**Introducción al Curso**

Según el Project Management Institute, más del 70 por ciento de las organizaciones han

incorporado algunos enfoques ágiles y el 77% de las empresas adopta el scrum.

Además, más del 25 por ciento de las empresas de fabricación utilizan Agile exclusivamente.

Y según un estudio de Price Waterhouse Coopers, los proyectos ágiles tienen un 28% más de éxito

que los proyectos tradicionales.

Lamentablemente, según Forbes, el 47% de todas las transformaciones ágiles fallan, la razón principal es la falta de experiencia en la implementación e integración de la metodología ágil.

La metodología ágil no es solo un proceso que se sigue, es una forma de pensar, una filosofía que requiere pensar de forma diferente como gestionar los equipos y proyectos. Los

líderes ejecutivos tienen que cambiar el enfoque tradicional a largo plazo y pasar a centrarse en ofrecer valor de forma rápida e iterativa para satisfacer a los clientes.

Para lograr la agilidad, es necesario cambiar la cultura de la empresa.

Otra razón por la que los proyectos ágiles fracasan es que la metodología ágil no es algo que se pueda aprender simplemente

de un libro sino que se necesita la experiencia y acompañamiento de un instructor, similar a cuando alguien aprender a conducir.

**Principios Ágiles**

La metodología ágil es un enfoque iterativo de la gestión de proyectos que permite a los equipos

responder y ofrecer valor a sus clientes con rapidez. A diferencia de otros enfoques de planificación en los que planificamos el trabajo de todo un año, con Agile planificamos pequeños incrementos y, a continuación, recibimos comentarios del cliente para ver si le gusta lo que estamos haciendo y luego nos ajustamos a medida que avanzamos.

¿Cuáles son algunas de las características definitorias?

Agile hace hincapié en esta planificación adaptativa: No planificamos todo un año, solo planificamos una pequeña iteración para ver si podemos ofrecer algo de valor al cliente, recibir comentarios y ver qué es lo que le gusta. Eso nos permite hacer un desarrollo evolutivo.

El desarrollo evoluciona con el tiempo: En lugar de construirlo todo de una vez, lo estamos construyendo en pequeños incrementos. Y luego evolucionamos a medida que avanzamos y respondemos a medida que avanzamos.

Entrega temprana: La entrega temprana es un componente clave de la agilidad. Poner algo en manos de los clientes, recibir comentarios de ellos para que lo entiendan, ¿les gusta? ¿Debemos dar un giro, debemos perseverar? Esta es realmente una característica fundamental que define la agilidad.

Mejora continua: Mejorar continuamente como equipo y Mejorar continuamente el producto que entregamos porque tenemos la oportunidad de recibir comentarios de nuestros clientes, porque lo entregamos. Y eso nos permite responder a los cambios. Por lo tanto, a medida que se producen cambios podemos responder a esos cambios.

Manifiesto Ágil.

- Valorar a las personas y las interacciones por encima de los procesos y las herramientas.

- Valorar al software funcional por encima de la documentación exhaustiva.

- Valorar la colaboración con los clientes por encima de la negociación de contratos.

- Valorar la respuesta a los cambios por sobre el hecho de seguir un plan predefinido.

Si bien los artículos de la derecha tienen valor, nosotros valoramos más los de la izquierda. No significa que no tengamos herramientas de procesamiento, solo significa que valoramos la interacción entre nosotros y con nuestros clientes más que el

proceso y las herramientas. No significa que no documentemos. La documentación es importante para que las personas entiendan cómo usar tu producto. No significa que no negociemos contratos,

todavía tenemos contratos, pero queremos colaborar con el cliente. A medida que las cosas

cambian, queremos ser receptivos, lo que no significa que no tengamos planes, sino que valoramos responder al cambio en lugar de simplemente seguir el plan. Por lo tanto, el desarrollo ágil de software es un enfoque iterativo del desarrollo de software que se ajusta al Manifiesto Ágil.

También queremos asegurarnos de hacer hincapié en la flexibilidad, la interacción con nuestros colegas, la interacción con nuestros clientes y en tener un alto nivel de transparencia, para que todos sepan en qué están trabajando los demás.

Todos se sienten responsables de ofrecer valor a los clientes. Esto es de vital importancia.

Y, por último, utilizamos equipos pequeños, ubicados en el mismo lugar, interfuncionales, autoorganizados y autogestionados.

La conclusión clave aquí es construir lo que se necesita, no lo que se planeó, a medida que las necesidades cambien, siempre podemos replanificar y responder a los cambios.

# Waterfall.

Todo comienza con una fase de requisitos. Las personas recopilan los requisitos, ven lo que el cliente quiere y se aseguran de que vamos a entregarle algo que el cliente desea. Al menos en ese momento. Por lo tanto, esta es una fase en la que lo único que tiene que hacer es documentar todos los requisitos que el cliente podría desear incluir en el sistema.

Luego pasa a la fase de diseño. Observe que estoy usando el término fases aquí. Esto es muy importante, porque había criterios de salida y criterios de entrada para pasar

de una fase a otra. Así que una vez que tenemos todos los requisitos, pasamos al diseño, con la esperanza de tener todos los requisitos que realmente necesitamos. Entonces, en la fase de diseño, los arquitectos están diseñando. Están descubriendo cómo tomamos esos requisitos, los convertimos en software que funcione y, por lo tanto, diseñan todo el sistema.

Y luego, una vez finalizada esa fase, pasamos a la fase de codificación. Así que aquí es donde los desarrolladores están hackeando la codificación.

Ahora, notarás que esto se llama Cascada y esas flechas fluyen hacia abajo. porque es muy difícil subir en una cascada, o sea volver para atrás. Así que este es un nombre apropiado, porque cuando estás en la fase de programación y descubrís que hay un problema con el diseño, no funciona, es muy difícil volver a empezar y rediseñar las cosas.

Después de la codificación, es la fase de integración.

Así que desde el principio, codificamos de forma aislada, no integramos mi módulo con el

módulo de la persona de al lado. Sabes, hay un momento en que todos esos módulos se unen. La primera vez que nos dimos cuenta, ¿todos estos fragmentos de código funcionan juntos?

Y luego pasamos a la fase de prueba, porque ahora tenemos un sistema que la gente

puede probar. Y cuando encuentran errores, tienen que volver, nadar hasta la cascada, abrir algunos errores en la fase de codificación y hacer algunos recodificadores.

Y luego, si uno de esos errores que probaron resulta ser que realmente necesitas cambiar

el diseño porque estas cosas no interactúan bien, muy arriba en cascada, es muy caro

volver a la fase de diseño.

Y finalmente, una vez realizadas todas las pruebas, implementamos el software.

Entonces, ¿qué tiene de malo este enfoque?

Bueno, no hay ninguna disposición para el cambio.

-Cada fase tiene criterios de entrada y salida. Cuando una termina comienza la siguiente. Y simplemente no hay ninguna disposición para volver atrás y cambiar el diseño o los requisitos

ni nada parecido. Así que el otro problema es que no sabes si funciona hasta el final, claro, no hay

entrega intermedia. No se entrega nada, hasta el último paso, cuando se lo damos al

equipo de operaciones y le decimos: vaya a poner esto en producción.

-La otra es que cada escalón termina cuando comienza el siguiente, de ahí viene el

nombre de la cascada, donde las cosas se transmiten de padres a hijos.

Y, por supuesto, cada una de estas es una oportunidad de perder información, ya sabes, y de que

ocurra un percance o de que la gente quede bloqueada, porque no pueden aceptar, ya sabes, el trabajo de la fase anterior.

Y ahora estás esperando para comenzar la siguiente fase.

-La otra es que los errores que se encuentran más adelante son muy, muy costosos para encontrar

algo que está mal diseñado y probarlo y volver a diseñarlo.

-Y, por último, están los largos plazos de entrega, justo entre la entrega de ese software. Desde el momento en que quieres el software por primera vez, diseñas el software, lo codificas y lo pruebas, para cuando finalmente lo entregues, largos, largos plazos de entrega.

Así que el problema aquí es que los equipos trabajan por separado, no son conscientes del impacto que tienen entre sí. Cierto, los diseñadores desconocen el impacto del código, los programadores desconocen su impacto en la integración de todo el código. Allí todos trabajan en sus pequeños silos para su pequeña fase. Y entonces esto debería ser una idea aterradora. Las personas que están más alejadas de un código, el pobre equipo de operaciones, tienen que ejecutarlo y administrarlo en producción, ¿verdad? Son los que menos saben sobre el código y son los que se espera que lo ejecuten.

# La programación extrema.

Es muy iterativo y habla de bucles.

Tienes estos importantes planes de lanzamiento en el circuito exterior. Y luego tienes un plan de iteración.

Y así, el lanzamiento tal vez meses, las iteraciones quizás semanas, la

aceptación tiene tal vez días, reuniones de pie una vez al día, sí, negociación de pares en horas,

pruebas unitarias en minutos, programación en pares en segundos.

Ya sabes, son esos ciclos cada vez más estrechos de hacer un trabajo y luego recibir comentarios rápidos.

Se basa en un enfoque iterativo del desarrollo de software.

Y, de hecho, aquí es de donde lo sacó Agile.

Y la intención aquí es mejorar la calidad del software responder a los cambios,

responda a los requisitos de los clientes y haga las cosas en pequeños incrementos.

Y mucha gente reconoce a la programación extrema como uno de los primeros métodos ágiles.

Así que hablemos de los valores de la programación extrema.

Simplicidad, cada uno hace lo que tiene que hacer y nada más, no sobrediseñamos, no sobrecodificamos, no entregamos más código del que el cliente solicitó, lo mantenemos simple.

Comunicación.

Todos los miembros del equipo deberían comunicarse y saber lo que hacen los demás, ya que esto fomenta mucha, mucha comunicación e interacción entre las personas.

Comentarios, no tienes ni idea de cómo te va a menos que recibas comentarios de quienes están interesados, por lo tanto, tener esos circuitos de retroalimentación es fundamental para la programación extrema y fundamental para el enfoque Agile en general.

Y respeto: todos sienten que son respetados en el equipo, que pueden ofrecer consejos, que pueden hacer sugerencias y que sus sugerencias son tan valiosas como las de cualquier otra persona del equipo. No hay jerarquía, todos tienen compañeros en el equipo y son respetados por sus ideas.

Y finalmente, coraje, claro, no aumentamos nuestras estimaciones, simplemente somos muy honestos acerca de que creemos que podemos hacer esto y no creemos que podamos hacer eso.

Y no te vamos a mentir diciendo que, ya sabes, podemos hacer esto, o no podemos hacer esto en este tiempo. Somos muy abiertos y honestos en Programación extrema, estas son las estimaciones, esto es con lo que nos comprometeremos.

# Kanban.

kanban significa literalmente letrero en vallas publicitarias, y se trata de un flujo continuo en la

planta de fabricación, donde estas tarjetas o billetes fluían con el producto de una

estación a otra, en el futuro.

Por lo tanto, el principio fundamental de Kanban es visualizar el flujo de trabajo. Si no puede ver la obra, no puede administrarla. Y esto es algo que también incorporamos a Agile. Descubrirás que vamos a utilizar tableros Kanban y Agile. Y se trata de visualizar el trabajo, es muy, muy importante que puedas entender el trabajo que hay que hacer y que todo esté contabilizado.

Kanban también hace hincapié en limitar el trabajo en curso. Cuando estás en una planta de fabricación, no querrás que el trabajo se acumule en una de las estaciones. Pero incluso cuando estás desarrollando software, no quieres que la gente trabaje en demasiadas cosas a la vez.

Porque, ya sabes, solo puedes enviar el 100% de una cosa que funciona, no puedes enviar el

50% de dos cosas. Así que si tienes a una persona trabajando en dos cosas, no es bueno. Y eso viene del Kanban para limitar el trabajo en curso.

Luego está la gestión y la mejora del flujo. En Kanban, siempre buscan mejorar.

Siempre buscan cambiar y entender cómo podemos lograr un flujo mejor y más rápido

en la planta de fabricación y, luego, hacer políticas explícitas para que todos entiendan

cómo funcionan las cosas.  Y todo el mundo entiende la definición de «hecho», lo que significa que algo esté «hecho». También lo tomamos de Kanban en nuestra planificación ágil para entender cuál

es la definición de «hecho».

Y luego, mejorar continuamente, es muy, muy importante para recibir comentarios y mejorar continuamente lo que estás haciendo.

Por lo tanto, Kanban consiste en hacer que los flujos funcionen mejor, entenderlos, verlos,

poder medirlos y, luego, mejorar continuamente la forma en que podemos fluir más rápido.

**Siendo Ágiles**

Cinco prácticas del trabajo Agile.

Explicar el trabajo en lotes pequeños, definir un producto mínimo viable, comprender el desarrollo impulsado por el comportamiento, entender el desarrollo impulsado por pruebas y explicar cómo funciona la programación en pares.

Trabajar en lotes pequeños

Esto lo obtenemos de la fabricación ajustada. No querrás hacer un lote grande de algo y luego descubrir que lo has hecho mal, porque es un desperdicio tener que volver y cambiarlo. A continuación, debes utilizar un mínimo de productos viables para averiguar qué es lo que le gusta al cliente y qué no le gusta. Además, querrá utilizar prácticas como el desarrollo basado en el comportamiento (BDD) y el desarrollo basado en pruebas (TDD) para mejorar las pruebas del sistema desde fuera hacia dentro y desde dentro hacia fuera. Y, por último, quieres aprovechar la programación en pares. Esto proviene de una programación extrema, pero la programación en pares es de vital importancia. Así que vamos a ver cada una de ellas por separado.

Trabajando en lotes pequeños

Digamos que necesito enviar mil folletos por correo. Y los pasos son doblar los folletos y, a continuación, insertarlos en los sobres, sellar los sobres y, finalmente, sellar los sobres con franqueo.

Supongamos que vamos a hacer estos mil en lotes de 50, sí, lotes grandes, 50 por pieza y supongamos que cada paso tarda unos seis segundos en completarse. Así que puedo hacer unos 10 pasos por minuto.

Así que hacemos el primer paso de plegado, ¿verdad? Y con 10 por minuto, tardo unos cinco minutos en

doblar 50 de ellas. Luego paso al siguiente paso. El siguiente paso es insertarlos todos y otra vez, si pudiera hacer unos 10 por minuto, y así tardaré cinco minutos más, más los cinco minutos anteriores. Ahora llevo 10 minutos. Los tengo todos doblados e insertados y luego paso al paso tres para sellarlos todos, ¿de acuerdo? Así que ahora, de nuevo, unos 10 por minuto, podría hacer otros 50 en otros cinco minutos. Así que tengo hasta 15 minutos, ¿verdad? Hora de reloj de pared. Y finalmente llegué al cuarto paso y tardé 16 minutos en obtener mi primer producto terminado, ¿verdad? El primero, en el que tenía cosas que podía inspeccionar y comprobar su calidad.

Entonces, ¿qué pasaría si los sobres no tuvieran pegamento, verdad? Pasarán 11 minutos antes de que me dé cuenta de que algo ha ido mal. ¿Qué pasa si hay un error tipográfico en el folleto? Tengo que volver

hasta el principio, ¿verdad? Por lo tanto, es un desperdicio trabajar en lotes tan grandes como este.

Ahora veamos el mismo ejemplo con un flujo de una sola pieza. Y de nuevo, supongamos que cada paso tarda unos

seis segundos en completarse. Con un flujo de una sola pieza, lo pliego, ¿verdad? Y ponlas en el sobre y

séllalas y ponles un sello y el primer producto acabado estará en unos 24 segundos. Ahora

puedo inspeccionarlo para ver si funciona, ver si es lo que pensaba que era y luego dejar que el resto del flujo se

rompa. ¿Y si los sobres no tuvieran pegamento? Bueno, lo averiguaría después de, quizás, 18 segundos, ¿verdad?

Y si hay un error tipográfico en el folleto, después de 24 segundos, es muy, muy importante que no

trabajemos en lotes grandes para poder recibir comentarios rápidos y poder reflexionar y entender,

ya sabes, ¿necesitamos hacer algo diferente y luego hacer los cambios que necesitamos hacer?

Producto mínimo viable.

Primero, entendamos lo que no es, porque existe la idea errónea de que un producto mínimo viable es como la primera fase del proyecto o, ya sabes, la primera versión beta o algo así. Y no lo es. Esto no es lo que es un producto mínimo viable.

Un producto mínimo viable es lo mínimo que se puede hacer para probar una hipótesis y aprender y comprender. La diferencia entre estos dos es que el primero tiene que ver con la entrega, ¿verdad? ¿Qué voy a entregar? Pero la segunda tiene que ver con el aprendizaje. ¿Qué puedo aprender? ¿Qué puedo aprender al publicar este MVP y recibir comentarios y, quizás, mejorar aún más el siguiente? Por eso es importante que, al final de cada MVP, decidas si quieres cambiar de camino o perseverar. Veamos un ejemplo. Este es un equipo que está desarrollando un producto mínimo viable para un cliente que quiere un automóvil. Y así, en la primera iteración, entregan una rueda. El cliente dice: «¿Qué voy a hacer con una rueda? ¿No puedo hacer nada con esto? Bueno, estamos trabajando en iteraciones, estamos intentando ser ágiles, ¿verdad? Así que, en la próxima iteración, os daremos algunas más. Y les dan un chasis y dicen: «De acuerdo, realmente no puedo hacer nada con esto». Y finalmente, ya sabes, les dan un auto sin volante y,

finalmente, se quedan con el auto, ¿verdad? Y entonces se quedan con este cupé. Ese equipo no sabía

cómo crear un producto mínimo viable. Solo estaban haciendo un desarrollo iterativo.

El segundo equipo entiende el valor de crear un MVP. Al principio, les dan una patineta y el cliente dice: «Te pedí un coche y me das una patineta». Ey, vaya, estamos probando el color. ¿Qué te parece ese color rojo? ¿Es ese el color que quieres? «Oh, sí, el rojo mola bastante, pero ya sabes que es muy difícil de conducir». No, lo arreglaremos en el próximo MVP y, luego, añadirán una forma de dirigirlo y el cliente dirá: «Bueno, vale, me diste una forma de dirigirlo,

pero no puedo ir muy rápido. Necesito una forma de locomoción mejor». Y luego nos ocuparemos de eso en

el próximo MVP. En la siguiente iteración, les dan pedales. En algún momento del camino, mientras el cliente

va en esa moto y siente el viento en el pelo, decide: «Quiero un descapotable». En el primer caso, el cliente obtuvo exactamente lo que había pedido meses antes porque simplemente estaba siguiendo un plan, pero en el segundo caso el cliente obtuvo exactamente lo que deseaba porque estaba trabajando de forma interactiva con el equipo de desarrollo y, al final, se desarrolla algo que es un poco diferente, pero que se acerca más a lo que el cliente realmente quiere.

El desarrollo impulsado por el comportamiento consiste en describir el sistema desde fuera hacia dentro. Por lo general, esto se hace a nivel de pruebas de integración, Por lo tanto, aquí es donde tomamos la

visión del cliente sobre el sistema, por lo general, cuando se realizan pruebas de BDD, las hacemos en función de la interfaz de usuario, Lo haces en contra de lo que ve el cliente para comprobar si el sistema se comporta de la

forma en que debe hacerlo. Entonces, si se trata de un sistema de comercio electrónico, cuando pongo algo en mi carrito, ¿se comporta de la manera que quiero? Cuando transfiero un carrito a un pedido, ¿se comporta de la manera que quiero? En realidad, se trata de adoptar un enfoque de afuera hacia adentro. No le preocupan demasiado las cosas que suceden debajo.

Lo bueno es que utiliza una sintaxis única que tanto los desarrolladores como las partes interesadas pueden

utilizar para describir el sistema y ponerse de acuerdo sobre su comportamiento y sus funciones. Por lo tanto, echemos un vistazo a esa sintaxis. Comienza con estos escenarios de funciones de BDD. Cuando digo:

«En resumen, quiero saber para quién es este papel. Necesito alguna función, ¿verdad? Entonces, ¿cuál

es la funcionalidad que realmente necesitamos para obtener algún valor empresarial?» Así que, ahora sabes

a quién va dirigida. ¿Es para los clientes? Para el administrador del sistema, ¿Quién

saca el máximo provecho de esto? ¿Qué necesita y por qué lo necesita? Y ese es el principio de tu

archivo de características. Y luego empiezas a analizar los escenarios y aquí es donde usamos esta

sintaxis común. Se conoce como sintaxis de Gherkin por el pepinillo de pepinillos. Y es más o menos así:

dadas una serie de condiciones previas, Tengo dos artículos en mi carrito. Tengo tres artículos que pedir, lo que sea. Cuando ocurre algún evento. Entonces debería ver algún resultado observable. Así pues, al usar esta

sintaxis de «cuándo y después», los desarrolladores pueden entenderlo, las partes interesadas pueden

entenderlo, todo el mundo puede entender esta sintaxis común de que cuando tengo algo en mi carrito y luego lo limpio, ya sabes, no veo nada en él. Por eso es muy, muy importante tener la sintaxis en estos escenarios de BDD. El desarrollo basado en pruebas consiste en probar un sistema desde adentro hacia afuera.

A diferencia del desarrollo impulsado por el comportamiento. Se refiere a las partes internas del sistema, los módulos individuales. Por lo general, esto se llama pruebas unitarias, ¿verdad? Y esto es algo que se basa en pruebas unitarias y queremos asegurarnos de que, dadas estas entradas, obtengo estas salidas a nivel de módulo pequeño. No sé si cuando las ponga todas juntas voy a conseguir el comportamiento que quiero.

Es una especie de BDD, pero en el caso del TDD ( desarrollo basado en pruebas), solo estoy probando el sistema a nivel de unidad, así que primero escribes un caso de prueba para el código que te gustaría tener y, luego, escribes suficiente código para que ese caso de prueba pase y, luego, empiezas de nuevo y refactorizas. Así que el flujo de trabajo es más o menos así: escribo un caso de prueba. Podrías decir: «¿Escribes casos de prueba antes de escribir el código?» Sí, esto me mantiene informado de lo que se supone que debe hacer el código. Cuando llamo a esto, ¿cómo quiero llamarlo? ¿Cuáles son los parámetros que creo que quiero transmitir y, entonces, qué me aporta? Entonces, primero escribes el caso de prueba, luego escribes suficiente código para que ese caso de prueba sea aprobado y, una vez que pase, puedes refactorizar el código porque ahora que tienes casos de prueba, sinceramente, sabrás si has roto algo. Ahora bien, cuando el caso de prueba se ejecuta y falla, normalmente se pone rojo y, cuando funciona correctamente, se vuelve verde; solemos llamarlo refactor rojo, verde. Observa un caso de prueba, escribe un caso de prueba , observa cómo falla, haz que escriba suficiente código para que pase, y luego refactoriza, haz que lo hagas, puede que no lo hayas hecho muy elegante solo para hacerlo pasar, ahora puedes volver atrás y añadir todo tipo de comprobaciones de errores y demás, hacerlo un poco más robusto y luego ejecutar los casos de prueba. Asegúrate de no haber roto nada.

Programación en pareja

Programación en pareja, se podría decir que dos programadores trabajan juntos en lo mismo, y lo primero es: ¿pago a dos personas para que hagan un trabajo? Pero en realidad funciona muy, muy bien porque lo que estás haciendo es tener a una

persona escribiendo código y luego tienes a otra persona que tiene otros ojos puestos en el código.

Así que la pregunta es, ¿preferirías encontrar errores en la producción? ¿Prefieres encontrar errores mientras

escribes el código? Es mucho más barato encontrarlos mientras escribes el código, así que al tener dos pares

de ojos, se comprueban y van y vienen. No es solo uno y luego el otro.

Normalmente en incrementos de 20 minutos, ¿verdad? Alguien está escribiendo código, el otro está mirando y

no solo está mirando. A veces buscan cosas, a veces discuten. «¿Cómo crees que

debo llamar a esta variable? ¿Cómo crees que debo llamar a esta función?» ¿Verdad? Son dos grupos de

mentes las que trabajan en torno a este código y lo que pasa es que al final del

día se obtiene un código de mayor calidad, ¿verdad? La calidad del código aumenta porque dos personas ya lo han revisado y, por eso

, cuando dices: «No puedo permitirme pagar a dos personas para que escriban una sola pieza de código». No puedes permitirte no pagar a dos

personas para que lo escriban porque escribir el código es la parte más barata. Depurarlo, mantenerlo en producción,

es la parte más cara y, por lo tanto, si quieres que la parte cara sea más barata, debes

hacer que tus programadores programen por parejas.

También es muy bueno juntar a un programador sénior con un programador junior. Ahora los programadores novatos pueden ver: ¿Cómo aborda el programador sénior el problema?

Y así aprenden, reciben tutoría y, luego, el programa sénior puede ver cómo le

va a la persona júnior. O consigues que personas que no están familiarizadas con el código trabajen con alguien que sí lo está. Ahora hay

más personas que aprenden el código porque lo están haciendo y están recibiendo como un tutorial de

la otra persona. Así que la programación en pareja es una forma muy, muy buena de llevar a todos los miembros de

tu equipo a un cierto nivel. En este vídeo aprenderás que trabajar en lotes pequeños significa

entregar algo útil rápidamente. Un MVP es lo más barato y fácil que puedes hacer para probar la

hipótesis y aprender. El desarrollo impulsado por el comportamiento garantiza que estás creando lo correcto.

Y el desarrollo basado en pruebas garantiza que estás creando las cosas correctamente.

Play video starting at :11:55 and follow transcript11:55

Por otro lado, la programación por pares te permite descubrir los defectos antes y aumentar la calidad del código.