 1;
  while (i ≤ n)
    {
       s += i;
       i++;
    }
  return s;
}
```


Ejemplo: Google C++ Code Style

```
int sumIndexes(int n) {
  int s = 0;
  int i = 1;
  while (i ≤ n) {
    s += i;
    i++;
  }
  return s;
}
```

Comentarios

- Además de escribir comandos, los lenguajes de programación permiten escribir comentarios.
- Tipos de comentarios en C++:
  - Comentario de línea: `// ...`
  - Comentario de bloque: `/* ... */`
- Tipos de comentarios en Python:
  - Comentario de línea: `# ...`
  - Comentario de bloque: `""" ... """`
- Tipos de comentarios en Haskell:
  - Comentario de línea: `-- ...`
  - Comentario de bloque: `{- ... -}`
- Es importante hacer un uso adecuado de los comentarios, para que no oscurezcan el código.

Comentarios

- Los comentarios no arreglan código de mala calidad! En lugar de comentar el código, hay que clarificarlo.
- Es importante expresar las ideas en el código, y no en los comentarios.
- Los mejores comentarios son los conceptos que no se pueden escribir en el lenguaje de programación utilizado:
  1. Explicar la intención del programador.
  2. Explicitar precondiciones o suposiciones.
  3. Clarificar código que a primera vista puede no ser claro.

Comentarios: Malos usos

- Ejemplo 1:

```
/**
  * Funcion que toma dos enteros y devuelve un float.
  */
float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

```
while (x<y) { // itero hasta que x es mayor o igual que y
  o ..
}
```

- Ejemplo 3:

```
cout << y; // Imprimo el valor de y
```


Comentarios: Buenos usos

- Ejemplo 1:

```
/**
  * Computa el modulo del vector de a partir de dos coordenadas.
  * Las coordenadas deben ser numeros no negativos.
  * Si la precondicion no se cumple, retorna -1.
  */
float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

```
// guardamos los Ids en un vector porque no cambian
// durante el algoritmo
vector<s> clientIds;
```

- Ejemplo 3:

```
// el llamado a f se hace siempre con y>0
x = f(y);
```

Sobre las funciones auxiliares

A la hora de especificar... vimos que partir el problema en problemas pequeños siempre es una buena estrategia. A la hora de programar, también es así.

- Las funciones deben ser pequeñas! Una función con demasiado código es difícil de entender y mantener.
  1. Encapsular comportamiento dentro de funciones auxiliares!
- **Regla fundamental.** Cada función debe ...
  1. ... hacer sólo una cosa,
  2. ... hacerla bien, y
  3. ... ser el único componente del programa encargado de esa tarea particular.

Cyclomatic Complexity

- Es una métrica que se utiliza para aproximar la dificultad para testear un programa
- Se calcula a partir del control-flow graph (en unas semanas hablaremos de esto!)
- Hay herramientas que la calculan automáticamente
- Se puede fijar un máximo por proyecto que todos deben respetar

Cognitive Complexity

- Es una métrica que se utiliza para aproximar la dificultad para que un ser humano entienda un programa
- Idea:
  - Code is not considered more complex when it uses shorthand that the language provides for collapsing multiple statements into one
  - Code is considered more complex for each "break in the linear flow of the code"
  - Code is considered more complex when "flow breaking structures are nested"
- Como con la cyclomatic complexity, se puede computar automáticamente a partir del control-flow graph
- Existen herramientas integradas que lo analizan y emiten *warnings* si se ha roto alguna regla.

Naming Conventions - Convenciones de nombre

- Indican que reglas hay que seguir para nombrar variables, constantes, funciones, archivos, tipos de datos, etc.
- Idea: evitar uso incosistente
  - `firstindexwithvalue`
  - `first_index_with_value`
  - `firstIndexWithValue`
  - `FIRST_INDEX_WITH_VALUE`
  - `FirstIndexWithValue`
- Cada lenguaje suele tener su propia convención. Oportunamente veremos las convenciones propias de Haskell y Python.

Formato vertical

- Los archivos del proyecto no deben ser demasiado grandes!
- Conceptos relacionados deben aparecer verticalmente cerca en los archivos.
- Las funciones más importantes deben estar en la parte superior, seguidas de las funciones auxiliares.
    1. Siempre la función llamada debe estar debajo de la función llamadora.
    2. Las funciones menos importantes deben estar en la parte inferior del archivo.
- Se suele equiparar un archivo de código con un artículo periodístico. Debemos poder interrumpir la lectura en cualquier momento, teniendo información acorde con la lectura realizada.

Modularización

- Cada archivo del proyecto debe tener código homogéneo, y relacionado con un concepto o grupo de conceptos coherentes.
- Los nombres de los archivos también deben ser representativos!
- Muchos lenguajes de programación dan facilidades para organizar archivos en paquetes (o conceptos similares), para tener una organización en más de un nivel.

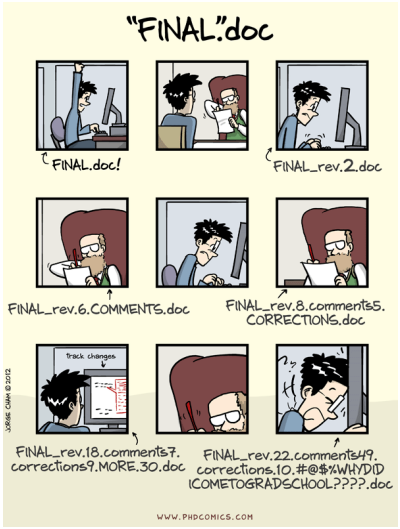
Modularización

- **Single responsibility principle:** Cada función debe tener un único motivo de cambio.
    1. Interfaz de usuario.
    2. Tecnología para la interfaz de usuario.
    3. Lógica de negocio.
    4. Consideraciones sobre los algoritmos.
    5. Almacenamiento permanente.
    6. ...
- El código responsable de cada uno de estos aspectos debe estar separado del resto!

Al momento de escribir código ...

- **Desarrollo incremental:** Escribir bloques pequeños y testeables de funcionalidad, teniendo en todo momento una aplicación (parcialmente) funcional.
    1. Esto implica conocer en todo momento el objetivo del código que estamos escribiendo!
    2. Si se usa un repositorio de control de versiones, el próximo commit debe estar bien definido.
- No incorporar expectativas en las estimaciones. Si vemos que los plazos no se van a poder cumplir, reaccionar rápidamente.
- Para el código más crítico, recurrir a pair programming: dos personas juntas sobre una única computadora.
- Descansar! Es importante destinar tiempo de descanso entre las sesiones de código.

Control de versiones



Version Control Systems (CVSs)

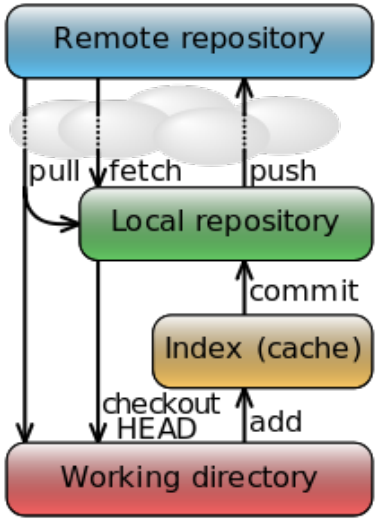
- Permite organizar el trabajo en equipo
- Guarda un historial de versiones de los distintos archivos que se usaron
- Existen distintas aplicaciones: svn, cvs, hg, git

Ejemplo: Git

- Sistema de control de versiones distribuido, orientado a repositorios y con énfasis en la eficiencia.
  1. Se tiene un servidor que permite el intercambio de los repositorios entre los usuarios.
  2. Cada usuario tiene una copia local del repositorio completo.
- Acciones: checkout, add, remove, commit, push, pull, status

Git: Workflow

Git básico: pull, push, commit, checkout...



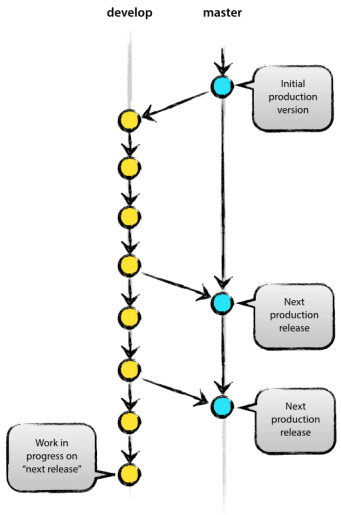
Otros conceptos

Git básico: branches y tags

- **Tag:** Nombre asignado a una versión particular, habitualmente para *releases* de versiones a usuarios.
- **Branch:** Línea paralela de desarrollo, para corregir un bug, trabajar en una nueva versión o experimentar con el código.
  - Master
  - Develop
  - Hotfixes

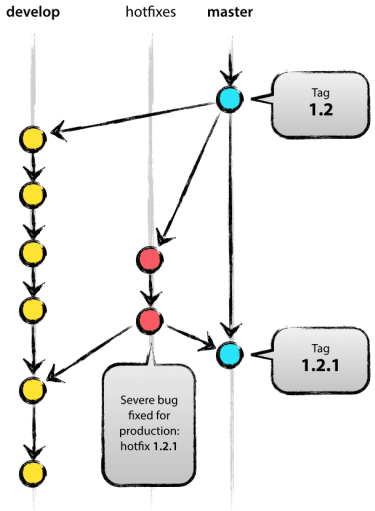
Master/Main-develop

Convenciones



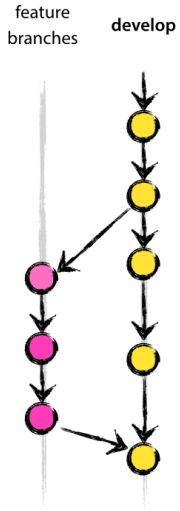
Hotfixes

Convenciones

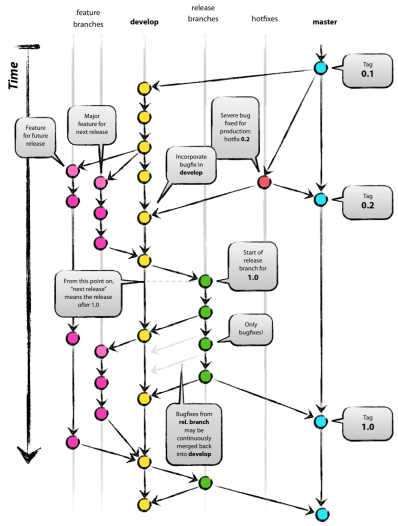


Nuevas features

Convenciones



Todo junto...



Consejos

- Hacer commits pequeños y puntuales, con la mayor frecuencia posible.
- Mantener actualizada la copia local del repositorio, para estar sincronizados con el resto del equipo.
- Commitear los archivos fuente, nunca los archivos derivados!
- Manejar inmediatamente los conflictos.

Un ejemplo

En el repositorio está toda la historia de lo que pasó con cada línea de código...

Links útiles

- **Repos hosts**
  - Bitbucket: `https://bitbucket.org`
  - GitHub: `https://github.com`
  - GitLab: `https://gitlab.com`
  - GitLab Exactas: `https://git.exactas.uba.ar`
- **Bibliografía**
  - **Git - la guía sencilla**: `http://rogerdudler.github.io/git-guide/index.es.html`
  - **Pro Git book**: `https://git-scm.com/book/en/v2`
  - **Try git**: `https://try.github.io`

Integración Continua (CI)

Muy por fuera del alcance de la materia... pero ya que estamos...

- Cada vez que hay una modificación (i.e. push) en el repositorio el servidor de integración:
  1. Descarga la última versión
  2. Compila el código
  3. Ejecuta el test suite
- Reporta a los desarrolladores si falló la compilación o el test suite.
- Antes de continuar, se deben *solucionar* todos los problemas de integración.
- El test suite puede ser distinto del test suite que corrió localmente el programador (i.e. más costoso)
- Evita acumular mucha deuda de integración

Integración Continua (CI)

Muy por fuera del alcance de la materia... pero ya que estamos...

- Servidores:
  - Jenkins (`https://jenkins.io/`) -open source
- Servicios Cloud:
  - CircleCI (`https://circleci.com/`)
  - TravisCI (`https://travis-ci.org/`) - gratuito para OSS.