

El Manifiesto Ágil

El Manifiesto Ágil, publicado en febrero de 2001, se considera el documento fundacional de todo el conjunto de metodologías del mismo nombre.

Establece un conjunto de valores y principios comunes a un número de ideas y corrientes que fueron desarrollándose en los años 90, críticas con el modelo rígido y pesado de desarrollo de software existente en la época.

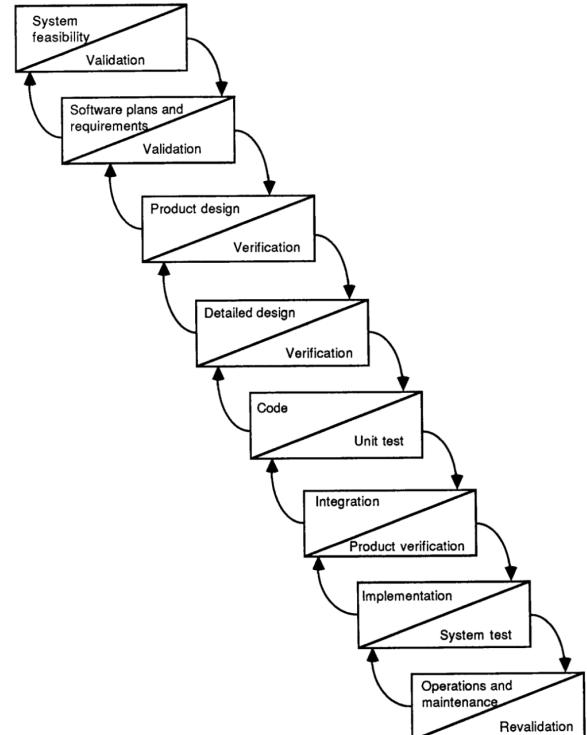
En esta sesión vamos a poner en contexto el manifiesto, explicando esas ideas previas, para pasar después a estudiar en detalle el contenido del mismo.

Tendencias previas al manifiesto ágil

Modelo de cascada

En los años 70 y 80 la forma más generalizada de desarrollar software era utilizando el denominado modelo de cascada (*waterfall model*). Las empresas y consultoras más importantes estaban convencidas de que la forma correcta de crear software era aplicando las metodologías predictivas de las ingenierías tradicionales, en las que se realiza primero toda la especificación y el diseño y se deja una segunda fase la implementación, pruebas y despliegue.

A la derecha podemos ver la ilustración del modelo de cascada tal y como se muestra en el artículo *A Spiral Model of Software Development and Enhancement* de Barry Boehm (1988). Vemos que el modelo se divide el desarrollo de un producto software en 8 fases secuenciales, cada una con su correspondiente verificación. Los errores detectados en la verificación de una fase pueden hacer que se vuelva hacia atrás y se corrijan algunos elementos definidos en la fase anterior.



Las 8 fases definidas son:

- Análisis de la factibilidad del sistema
- Planes de software y requerimientos
- Diseño general del producto
- Diseño detallado
- Códificación (tests unitarios)
- Integración (tests de integración)
- Despliegue (tests del sistema)
- Operaciones y mantenimiento

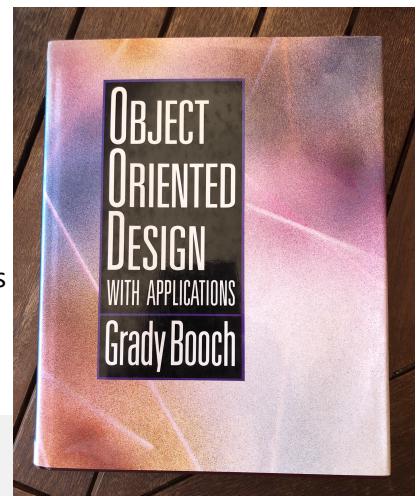
Este enfoque tradicional obligaba a una cantidad muy grande de documentación y a procesos muy burocratizados. Los errores y problemas detectados en las fases inferiores del proceso debían ser reportados usando protocolos muy rígidos y los cambios debían ser aprobados por gerencia y contratados por los clientes antes de ser incorporados en el software.

Programación y diseño orientado a objetos

A finales de los años 80 se popularizó un nuevo paradigma de programación: la Programación Orientada a Objetos. Se hicieron populares nuevos lenguajes de programación orientados a objetos como Smalltalk, Object Pascal o C++ y las grandes empresas informáticas comenzaron a usarlos con la esperanza de intentar mejorar los malos resultados que se estaban obteniendo en los proyectos.

Este nuevo paradigma de programación hizo posible nuevas formas de diseño y arquitecturas de software, como el diseño orientado a objetos de Grady Booch, en las que el software era mucho más flexible y modular y era mucho más fácil introducir cambios en partes del sistema sin alterar el resto.

Por ejemplo, en su libro de 1991 *Object Oriented Design with Applications* Grady Booch afirma que el software creado con el diseño orientado a objetos es resistente al cambio y tiene un nivel de abstracción muy alto, con lo que puede ser entendido mucho mejor:



By applying object-oriented design, we create software that is reslient to change and written with economy of expression.

Y el diseño orientado a objetos también rompe con el modelo tradicional de desarrollo en cascada y va de la mano de procesos iterativos:

With object-oriented design we never encounter a "big-bang" event of system integration. Instead, the development process results in the incremental production of a series of prototypes, which eventually evolve into the final implementation.

La llegada (para quedarse) del paradigma orientado a objetos hizo cristalizar un conjunto de metodologías iterativas ya existentes y puso en cuestión el modelo de cascada tradicional. Además, el nacimiento de la [Web](#) en 1993 y las fuertes inversiones y especulaciones alrededor de las [empresas punto-com](#) a finales de los 90 produjo una necesidad añadida de buscar nuevas formas de desarrollar software, más cercanas a un mundo cada vez más cambiante y rápido.

Desarrollo iterativo

En toda la historia del desarrollo de software han existido propuestas de desarrollo iterativo o evolutivo. Craig Larman y Victor Basili las recogen en su artículo de 2003 [Iterative and Incremental Development: A Brief History](#):

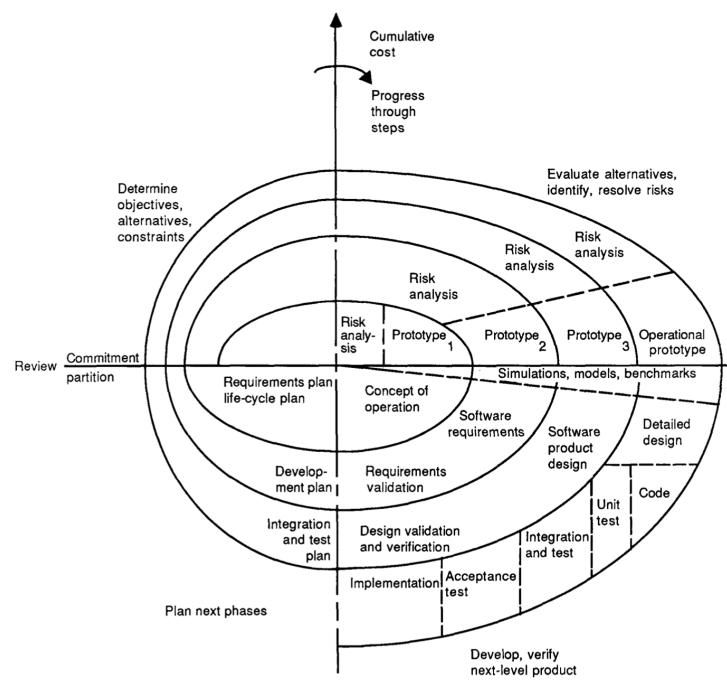
- (1940s) Ciclo PDSA (*Plan-Do-Study-Act*) de Edwards Demming.
- (1950-60s) Proyectos militares y espaciales: jet hipersónico X-15 y proyecto Mercury.
- (1970s) Harlan Mills *Top-down programming in large systems* aplicada en su trabajo en IBM en contratos con el Departamento de Defensa la NSA americana.

- (1976) En su libro *Software Metrics* Tom Gilb es el primero que habla de *evolution* para referirse al desarrollo de software iterativo:

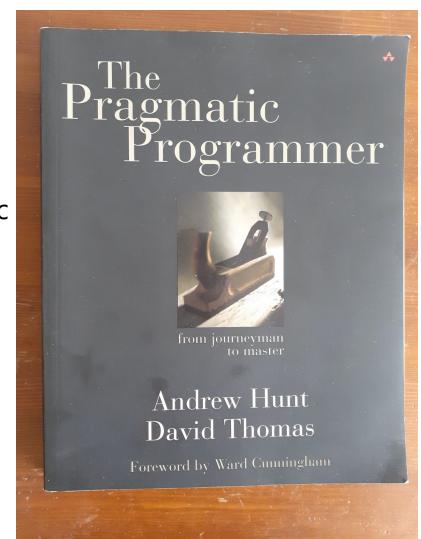
“Evolution” is a technique for producing the appearance of stability. A complex system will be most successful if it is implemented in small steps and if each step has a clear measure of successful achievement as well as a “retreat” possibility to a previous successful step upon failure. You have the opportunity of receiving some feedback from the real world before throwing in all resources intended for a system, and you can correct possible design errors.

Tom Gilb (Software Metrics, 1976)

- (1983) Grady Booch publica *Software Engineering with Ada* en el que propone por primera vez su metodología iterativa de diseño orientado a objetos.
 - (1985) Barry Boehm publica uno de los artículos fundamentales del desarrollo iterativo: *A Spiral Model of Software Development and Enhancement*.



- (1990s) El **movimiento de código abierto** demostró que era posible crear de forma distribuida y auto-organizada complejos sistemas de software con decenas de miles de líneas de código como sistemas operativos ([Linux](#)), editores de texto extensibles con un lenguaje de programación incluido ([Emacs](#)) o bases de datos ([MySQL](#)). En 1999 Eric Raymond publica su libro *The Cathedral and the Bazaar* en donde analiza el modelo de desarrollo del movimiento open source. Y en el mismo año Andrew Hunt y David Thomas publican su influyente libro [The Pragmatic Programmer](#) en el que recogen buenas prácticas de programación relacionadas también con el open source, Linux/UNIX y el desarrollo iterativo.



Metodologías ligeras

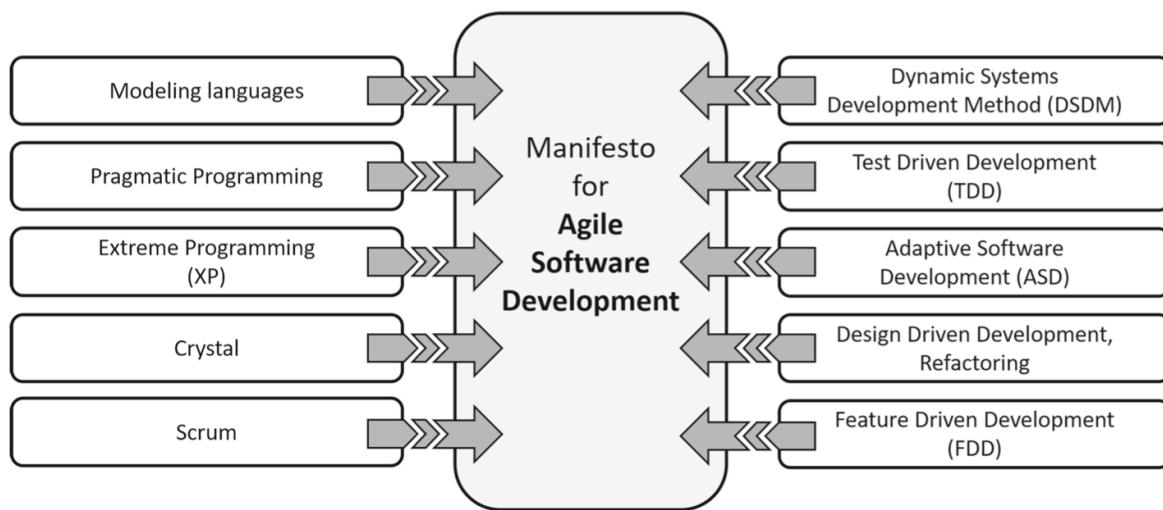
Antes del manifiesto no se utilizaba la palabra "ágil", se hablaba de metodologías "ligeras" (*lightweight*) para contraponerlas a las distintas variantes de la metodología tradicional de cascada existentes, denominadas todas ellas "pesadas".

En los años 90 todos estos enfoques cristalizan en un grupo de propuestas y metodologías ligeras que se hacen populares:

- Rapid Application Development (RAD) y Dynamic Systems Development Method (DSDM) de James Martin se hacen populares en Europa a mediados de los 90.
- Extreme Programming (XP) nace en 1996 promovida por Kent Beck, John Reffries y Ward Cunningham.
- Scrum se propone en 1999 por Jeff Sutherland y Ken Schwaber
- Feature Driven Development (FDD) desarrollada por Jeff De Luca y Peter Coad en 1999.
- A mediados de los 90 Alistair Cockburn propone los Crystal Methods después de una extensa investigación con equipos exitosos en la que concluye que la característica fundamental de estos equipos está basada en las personas y sus interacciones y no en sus procesos.

De todas estas propuestas, la más popular a finales de los 90 era con diferencia XP. Kent Beck la había hecho popular en conferencias, con su libro de 1999 *Extreme Programming Explained* y en cursos organizados junto con Bob Martin.

En el artículo *"Back to the future: origins and directions of the Agile Manifesto – views of the originators"* se presenta esta interesante figura con las influencias que dieron lugar al manifiesto.



Reunión en Snowbird

Martín Fowler, en su artículo [Writing The Agile Manifesto](#) explica el origen de la reunión en la que se desarrolló el Manifiesto Ágil.

En la primavera de 2000 Kent Beck invitó a una reunión a un grupo de líderes en la comunidad XP (Bob Martin, Ward Cunningham, John Refries, , Martin Fowler) junto con otras personas interesadas pero que no formaban parte estrictamente de XP como Alistair Cockburn, Jim Highsmith y Dave Thomas.

En la reunión se discutió la relación entre XP y métodos similares a los que en esa época se denominaban *lightweight methods*. Se analizaron las ventajas de XP como un método muy centrado en la realidad del desarrollo, pero también se acordó que habían muchos elementos comunes entre XP y el resto de métodos.

Como resultado de esto Bob Martin decidió organizar una reunión de gente interesadas en este amplio rango de métodos.

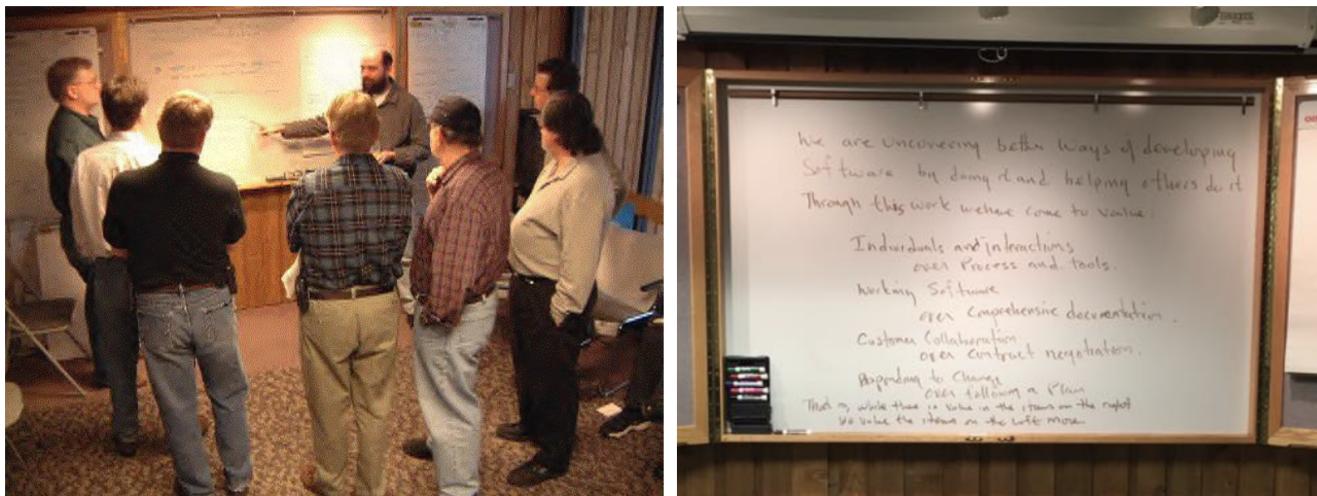
Entre Bob Martin y Martin Fowler se configuró una lista de personas posiblemente interesadas. A esta lista se unió otra de Alistair Cockburn, con lo que se contactaron básicamente a casi todos los principales representantes de las existentes metodologías *ligeras*.

Se definió la fecha y el lugar de la reunión del 11 al 13 de febrero de 2001 en el resort de Snowbird en Utah y se envió invitaciones a todos los seleccionados para participar en la reunión que Bob Martin bautizó con el nombre de "The Light Weight Process Summit" (lo explica en su libro de 2019 [Clean Agile: Back to Basics](#)).

Bastantes de las personas invitadas no pudieron asistir finalmente, con lo que al final quedaron en 17 asistentes.

El Manifiesto Ágil

Del 11 al 13 de febrero de 2001 tuvo lugar la reunión en el resort de Snowbird. Los participantes pusieron ideas en común, escribieron tarjetas, debatieron y terminaron con dos cosas concretas que pocos de ellos esperaban al comenzar: un manifiesto y una palabra que agrupara todo el movimiento: "Agile".



El manifiesto ocupaba una pizarra. Lo vemos en la parte derecha de la fotografía. En la parte izquierda la famosa fotografía que hace de background en la web en la que está publicada el manifiesto:
<http://agilemanifesto.org>.

La palabra "Agile" se escogió después de descartar algunas otras como "Light Weight" o "Adaptive". A casi nadie le gustaba el nombre de "ligeros". Comenta Fowler en el artículo mencionado anteriormente que el ser ligero no era el objetivo de los métodos, era sólo un síntoma. Al final se consideró que la palabra "ágil" capturaba bien la adaptatividad y la respuesta al cambio que se consideraba importante para todos los enfoques.

Firmantes del manifiesto

Los firmantes del manifiesto eran representantes de las principales metodologías *ligeras* existentes en la época.

- **XP:** Kent Beck, Bob Martin, Ron Jeffries, Ward Cunningham, Martin Fowler, James Grenning

- **Scrum:** Ken Schwaber, Mike Beedle, Jeff Sutherland
- **Feature Driven Development:** Jon Kern
- **Dynamic System Development Method:** Arie van Bennekum
- **Crystal Processes:** Alistair Cockburn
- **Pragmatic Programmers:** Andy Hunt, Dave Thomas
- **Model Driven:** Steve Mellor
- **Consultores:** Brian Marie y Jim Highsmith

Muchos de los firmantes del manifiesto siguen activos en la actualidad, en diversas organizaciones o en redes sociales. A continuación los listamos por orden alfabético, con enlaces a su cuenta de Twitter y la organización en la que participan.

- Kent Beck ([Twitter](#))
- Alistair Cockburn ([Twitter](#), [Blog](#))
- Ward Cunningham ([Twitter](#))
- Martin Fowler ([Twitter](#), [Blog](#))
- Andrew Hunt ([Twitter](#), [Pragmatic Programmer](#))
- Ron Jeffries ([Twitter](#), [Blog](#))
- Robert C. Martin ([Twitter](#), [Clean Code](#))
- Ken Schwaber ([Twitter](#), [Scrum.org](#))
- Jeff Sutherland ([Twitter](#), [Scrum.org](#))
- Dave Thomas ([Twitter](#), [Pragmatic Programmer](#))

Valores del Manifiesto Ágil

El texto principal del manifiesto lista las cuatro creencias fundamentales en forma de **valores**, contraponiéndolas a elementos de las metodologías tradicionales. El manifiesto establece claramente que se prefiere los primeros, pero que éstos no reemplazan, sino que complementan, los segundos.

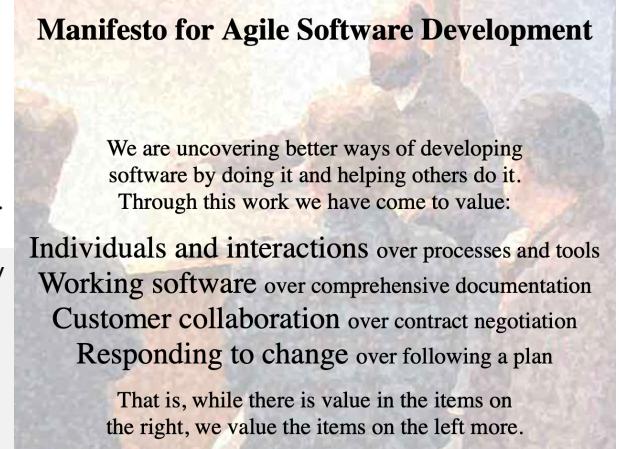
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

El primer gran acierto del manifiesto fue estructurarlo alrededor de valores. Los valores son elementos básicos de una cultura, creencia o metodología, que impregnán todo el resto de componentes.

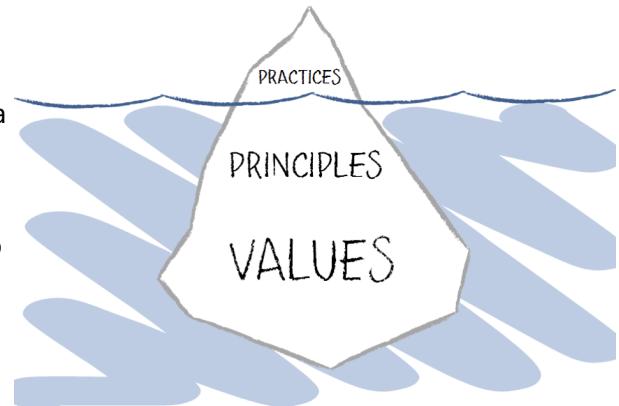
Antes de estudiar en detalle los valores concretos del Manifiesto Ágil vamos a reflexionar brevemente sobre la distinción entre valores, principios y prácticas.



Valores, principios y prácticas

Muchas de las recetas e ideas que se proponen en conferencias, libros o metodologías no son generalmente aplicables a cualquier situación, sino que son dependientes del contexto. Por ejemplo, no es la misma situación la de una pequeña empresa familiar con sólo dos programadores que se encargan de la instalación y mantenimiento de software adquirido que el de una gran empresa con un departamento de informática formado por una veintena de personas.

Por eso es muy importante siempre contemplar las prácticas como ideas o inspiraciones concretas y buscar el conocimiento más abstracto y general que las soportan. Estas ideas más generales son más abstractas, pero también son más generales y aplicables a muchos más contextos.



Por ejemplo, la idea de "colaboración con el cliente" es una idea general que será llevada a cabo de forma distinta en la empresa familiar y en la empresa grande. En la empresa pequeña se puede realizar de forma informal, intercambiando mensajes rápidos y frecuentes por Whatsapp. Pero en la empresa grande habrá que formalizarlo definiendo un responsable que haga el papel de product owner.

La idea de "colaboración con el cliente" es lo que se denomina un valor, un elemento general que guía las acciones y son aplicables a situaciones muy diversas.

En la definición inglesa de la palabra *value* nos encontramos con la acepción que buscamos:

Valor:

2. Principio o estándar de conducta; juicio personal de lo que es importante en la vida.



Hoy en día el término se ha popularizado en el mundo de los negocios. Se habla de **visión** de la empresa y de **valores** asociados a la misma. Por ejemplo, Starbucks define su **misión y valores** de la siguiente forma:

Misión: Inspirar y cuidar el espíritu humano - persona a persona, taza a taza y barrio a barrio.

Valores: Con nuestros socios, nuestro café y nuestros clientes en el centro de nuestra experiencia, vivimos estos valores:

- Crear una cultura de calidez y **de pertenencia**, donde todo el mundo es bienvenido.
- Actuar con **valentía**, enfrentándose al *status quo* y encontrando nuevas formas de hacer crecer nuestra empresa y cada uno de nosotros.
- Estar **presente y conectar** con transparencia, dignidad y respeto.
- Entregar **lo mejor de nosotros** en todo lo que hacemos, pudiendo dar cuenta en todo momento de nuestros resultados.

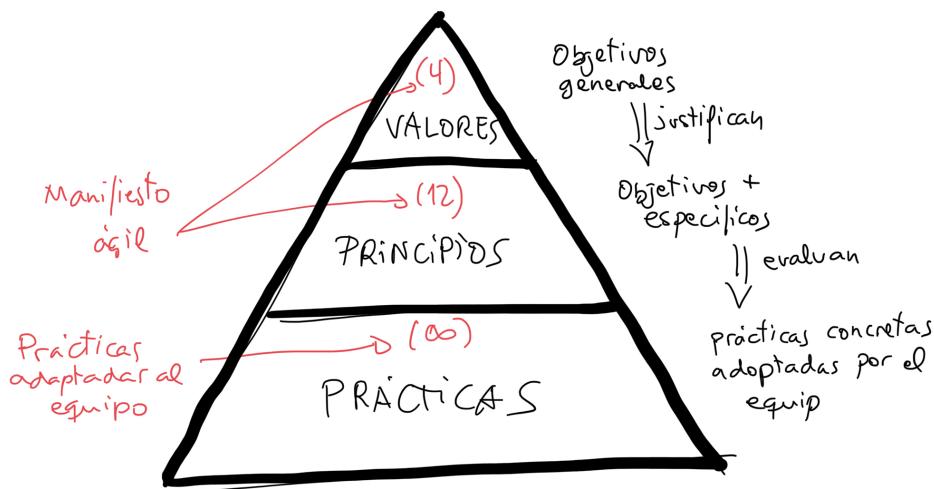
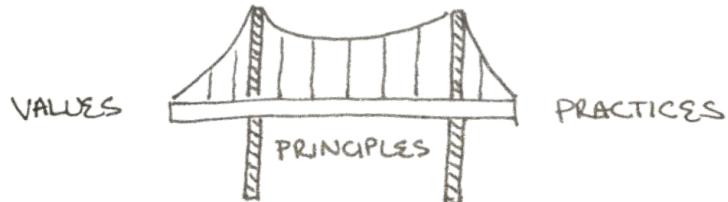
Nuestro objetivo es el rendimiento, a través de la lente del humanismo.

Los **principios** son ideas que se derivan de los valores, menos abstractas, pero no tan concretas como las prácticas. Por ejemplo, en el caso de la "colaboración con el cliente" podríamos definir el siguiente principio:

"Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software valioso"

Estamos dando un ejemplo concreto de colaboración con el cliente. Es una idea menos general que la del valor, pero todavía no es algo tan concreto como una práctica. Por ejemplo, no estamos definiendo la periodicidad, ni la forma de las entregas, ni las acciones que puede hacer el cliente cuando recibe el software.

Kent Beck ilustra la utilidad de los principios con la ilustración que vemos a la derecha. Los principios son como un puente que permiten movernos desde los valores hasta las prácticas. Los principios son más sólidos (concretos) que los valores, que son demasiado abstractos como para poder ser puestos en marcha directamente por las prácticas.



La figura de la izquierda muestra cómo se pueden definir los valores, principios y prácticas en una pirámide de objetivos. En la parte superior, se encuentran los valores, que son un número pequeño de objetivos generales (4 en el manifiesto ágil) que justifican los objetivos más específicos y numerosos en el siguiente nivel (principios, 12 en el manifiesto ágil). Por

último, el número de posibles prácticas son casi infinitas, ya que variarán según el equipo de desarrollo. Pero, en cualquier caso, deben evaluarse en función de si sirven o no para mejorar los objetivos definidos por los principios.

Es fundamental conocer los valores y principios subyacentes a las prácticas para no caer en la repetición vacía de las mismas y en "rituales ágiles" que son cualquier cosa menos ágiles. La aplicación ciega de las prácticas, sin contemplar el contexto de la situación, puede terminar empeorando el proceso desarrollo. Los valores y principios asociados a las prácticas nos permiten razonar sobre el por qué de su aplicación y decidir si es o no conveniente.

Los valores y principios también proporcionan una visión común a todos los miembros implicados en el desarrollo del proyecto. Muchas veces los objetivos a corto plazo de distintos miembros del proyecto entran en conflicto y la mejor forma de arbitrar soluciones es usando los valores y principios comunes. Si no existen esos valores comunes se crea el problema de la *perspectiva fracturada*. En esta situación, cada miembro del equipo tiene una forma distinta de ver las distintas prácticas de desarrollo y se crean disfunciones y conflictos.

Por ejemplo, si una idea compartida por todos los miembros del equipo es la de "entregar valor" al cliente, el equipo de desarrollo frenará su tendencia a crear una arquitectura de software superavanzada y especializada

y se centrará en intentar añadir al software algo que entregar al cliente en la siguiente iteración.

La visión común de los valores y principios también ayuda a unificar las distintas prácticas que se suelen aplicar en los distintos roles del desarrollo de software. En la figura de la derecha, tomada del libro *Learning Agile*, se muestran prácticas ágiles que habitualmente utilizan los managers, product oners, scrum masters (o team leaders) y desarrolladores.

Veamos a continuación uno a uno los cuatro valores del manifiesto ágil, explicando cada uno de ellos en cierta profundidad.

Individuos e interacciones sobre procesos y herramientas

La mejor garantía de que el proyecto tenga éxito es que el equipo que lo desarrolla funcione bien. Los métodos, procesos y herramientas son secundarios. Pueden usarse bien o mal. Es más importante fijarse en los miembros del equipo, sus motivaciones, preferencias e interacciones.

Las metodologías tradicionales hacen gran énfasis en la documentación escrita. Sin embargo, más importante que una buena documentación es una buena interacción (comunicación continua para que todo el equipo esté informado de las decisiones, temas abiertos, conceptos de negocio, etc.).

Cuando no existen problemas de comunicación los equipos funcionan mucho mejor (en algunos estudios se habla de hasta 50 veces mejor que la media). Para facilitar la comunicación las metodologías ágiles se basan en ciclos frecuentes de inspeccionar-y-adaptar. Estos ciclos van desde cada pocos minutos con el *pair programming*, a cada pocas horas con la integración continua, a cada día con las reuniones diarias *standup*, a cada iteración con la revisión y la retrospectiva.

Para que funcione bien la comunicación y los ciclos de inspeccionar-y-adaptar es necesario que los miembros del equipo muestren bastantes conductas claves:

- respeto por el bienestar de cada persona
- verdad en cada comunicación
- transparencia en todos los datos, acciones y decisiones
- confianza en que cada persona va a apoyar al equipo
- compromiso al equipo y a los objetivos del equipo

Para promover este tipo de conducta, se debe facilitar un entorno que apoye y sea inclusivo.

Se debe buscar de forma deliberada este tipo de comportamientos, porque la mayoría de equipos evitan la verdad, la transparencia y la confianza debido a normas culturales o conflictos previos generados por ser honestos en la comunicación. Para combatir estas tendencias, los líderes y los miembros del equipo deben facilitar el conflicto positivo.



Cuando los equipos no esconden el conflicto, sino que se enfrentan a él de forma positiva se obtienen muchos beneficios:

- La mejora de los procesos depende de que los equipos generen una lista de impedimentos o problemas en la organización, enfrentarse a ellos de forma clara priorizándolos y eliminándolos sistemáticamente.
- La innovación sucede únicamente como consecuencia del libre intercambio de ideas enfrentadas.
- La resolución de intereses enfrentados es consecuencia de que los equipos se alinean alrededor de objetivos comunes y exponen sus preocupaciones y potenciales conflictos.
- El compromiso del trabajo conjunto sucede sólo cuando la gente se pone de acuerdo en objetivos comunes y se esfuerzan en mejorarlo tanto individualmente como en equipo.

Software en funcionamiento sobre documentación exhaustiva

Frente a la documentación exhaustiva propia de los métodos tradicionales, el manifiesto ágil promueve la idea del software en funcionamiento, siendo utilizado por los clientes.

El software que funciona es software que proporciona valor. En muchas ocasiones este valor se puede calcular en forma de dinero: los clientes ganan más con el software de lo que les ha costado comprarlo. Nosotros ganamos más con él de lo que nos ha costado desarrollarlo.

Hay mucha documentación necesaria: manuales de usuario, documentación técnica que se va a consultar. Sin embargo, hay que eliminar la documentación que no se va a usar y que no aporta nada a lo que ya sabemos.

Por otro lado, un buen código es la mejor documentación. Cuando se usa TDD, primero se hacen las pruebas y éstas sirven para validar el sistema y para documentar. Los ejemplos de validación hechos por el *product owner* y los clientes son otro ejemplo de documentación imprescindible.

El equipo debe definir lo que considera "software que funciona", esto es, definir claramente cuando considera que una determinada característica está terminada y lista para salir a producción. En un alto nivel, un trozo de funcionalidad está completo cuando sus características pasan todos los tests y puede ser utilizado por el usuario final. Como mínimo se deben realizar tests unitarios y tests a nivel de sistema. Los mejores equipos incluirán también en la definición de terminado tests de integración, tests de eficiencia y tests de aceptación por el cliente.

Es conveniente definir tests de aceptación cuando se está definiendo una nueva característica, ejecutarlos tan pronto la característica se haya terminado de implementar y corregir *bugs* identificados como de alta prioridad tan pronto como sea posible.

Colaboración con el cliente sobre negociación de contratos

Se promueve la colaboración con el cliente como forma de que éste termine satisfecho. Un contrato tradicional obliga a definir a priori con todo detalle el conjunto de requisitos del sistema y hacer una planificación a largo plazo que siempre va a ser muy difícil de cumplir. Sobre todo en procesos no deterministas como el desarrollo de software.

Esto se aplica también al trabajo dentro de la empresa. Obligar a que "me den el encargo de trabajo por escrito" para poder después cubrirme las espaldas si hay un error no es un ejemplo de colaboración.

La flexibilidad y apertura de la colaboración permite cometer fallos sin que nadie se sienta señalado. El equipo es el responsable porque hay un objetivo final en el todos estamos comprometidos.

Muchas veces no es posible trabajar mano a mano con el cliente final, por lo que se crea la figura de un "proxy", llamado en Scrum Product Owner. Es el encargado de concretar la lista de características que debe tener el producto y priorizarlas.

El product owner es un miembro más del equipo: participa en las reuniones, propone ideas, proporciona ejemplos de prueba y, lo más importante, se siente tan autor del producto final como los demás miembros del equipo.

Responder al cambio sobre seguir un plan

Responder y abrazar el cambio es una de las características fundamentales de XP, que se incorpora al manifiesto ágil.

Habrá un plan general, pero flexible y adaptable a los cambios que se puedan introducir en el desarrollo (sobre todo si es un proyecto largo).

Hay que dejar de ver los cambios como errores y verlos como oportunidades de entregar más valor. La actitud del gestor del proyecto, por tanto, debe ser la de estar continuamente comprobando y reaccionando (sobre todo al final de cada iteración), no sólo la de planificar una vez al principio.

Los planes de los equipos ágiles se centran en entregar primero el mayor valor de negocio. Debido a que el 80% del valor reside en el 20% de las características, los equipos ágiles tienden a tener listo pronto un producto mínimo que proporcionará un valor claro al cliente. De esta forma se evita el riesgo de no entregar el producto en plazo o de tener que cancelarlo.

Los equipos ágiles se basan el conocimiento de que, para que funcionen correctamente, los planes deben cambiar y adaptarse. Por eso tienen establecido procesos, como revisiones y retrospectivas, que están diseñados específicamente para modificar las prioridades de forma regular basándose en la retroalimentación del cliente y en el valor de negocio.

Para que sea posible el cambio el software debe estar construido correctamente, haciendo frecuentes refactorizaciones que eviten la acumulación de deuda técnica. Si dejamos que los parches y las soluciones rápidas se queden en el código, tarde o temprano esto repercutirá en el proyecto y cada vez será más complicado introducir nuevas características. El software se hará cada vez más rígido y dejará de ser fluido y "soft". Y la velocidad de desarrollo se hará haciendo cada vez mayor.

Principios ágiles

Durante la reunión de Snowbird se elaboró sólo la parte del manifiesto dedicada a los valores. Una vez terminada la reunión se trabajó durante unas semanas para terminar de configurar la web en la que se publicaría el manifiesto. Se intercambiaron correos electrónicos entre los participantes y se terminó definiendo una lista de 12 principios que terminan de configurar y definir el manifiesto.

Principles behind the Agile Manifesto

We follow these principles:

- ① Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- ② Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- ③ Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- ④ Business people and developers must work together daily throughout the project.
- ⑤ Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

- ⑥ The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- ⑦ Working software is the primary measure of progress.
- ⑧ Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- ⑨ Continuous attention to technical excellence and good design enhances agility.
- ⑩ Simplicity--the art of maximizing the amount of work not done--is essential.
- ⑪ The best architectures, requirements, and designs emerge from self-organizing teams.
- ⑫ At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1. Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.

El principio incluye tres ideas juntas: lanzar pronto el software, entregar valor continuamente y satisfacer al cliente.

Por mucho que se intenten especificar los requisitos del software, el hecho es que hasta que el cliente no tiene en sus manos software que funciona no van a poder visualizar correctamente cómo éste va a funcionar y qué valor concreto puede proporcionarles. Cualquier aproximación a priori del valor del software no es más que una estimación, que se concretará cuando se ponga éste en funcionamiento.

Por eso, cuanto antes entreguemos software al cliente, antes podrá concretar sus estimaciones sobre su valor y reaccionar frente a ese nuevo conocimiento.

Además, el software entregado es valor real que ponemos en manos del cliente. Aunque no esté terminado, el cliente podrá aprovechar parte de su funcionalidad.

Puede ser que los clientes no estén acostumbrados a trabajar de esta forma, con software incompleto. Pero deben entender que esta es una nueva forma de colaborar. En lugar de especificar todo a priori en un contrato, el cliente va a poder decidir qué funcionalidades son las que quiere incorporar en la siguiente iteración. De esta forma, todos ganan en el corto plazo porque se entrega pronto valor, y en el largo plazo, porque se maximiza el valor entregado en el producto final.

2. Damos la bienvenida a requisitos cambiantes, incluso al final del desarrollo. Los procesos ágiles aprovechan el cambio para la mejora competitiva del cliente.

La mayoría de desarrolladores tienen problemas con este principio: no es fácil tener que modificar código que ya está hecho. Y menos si no es por nuestra culpa. Hemos puesto mucho cuidado en entregar un conjunto de funcionalidades, le hemos dedicado muchas horas de trabajo, y ahora nos dicen que no están bien y que hay que cambiarlas. Es difícil no sentirse afectado emocionalmente. Pero dar la bienvenida al cambio es una de las herramientas más poderosas del desarrollo ágil.

Hay que empezar viéndolo desde la perspectiva del cliente. Para el cliente tampoco es fácil pedir un cambio. El cliente sabe que si incorpora algún cambio, otras cosas se van a quedar fuera del proyecto. Muchas veces lo hace por necesidad (ventaja competitiva), porque algo ha cambiado en el negocio desde que se inició el proyecto. Por ejemplo, ha habido un cambio legal o ha surgido alguna nueva tendencia en la sociedad que es interesante incorporar al producto. O porque después de probar con las funcionalidades entregadas se ha dado cuenta de que el valor real se encontraba en otras funcionalidades que se habían desestimado al principio.

Es normal que haya que hacer cambios cuando las tareas que conlleva el desarrollo son tan complicadas. Al equipo se le pide que lea la mente del cliente y al cliente se le pide que sea capaz de ver el futuro. Son dos cosas obviamente imposibles. Cuando lo miramos desde este punto de vista es más fácil aceptar los cambios.

Hay que pensar que el producto obtenido después de los cambios va a proporcionar más valor al cliente y que tiene características que lo hacen destacar frente a los productos de la competencia (que no ha tenido tiempo de introducir los cambios en el negocio).

¿Qué significa dar la bienvenida a los cambios? Significa:

- No hay repercusiones negativas cuando hay que hacer cambios. Reconocemos que somos humanos y promovemos un ambiente en el que está permitido tener errores y corregirlos frecuentemente, en lugar de esperar que las cosas salgan perfectas a la primera.
- Estamos todos juntos en esto. El equipo, los clientes, gerencia. Si los requisitos están mal no tiene sentido buscar culpables.
- Cambiamos lo antes posible. Si descubrimos algo que mejorar lo intentamos corregir lo más pronto posible.
- Dejamos de pensar en que los cambios son equivocaciones. Son una forma de aprender sobre el producto que estamos desarrollando.

3. Entregar software que funciona frecuentemente, desde un par de semanas a un par de meses, siendo preferible la escala de tiempo más corta.

Ya hemos hablado de las ventajas de las iteraciones pequeñas como forma de gestionar un proceso no predecible. Al final de cada iteración tenemos una retroalimentación que nos permite aprender más sobre la implementación del proyecto y sobre lo que da valor al cliente.

En cierta manera, el estilo de planificación de los equipos ágiles se parece al utilizado por los equipos militares denominado [command-and-control](#). Para llevar a cabo una misión militar se deben dar órdenes (*commands*) al equipo y se debe gestionar (*control*) los resultados y cambios a corto plazo, monitorizando la evolución del equipo con respecto al objetivo a largo plazo.

En este tipo de planificación a corto plazo hay que tener cuidado de no perder la visión de más alto nivel y realizar también una revisión a largo plazo con cierta periodicidad.

4. La gente del negocio y los desarrolladores deben trabajar juntos diariamente y a lo largo de todo el proyecto.

Un problema es que la gente de negocio (los clientes) tienen un trabajo que hacer, distinto de ayudar a los desarrolladores. Pero el problema es crítico: cada correo electrónico sin contestar retrasa el proyecto.

La gente de negocio debe entender que el equipo va a entregar software valioso para la empresa, que va a solucionar parte de sus problemas y que va a merecer la pena ayudar, y formar parte del equipo (normalmente a través del *product owner*).

Por eso es importante que el equipo priorice las características de más valor.

Un buen *product owner* puede ayudar a reducir la cantidad de tiempo que la gente de negocio pasa con el equipo. Puede que sea necesario que se reúnan diariamente, pero las reuniones deben estar preparadas y servir principalmente para validar información preparada por el *product owner*.

En algunos casos es posible tener a alguien de negocio trabajando junto con el equipo. Su función sería aclarar requisitos que pueden estar confusos y definir las especificaciones de los tests de aceptación.

5. Construir los proyectos alrededor de individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en que van a hacer su trabajo.

Los proyectos funcionan mejor cuando todo el mundo en la empresa reconoce que el equipo está construyendo software valioso y cuando todos en el equipo reconocen que el producto que se está entregando proporciona valor.

La motivación del equipo es fundamental para crear un ambiente en el que todos los miembros se sienten valorados y se facilita la comunicación y la compartición de objetivos.

Hay que tener cuidado con las motivaciones extrínsecas en forma de incentivos basados en evaluaciones individuales que funcionan en contra del equipo como un todo. Por ejemplo, penalizaciones por el número de bugs del producto o premios por el número de líneas escritas.

Si cada uno en el equipo intenta cubrirse las espaldas y acusar a otros de los problemas, se genera una atmósfera perniciosa para el desarrollo.

Al final, el rendimiento individual debería estar basado en lo que entrega el equipo, más que en el rol específico que cada uno ha jugado. La motivación por el proyecto que se está desarrollando y la confianza son valores que hay que fomentar.

El líder del equipo debe aplicar técnicas de trabajo en grupo para conseguir más confianza y sinceridad.

6. La forma más eficiente y efectiva de compartir información con y dentro de un equipo de desarrollo son las conversaciones cara-a-cara.

Las conversaciones son más valiosas que la documentación, permiten resolver dudas y transmitir información de forma efectiva.

Pero, por otro lado, es poco eficiente tener que estar repitiendo la misma información una y otra vez en conversaciones individuales. Si la información está escrita en alguna documentación o en el código, es conveniente ahorrar tiempo revisando esas fuentes antes de la conversación. Respeta el tiempo de los demás y prepara bien la conversación para que sea eficiente.

El fin último de las conversaciones es crear un sentimiento de comunidad de forma que haya un montón de conocimiento implícito que no sea necesario comunicar una y otra vez.

7. El software en funcionamiento es la primera medida de progreso.

La mejor forma de medir el progreso del proyecto es comprobando la cantidad de funcionalidades implementadas y probadas.

En el momento en que ves el software funcionando, lo "pillas". Puedes comprobar lo que se ha hecho y lo que falta por hacer.

Al probar el software al final de cada iteración todo el mundo se hace una idea mucho mejor del progreso que al leer informes y diagramas.

8. Los procesos ágiles promueven un desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deberían poder mantener un ritmo constante de forma indefinida.

No caer en la práctica habitual de las horas extras y los fines de semana cuando se acerca la fecha de entrega. A largo plazo esto no funciona.

El desarrollo iterativo es mucho más realista, porque un ritmo de desarrollo sostenido durante la iteración nos permite estimar mucho más fielmente lo que se va a entregar al final de las dos, cuatro o seis semanas que dure la iteración.

9. La atención continua a la excelencia técnica y al buen diseño mejor la agilidad.

Es importante resolver los bugs tan pronto como aparecen. Cuanto más se tarde en eliminar un bug más difícil es hacerlo.

Hay que utilizar buenas prácticas de diseño, buenas herramientas. Pero no sobre-diseñar.

10. Simplicidad --el arte de maximizar la cantidad de trabajo no hecho-- es esencial.

Borrar código no es una operación demasiado destructiva, porque siempre lo puedes recuperar del sistema de control de versiones. Es peor escribir código de más.

Cuanta más líneas de código, más dependencias y más difícil solucionar los errores, ampliar las funcionalidades y realizar cambios.

11. Las mejores arquitecturas, requisitos, y diseños emergen de equipos auto-organizados.

Lo contrario de un equipo auto-organizado es un equipo que obedece ciegamente los diseños propuestos en un proceso rígido como el de cascada. En un equipo ágil todo el equipo comparte la responsabilidad de la arquitectura del proyecto.

En lugar de un gran diseño al principio de todo, el diseño va emergiendo de forma incremental, conforme se van desarrollando historias de usuario (de mayor a menor valor). Esto obliga a usar técnicas de diseño que permitan construir el sistema poco a poco, ampliando los módulos, esquemas de la base de datos, etc.

Equipos multi-funcionales.

12. A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo, y ajusta su conducta de forma acorde.

El equipo ágil no sólo debe mejorar el software de forma continua, sino que también debe mejorar la propia forma de construir el software. En las reuniones denominadas "retrospectivas" el equipo evalúa lo que ha funcionado y lo que no en la última iteración.

No sólo se debe pensar en qué mejorar a nivel técnico, sino a nivel de proceso. ¿Qué podemos hacer para que no hayan tantos bugs? ¿Cuál es el problema que ha causado que no haya funcionado la última demo? Es importante no repartir culpas, sino buscar las causas últimas (*root causes*) de los problemas. Una técnica muy útil es la de los [5 por qués](#): a cada respuesta sobre la causa de un problema se hace una pregunta de ¿por qué?.

Al principio es algo incómodo hablar de errores y cosas que se han hecho mal. Pero con el tiempo la gente se acostumbra a ello y se convierte en una forma de mejorar a base de hacer críticas constructivas (y valorar lo que se ha hecho bien).

Existen una gran cantidad de técnicas específicas para facilitar retrospectivas. Algunos libros sobre el tema:

- [Improving Agile Retrospectives: Helping Teams Become More Efficient](#)
- [Agile Retrospectives](#)
- [Project Retrospectives: A Handbook for Team Reviews](#)

Referencias

- Craig Larman, Victor Basili (2003) [Iterative and Incremental Development: A Brief History](#)
- Martin Fowler (2006) [Writing The Agile Manifesto](#)
- Robert Martin (2019) [Clean Agile: Back to Basics, cap. 1](#)
- Philipp Hohl y otros (2018) [Back to the future: origins and directions of the "Agile Manifesto" – views of the originators](#)
- Andrew Stellman, Jennifer Greene (2014) [Understanding Agile, cap. 2](#)