Leonardo Larrosa

Javascript / Typescript Lead Developer

Front-End / Full-Stack

Digital Profile https://leonardolarsan.github.io/CV

Contenidos

Perfil

Roles

- Javascript/Typescript Full-Stack/Front-End Developer
- React Developer
- Node.js Developer
- Vue Developer
- React Native Developer
- Ionic Vue Developer

Aptitudes

- Destacada experiencia en el análisis, diseño y desarrollo de aplicaciones Web.
- Búsqueda constante de nuevas tecnologías.

- Gran capacidad de anticipación de problemas y resolución de los mismos.
- Codificación legible, escalable, reutilizable y eficiente.
- Fortalezas para organizar y estimar tiempos de trabajo.
- Orientado a resultados y a la toma de decisiones oportunas.
- Habilidades para interactuar con los clientes y usuarios.
- Facilidad para transmitir conocimiento y fomentar el trabajo de equipo.
- Implementación de prácticas modernas y las últimas tendencias de desarrollo Web.
- Ideas y opiniones orientadas a evitar contraer deuda técnica a largo plazo.
- Creación de pruebas de unidad y de integración.
- Desarrollo de herramientas de automatización para las tareas diarias.

Conocimientos y Habilidades

Lenguajes de Programación

- Javascript
- TypesScript
- HTML5-CSS3 Responsive

React

- React.js
- react-create-app
- React Native
- Redux
- Mobx
- Class Components
- Hooks Components
- Higher-Order Components
- Styled Components
- SASS
- Next
- Axios
- Mocha
- Chai
- Jest
- Enzyme
- React Testing Library
- Material UI
- Native Base

Vue

- Vue.js
- vue-cli
- Vuex
- Vue.observable()
- Styled Components
- Tailwind CSS
- Post CSS
- SASS
- Directives
- Vue PWA
- Capacitor
- Nuxt
- Native Script Vue
- Ionic Vue
- Vuetify
- BootstrapVue
- Element
- Framework7 Vue
- Axios
- Vue Test Library

Node

- Express.js
- express-generator
- Nest
- Micro.js
- Adonis
- Mongoose
- Socket.io
- Open Api
- JWT
- Mocha
- Chai

Base de Datos

- MongoDB
- IndexedDB
- Redis
- CouchDB
- PouchDB
- MySQL
- MariaDB
- SQLServer

Test E2E

- Cypress

Prácticas y filosofías de desarrollo

- Clean Code
- TDD
- SOLID
- KISS
- DRY
- GIT Flow
- LEAN

Herramientas de desarrollo

- Visual Studio Code
- GIT
- GIT Kraken
- SQLectron
- MySQL Workbench
- Data Grip
- Robomongo
- Robo 3T
- Postman

Arquitecturas de programación y patrones de diseño

- MVC
- MVP
- MVVM
- Factory
- Flux
- Reducer
- Clase
- Observer
- DAO
- Services
- Repoitory
- Container
- Component

Metodologías Ágiles, herramientas de Análisis y organización

- Star UML
- Kanban
- Scrum
- Scrumban
- Slack
- Trello
- Jira

Diseño Gráfico

- Gimp
- Inkscape
- Krita

Sistemas Operativos

- Linux(Manjaro)
- Windows

Ofimática

- Libre Office
- Microsoft Office

Idiomas

- Inglés Técnico
- Español Materno

Datos

Nombre Completo:

Leonardo Gabriel Larrosa Sánchez

Fecha Nacimiento:

15-10-1988

Nacionalidad:

Argentino

Estado Civil:

Soltero

Dirección:

Suarez 1■■■ 4to 11, La Boca, Ciudad de Buenos Aires

CUIL:

■■-■■■■■■■■-■

Teléfono:

11-2338-0144

Email:

leogab.larsan@gmail.com

Linkedin:

https://www.linkedin.com/in/leonardolarrosa/

Educación

Udemy

Vue.js

25/12/2017 - 14/1/2018

- Vue.js
- Vue Instance
- data
- methods
- computed and filter
- Directives
- Transitions
- Components
- Props
- component lifecycle
- vue-router
- vue-cli
- Vue2

Udemy

React.js

05/11/2017 - 20/12/2017

- React.js
- Props
- State and Lifecycle
- JSX
- Stateful and Stateless
- Components
- React Router
- Wrap Components
- Layout Components
- HOC Components
- Redux
- CSS
- Animations width dinamic classNames
- Axios
- webpack
- create-react-app
- React Native

UDACITY

Análisis y Gestión de Proyectos

02/2016 - 04/2016

- Kanban
- Scrum
- XP
- Entrevistas
- UML
- Microsoft Project

Centro de formación profesional n° 27 Luz y Fuerza

Desarrollador Stack MEAN

04/2015 – 11/2015

- Javascript
- jQuery
- AJAX
- JSON
- Mustache.js
- Angular.js
- mongoDB
- Express.js

Centro de formación profesional n° 27 Luz y Fuerza

Desarrollador JAVA

04/2015 – 11/2015

- NetBean
- Desarrollo JAVA para Escritorio

- Desarrolo JAVA con Base de Datos utilizando el patron DAO
- Servicio Web Java (JSP)

Centro de formación profesional Nº 8 SMATA

Diseñador y Desarrollador Web Full-Stack

04/2014 – 11/2014

- HTML5CSS3: responsive and animation
- GIMPHTML5-CANVASHTML5-SVG
- XAMPP
- .htacces (URL amigables)
- PHP (Estructurado y Modular)
- PHP POO (orientado a objetos)
- PHP-POO-MySQL
- Javascript
- jQuery
- Ajax
- JSON
- WebSocket

Centro de formación profesional Nº 8 SMATA

Arquitectura, Construcción y Administración de Base de Datos Relacionales

04/2012 - 06/2013

- Teoría de Base de Datos Relacionales
- MySQL
- OracleDB
- PL/SQL
- Arquitectura y construcción de Base de Datos
- Administración de Base de Datos

Centro de formación profesional Nº 8 SMATA

Analista - Programador Java

04/2012 – 11/2012

- Introducción a la programación con Python
- Programación estructurada y modular con Python
- Teoría de objetos: clases, atributos, métodos y herencias con Java
- Lenguaje de modelado unificado UML con StarUML
- Organización de proyectos con ProjectLibre

Centro de formación profesional Nº 8 SMATA

Operador de Imágenes Digitales

04/2012 – 11/2012

- Adobe Photoshop
- Adobe Ilustrator
- CorelDraw
- Corel Photo-Paint

Proyecto Programar.org

Programador Web

04/2011 - 11/2011

- Notepad++
- HTML4
- CSS3
- XAMPP
- PHP
- .htaccess
- Javascript

- Ajax
- MySQL
- FileZilla
- Ubuntu
- LAMPP
- Inglés Técnico
- Scrum
- Organización de Proyectos

Escuela Comercial Nº4

Perito Mercartil

04/2001 - 11/2006

- Contabilidad
- Administración de Empresas
- Microeconomia
- Macroeconomía

Experiencia Laboral

Siembro (10/2020-actual)

Front-End Team Leader

Liderazgo y desarrollo de las nuevas aplicaciones Back-Office y Front-Office utilizando buenas practicas, arquitecturas limpias y patrones de diseños modernos.

- React.js
- Ionic React
- Styled Components
- React Hooks
- React Test Library
- Axios
- Typescript

Digital House (6/2019-10/2020)

Front-End React/Vue Javascript/Typescript

Desarrollo y mantenimiento de Playground, una Web App para la educación digital, usando las siguientes tecnologias: - React.js - React Hooks - Styled Components - Redux - Ajax - SASS
- Monaco, - Jest y Enzyme

Mantenimiento y Desarrollo de la Web de Digital House: - Nuxt.js - Vue - Vuex - SASS - Vue Test Utils

Cablevisión Flow (11/2018-6/2019)

Front-End React Javascript

Desarrollo de Flow Web Client, la famosa Web App de TV digital, y Radio flow, una App para Smart TV que permite sintonizar y escuchar radios, usando las siguientes tecnologias: - React.js - Javascript - Mobx - React Hooks - SASS - Mocha - Chai - Jest - Enzyme y React Test Library

NCR (10/2016-11/2018)

Full-Stack Javascript/Typescript

Análisis, diseño y desarrollo de aplicaciones web y móviles para uso interno de la corporación. Las herramientas empleadas fueron: - Javascript - Node.js - Express.js - PM2 - GIT - React.js - React Native - Typescript - Vue.js - Material UI - Vue Material - Native Base - Bootstrap - MongoDB - SQLServer - IndexedDB - WebSocket - Ajax - Fetch y Axios

FreeLancer (11/2011-09/2016)

Full-Stack PHP-MYSQL-Angular.js

Análisis, diseño y desarrollo de aplicaciones Web y móviles en un equipo freelance de 5 personas, el cuál fundé, lideré y participé durante más de 4 años utilizando las siguientes tecnologías:

- PHP5-MySQL
- HTML-CSS
- Javascript Vanilla

- JQuery
- Phonegap
- NW.js(Node Web Kit)
- Angular.js

Sobre mi

Breve descripción (de mi mismo)

Argentino, 32 años. Vivo el día a día intentando ser bueno en mi trabajo y con los demás. Convivo con mi pareja desde hace más de 7 años, y con mi gato siamés de más de 10 años de edad.

En mi tiempo libre, cuando no estoy en mi computadora trabajando, sigo en la computadora, pero estoy jugando juegos online, diseñando planos de casas 3D o mirando alguna serie o documental de ciencia, arquitectura, medicina, etc.

Anhelo tener una casa propia y formar una familia, sueño que es muy difícil de alcanzar. Vivo de alquiler continuamente, no pudiendo construir una historia en un solo lugar, porque soy inquilino y mudarme es algo normal en mi vida.

Como buen milenial, entiendo lo que significa la "modernidad líquida" en la que vivimos, donde todo es incierto y temporal. Abrazo los cambios. Dejo de lado una vida monótona pero inestable a largo plazo. Sacrifico mi presente trabajando fuertemente, a cambio de la estabilidad que pueda obtener en el futuro.

Descripción profesional

Desarrollador y líder técnico full stack Javascript y Typescriopt, con más de 10 años de experiencia. Mi Stack principal es Node.js con Typescript, React, SASS, Ionic, Vue, Express, Fastify, Express, mongoDB, Redis, MySQL, Jest y Cypress, PM2 y Docker. Tengo mayor experiencia en el Front-end. Hago énfasis en la calidad del trabajo, en su planificación, organización, resolución, en la toma de decisiones y en el liderazgo.

Reflexiones y lecciones que han marcado mi evolución como desarrollador

¿Por qué apostar a un código de alta calidad y no a la productividad inmediata?

A menudo vemos que los desarrolladores son presionados para que aumenten su productividad con fechas de entrega límite que son muy exigentes, lo que provoca que estos caigan en malas prácticas de desarrollo que arruinan la calidad del código. Frente a esta situación, las soluciones rápidas, y en el momento, son fáciles de implementar cuando los proyectos recién comienzan, pero cuando estos aumentan de tamaño, y se va agregando mayor número de características, estas van aumentando en complejidad, a tal punto que se vuelve demasiado difícil para un desarrollador poder mantener esos proyectos cuando el código escrito es ilegible, desordenado e impredecible.

Este problema es conocido como la tan temida "deuda técnica" y, a pesar de que es un tema muy conocido en el ámbito de desarrollo de software, muchos líderes técnicos y CTOs subestiman enormemente este problema. Ellos piensan que las soluciones o desarrollos rápidos, centrados únicamente en la funcionalidad y no en la calidad, son sinónimo de alta productividad, ya que ellos pueden hacer entregas rápidas de funcionalidades nuevas a los clientes. Sin embargo, cuando el código de baja calidad aumenta de tamaño se vuelve exponencialmente más difícil de mantener, y esto se traduce en una perdida de productividad en el sector de desarrollo, y esto intentan resolverlo presionando aun más a los desarrolladores para que alcancen los niveles de productividad que tenían al principio de los proyectos, cuando aun eran mantenibles, de manera que los desarrolladores terminan estresados, desmoralizados y desgastados mentalmente. Este ritmo de trabajo solo puede ser sostenido por los desarrolladores por un periodo limitado, luego de ese tiempo bajan considerablemente su productividad al encontrarse padeciendo el síndrome de burnout.

Intentar contratar desarrolladores nuevos para resolver esta situación también es un error. Los nuevos miembros del equipo, que ingresan con muchas ganas de aportar, tendrán que luchar con el desorden y el caos de los proyectos. Y, ante la necesidad de mostrarse resolutivos, y al no haber patrones, arquitectura, ni nada definido, ellos trataran de resolver sus problemas de la manera que mejor les parezca y, por ende, introducirán soluciones rápidas que aumentaran aun más la deuda técnica existente. El tiempo seguirá avanzando, y con él la complejidad de los proyectos y su deuda técnica. La productividad bajará tanto que, aunque incorporen nuevos miembros al equipo, es poco lo que se podrá seguir avanzando, ya que, ante el más mínimo cambio u agregado en el código, se deberá invertir una gran cantidad de esfuerzo para solucionar los nuevos errores introducidos. En este tipo de casos, como lamentablemente nunca se "pagó" la deuda técnica, los proyectos entran en bancarrota técnica, y es entonces cuando los sectores de IT deciden migrar los proyectos a nuevas tecnologías, culpando a las anteriores tecnologías como causante de los fracasos.

Todo lo anterior sucede muy a menudo porque los líderes suelen subestimar el valor del trabajo técnico y su calidad, también porque ignoran o no comprenden como las buenas prácticas mejoran la calidad del código y, ante esta carencia, naturalizan la dificultad exponencial que va surgiendo en el día a día, durante el proceso de desarrollo de software.

Lamentablemente muchos líderes justifican sus malas decisiones argumentando que para poder escribir código de alto nivel se necesitaría invertir una mayor cantidad de tiempo y esfuerzo para realizar nuevas funcionalidades, y que esto provoca pérdidas de productividad en el desarrollo. Sin embargo, a pesar de que tienen razón en el corto plazo, ya que las nuevas funcionalidades se terminan entregando más tarde, ellos ignoran completamente que a mediano y largo plazo, un código inmanejable se convierte exponencialmente más difícil de mantener. Por lo tanto, un código de alta calidad resulta ser más fácil de mantener y es más productivo en la mayoría de los proyectos de hoy en día, ya que estos aumentan de tamaño y complejidad con el tiempo.

¿Por qué a veces lo mejor es priorizar la calidad por sobre las metodologías?

Habitualmente, cuando los proyectos inician y son mantenibles, no se le da importancia a las metodologías ágiles ni a los procesos formales de desarrollo. Estos suelen cobrar importancia tardíamente, cuando el código ya se ha tornado inmantenible por su aumento de tamaño y complejidad, y genera baja productividad.

Opino que es un error, en este tipo de situaciones, que se priorice las metodologías por sobre la calidad del código. Al comienzo del proyecto, la metodología, la organización y los procesos de desarrollo ayudan pero, con el tiempo la deuda técnica avanza, y aunque se hagan enormes esfuerzos para prevenir errores con el sector de QA, y que los problemas de estimaciones se intenten solucionar afinando más la organización, planificación, y los procesos formales de desarrollo, con cada sprint habrá más y más retrasos, y más bugs. Todos los esfuerzos que se hagan por fuera del código para compensar las deficiencias de éste no sirven, porque es como la basura que se esconde debajo de la alfombra, la basura que es el código seguirá acumulándose y la alfombra en un punto ya no alcanzará para tapar tanta basura. Lo mejor es arreglar todo ese desastre y no intentar taparlo.

Los problemas del código se solucionan trabajando en el mismo código, mejorando la calidad del mismo por etapas, de a poco con sucesivos refactors, aplicando buenas prácticas hasta llegar a obtener un código limpio y mantenible.

¿Por qué defiendo Clean Code y no otras prácticas de desarrollo?

Clean Code es una filosofía de desarrollo de software, con un listado de reglas y un conjunto de buenas prácticas de desarrollo, muchas de ellas tomadas de otros lados, que facilitan la escritura y lectura de un código, haciéndolo más fácil de entender y, que por lo tanto, aseguran su mantenibilidad. A diferencia de otras prácticas más difíciles de implementar, como TDD que requiere tener mucho de conocimiento de testing, Clean Code puede ser aplicada por todos los desarrolladores, con tan solo aplicar algunas reglas simples ya se puede asegurar la viabilidad de un proyecto a largo plazo. Un código limpio y elegante es fácil y barato de evolucionar, escalar y mantener.

¿Por qué prefiero implementar Clean Code y no una buena documentación?

Una documentación muchas veces miente o no dice toda la verdad. Es texto que, frecuentemente, queda desfasado de la realidad, o no refleja la funcionalidad del código que ha sufrido reiterados cambios. También puede suceder que los desarrolladores no mantengan actualizada la documentación. A fin de cuentas, no hay mejor fuente de la verdad que el código mismo, porque el código se ejecuta y funciona o no. Por esta razón es importante que el código sea legible, predecible y mantenible. Cualquier regla de negocio compleja que se ha olvidado, fácilmente se la puede recordar leyendo un código bien escrito.

¿Por qué un lenguaje tipado y no uno dinámico?

Es típico ver a un desarrollador JavaScript debugueando, teniendo que ejecutar su aplicación e insertando “console.log” en el código para ver en la consola si se están enviando correctamente los datos, o si estos llegan con el formato correcto, etc. Los lenguajes dinámicos cuando son usados correctamente resultan ser muy prácticos, pero cuando se abusa de las virtudes y flexibilidad que poseen, pueden generar código poco predecible, ilegible y muy propenso a errores.

Lamentablemente, los programadores que solo han usado lenguajes dinámicos, no tienen bien claro como y cuando es correcto usar las virtudes de estos lenguajes, y cuando esto sucede, crean código dinámico que es muy difícil de mantener. Los lenguajes tipados vienen a solucionar este problema, ya que son lenguajes bien documentados y predecibles, y son muy estrictos en como manejan los datos.

Los defensores de los lenguajes dinámicos dirán que los lenguajes tipados agregan una capa de abstracción innecesaria para el manejo de los datos, y que merma la productividad. Esto es medianamente cierto porque al inicio hay que invertir tiempo en definir en como va a ser los datos, pero a mediano y largo plazo, estos lenguajes ayudarán a prevenir la mayoría de los errores, sin la necesidad de ejecutar el código en vivo, lo que es muy favorable también a la hora de hacer refactors.

Valores por defecto según su tipo y no resultados dinámicos.

Muchas veces he tenido que consumir APIs hechas por otros desarrolladores, las cuales lamentablemente respondían con datos difíciles de tratar, ya sea porque los objetos eran dinámicos o porque las propiedades de los objetos variaban en su tipo de dato, o porque estos objetos variaban en la cantidad de propiedades que contenían, etc. Esto me obligó a tener que desarrollar permanentemente a la defensiva, es decir, debía preguntar en el código, usando condicionales, si los datos existían y si tenían el formato correcto para prevenir errores de ejecución, y si no tenían el formato correcto, debía reconvertirlo u obtenerlo de otro lado de la respuesta, cuestión que terminaba ensuciando mi código con el uso de muchos condicionales que prevenían errores y que, de alguna forma u otra, intentaban obtener los datos para poder manejarlos basándose en decisiones, es decir, más condicionales.

Otra cuestión que me sumaba complejidad técnica, sucedía cuando los datos no tenían un nombre bien descriptivo de que significaba, ni mucho menos podía intuir de que se trataba, O peor aun, cuando los datos no declaraban conclusiones o deducciones, y yo mismo con condicionales debía averiguar de que tipo de datos se trataban.

Considero que colocar valores por defecto es respetar el tipo de dato de una variable o propiedad, para que siempre mantenga el mismo tipo de dato, incluso cuando se ha intentado asignarle un valor vacío este debería seguir manteniendo el mismo formato. Cuando hablo de valores vacíos por defecto según su tipo, me refiero a que, por ejemplo, una propiedad que es un listado de objetos, jamás deberá tener otro valor que no sea un listado, y si el listado es vacío, no debe ser null, en su lugar debe ser un listado sin valores.

Lo mismo sucede con las propiedades que son números, que en todo momento no deben dejar de serlo, y en caso de querer representar un valor vacío por defecto será 0. Y en cuanto a las propiedades que son texto, estas deben tener un texto vacío y no un null, ni mucho menos, otro tipo de dato.

Por culpa de la mala práctica de otros, la complejidad técnica que esto sumaba a mi código era tal que mi productividad se reducía considerablemente. Por suerte, aprendí, durante una meetup, una estrategia de como solucionarlo, normalizando los datos en la capa donde estos ingresan en la aplicación. Ya no tengo que intentar arreglar los datos, ni impedir errores en otras partes de la aplicación, he aprendido a solucionar la mayoría de los problemas relacionados con el mal manejo de los datos.

¿Por qué programación declarativa y no imperativa?

En varias ocasiones me he encontrado manteniendo código muy complejo, lleno de iteradores dentro de otros iteradores, y con muchos condicionales dentro. Resulta muy difícil poder entender el objetivo, o lo que se intenta resolver, dada la enorme cantidad de información visual expresada en el código sobre diferentes procesos, todos mezclados en una única solución.

El código que es declarativo, en cambio, ayuda a entender inmediatamente que es cada parte del conjunto de la solución, ya que cada avance en la solución es bien descriptiva y su código es escalado. Usar un código declarativo me hace ser más productivo al momento de agregar nuevas características, y conllevan a mayor productividad en la creacion de nuevas caracteristicas y en su mantenimiento a mediano y largo plazo.

¿Por qué programación funcional y no orientada a objetos?

Cuando comencé a incursionar sobre temas relacionados sobre la programación declarativa, y sobre otras prácticas para crear código mantenible, dí con lo que es la programación funcional y, gracias a ella, considero que he mejorado enormemente la prolijidad con la que escribo código.

Gracias a la programación funcional he podido reducir líneas de código, al mismo tiempo que he podido desglosar soluciones complejas en muchas partes, las cuales fácilmente pueden ser reutilizadas por otras partes de la aplicación. Incluso puedo armar funciones nuevas utilizando otras ya existentes. Su inmutabilidad y el uso de funciones puras, me ayudan a crear código que es más predecible, sin tener que tratar con efectos colaterales indeseados. Y gracias a su recursividad y funciones de primera clase, puedo hacer partes de código muy flexibles y reutilizables.

La programación orientada a objetos la utilizo principalmente para modelar los datos, solo eso, el resto lo trabajo con la programación funcional, ya que me permite programar más fácilmente soluciones que son muy complejas.

¿Por qué Clean Arquitecture y no cualquier improvisación u solución rápida?

Es muy común ver en proyectos, en los que hay que realizar mantenimiento, que los diferentes tipos de funcionalidades del código no estén agrupados, ni clasificados, ni ordenados, e incluso que estén mezclados con otros tipos de funcionalidades. Esto hace que sea difícil saber donde colocar las nuevas funcionalidades y, además, que sea poco intuitivo donde encontrar las funcionalidades que ya fueron hechas.

Otra situación frecuente, es ver que los proyectos no tienen un único patrón para realizar una determinada funcionalidad, o que las diferentes partes de la aplicación se comunican entre sí sin una jerarquía clara, ni tengan un flujo intuitivo en como se comunican estas partes. Todo este tipo de situaciones hacen que merme la productividad de todo un sector de desarrollo.

Clean Arquitecture viene a solucionar todos estos problemas relacionados con la organización del código. Hereda todas las prácticas de Clean Code, que están centradas en la calidad del código, pero las lleva a una escala superior para los proyectos, ordenando, clasificando y definiendo flujos de comunicación de cada una de sus partes, para lograr una correcta homogeneización. Establece reglas claras que resultan ser muy intuitivas para los desarrolladores cuando estos tienen que agregar y modificar características al proyecto.

Es cierto que con un simple y bien escalado MVC, MVP, MVVM es más que suficiente, pero existen otras arquitecturas que, si bien son más complejas de entender y aprender, son idóneas para proyectos que son realmente enormes y que deben durar mucho tiempo.

¿ Por qué GitHub Flow y no GitLab Flow ?

Usar GitLab Flow tiene sus ventajas y desventajas. Permite atajar todos los errores de código con la utilización de múltiples ramas de desarrollo. Sin embargo, también es muy probable que surjan diferencias y conflictos de código entre las ramas. Esto implica que los desarrolladores continuamente deban invertir esfuerzo para resolver los conflictos, corriendo el riesgo de corregirlos incorrectamente. Esta dificultad se acentúa todavía más cuando la metodología ágil utilizada es SCRUM, porque las entregas se hacen después de mucho tiempo de desarrollo y las diferencias de código y conflictos son más grandes.

GitHub Flow, en cambio, nos propone un flujo de trabajo mucho más simplificado, con una única rama de desarrollo como la real, y cada cambio u modificación que se le haga a esta, debe ser tratado como si fuera el producto final. En vez de tener que invertir tanto esfuerzo en resolver conflictos de código, se lo invierte en mejorar la calidad del código nuevo.

GitHub Flow incentiva a los desarrolladores a hacer entregas continuas, pequeñas y de alta calidad, lo que es idóneo cuando se desea incorporar “integración continua” en el ambiente de producción, sin tener que esperar largos periodos de tiempos para que el cliente pueda ver los nuevos avances en su producto.

Siguiendo la misma línea de Kaizen, GitHub Flow plantea ir conquistando pequeños objetivos para lograr una meta final mayor, por lo tanto, se integra naturalmente a Kanban y Scrumban, metodologías que también son extremadamente ágiles y se basan en pequeñas tareas entregables.

¿Por qué Kanban y no SCRUM?

Scrum es sin duda una metodología excelente cuando se la aplica correctamente, pero es realmente difícil implementarla para la mayoría de las empresas, en especial para todas aquellas que son pequeñas y medianas.

La dificultad de implementar SCRUM consiste en que no está preparada para afrontar cambios abruptos en el transcurso de un Sprint. El análisis, la planificación, la organización y la estimación de las tareas se realiza antes de que comience el Sprint. Sumar una tarea que no fue contemplada con anticipación acarrea caos. Cuando los desarrolladores se salen de las estimaciones por causas imprevistas, tienden a generar soluciones rápidas y de mala calidad, por ende acumulan deuda técnica que en el futuro afectará a la productividad.

SCRUM solo debe ser utilizado en equipos de trabajo grandes, porque se necesitan muchos roles adicionales y específicos para poder llevar a cabo la metodología ágil.

Kanban comparte similitudes con SCRUM, pero su enfoque es opuesto, ya que no cuenta con Sprints rígidos y muy planificados. Kanban recibe las tareas y su planificación forma parte del estado de la misma tarea, es decir, quienes crean, planifican, priorizan y diseñan las tareas, también participan en el tablero con los estados de estas. Por lo tanto, de esta manera es fácil ver como las tareas fluyen entre los diferentes estados, y se puede detectar en que estados hay cuellos de botella u otros problemas.

Gracias a que Kanban tiene estados para describir en que fase se encuentran las tareas, y que no requiere de reuniones largas y extensas (como las planning de SCRUM), las reuniones de planificación se hacen por tarea de forma separada de las demás, es decir, son reuniones mucho más cortas que no ocupan jornadas enteras de trabajo.

Kanban está alineado con la práctica Kaizen que consiste en dividir tareas complejas en objetivos pequeños e incrementales, donde cada tarea que representa un objetivo no puede durar más de dos o tres horas de trabajo, y si así lo fuese, se deberá dividir el objetivo aun más en tareas más pequeñas y concretas. Gracias a esto se puede estimar cuanto tiempo de desarrollo puede tomar una historia de usuario, contando la cantidad de objetivos y multiplicándolos por dos o tres horas.

En lo que se refiere a los cambios inesperados, y de último momento, que inevitablemente surgen, con Kanban se pueden incorporar perfectamente. Cada cambio será considerado como una nueva tarea. Luego se deberá establecer que tareas tienen que resolverse con mayor urgencia que otras y se les dará prioridad.

Kanban también es apropiado para equipos de trabajo grandes porque escala muy bien en ellos. Permite visualizar todas las tareas de todos los sectores, o de uno en concreto, filtrándolas por un sistema de etiquetado, y cada sector o grupo de trabajo puede tener sus propias etiquetas para identificar sus tareas con mayor facilidad.

Como si esto fuera poco, dado que las reglas de Kanban son muy concretas y simples, se le puede incorporar prácticas de otras metodologías. Es muy común que se incorporen ciertas prácticas de SCRUM por las necesidades de las empresas y de los clientes, transformando Kanban en Scrumban.

Según sea el caso, a veces es mejor armar pequeños grupos de trabajo multidisciplinarios, otras veces es mejor separar al equipo de IT en áreas grandes y especificas. Lo que si es seguro y tengo bien en claro es que Kanban es una metodología verdaderamente ágil, muy potente y altamente productiva.

¿Por qué acordar tareas en base a descripciones y no en base a palabras?

A la hora de asignar y pactar las tareas que se van a resolver en el día, estas deben contar con una descripción bien detallada, y no pactarlas de palabra porque es muy frecuente que las personas se olviden lo que acordaron en reuniones previas. Además, en el caso de que se hayan tomado malas decisiones con respecto a los objetivos de las tareas, se puede recurrir a las dichas descripciones para respaldar el trabajo hecho y las decisiones que se tomaron de conjunto.

Considerar todos los puntos de vista y no asumir veracidades propias

La información y los datos son clave para la planificación de las tareas. Mientras más datos e información se posea, más certera va a hacer la planificación y el planteamiento de los objetivos, por esto, es importante hacer reuniones cortas con diferentes miembros del equipo, e incluso de diferentes áreas, los usuarios y los clientes, para poder conocer sus puntos de vista y lograr encontrar una convergencia entre todas las ideas. De esta manera, es posible detectar cuestiones que no fueron consideraras y que podrían acarrear grandes problemas en el futuro.