

**BodyofKnowledge(BoK)**

Un “cuerpo de conocimiento” es un requisito para calificar a cualquier área de la ingeniería como una profesión y describe el conocimiento relevante para una disciplina.

**SWEBoK(Software Engineering BodyofKnowledge)**

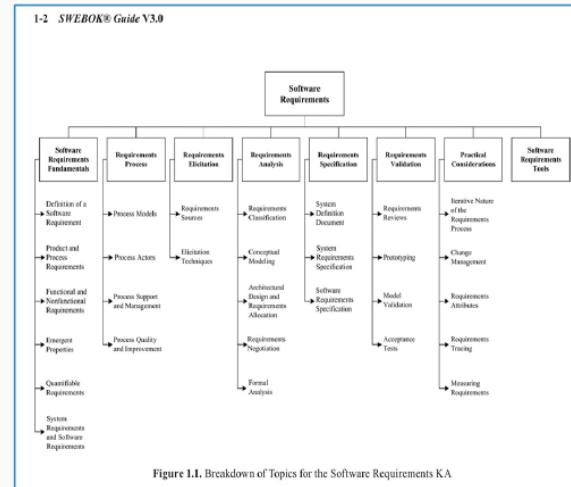
- o Describe el conocimiento que existe de la Ingeniería de SW. Comenzó en 1998, liderado por la IEEE Computer Society, para “convertir a la ingeniería del software en una disciplina legítima y una profesión reconocida”.
- o Está conformado por 15 Áreas de Conocimiento (KnowledgeAreas –KA) que sintetizan conceptos básicos y referencian a información más detallada.
- o La versión actual es SWEBoK3.0 (publicada en el 2014). formada por 15 áreas KA de conocimiento.
- o Tiene reconocimiento internacional como ISO/IEC TechnicalReport19759:2015.

Versión actual 3.0:

## Body of Knowledge (BoK)

### The 15 SWEBOK KAs

1. Software Requirements
2. Software Design
3. Software Construction
4. Software Testing
5. Software Maintenance
6. Software Configuration Management
7. Software Engineering Management
8. Software Engineering Process
9. Software Engineering Models and Methods
10. Software Quality
11. Software Engineering Professional Practice
12. Software Engineering Economics
13. Computing Foundations
14. Mathematical Foundations
15. Engineering Foundations



## Paper - Classic mistakes

Personas, Producto, Proceso y Tecnología.

Mistake	Description
Abandonment of planning under pressure	Projects make plans and then routinely abandon them when they run into schedule trouble. This would not be a problem if the plans were updated to account for the schedule difficulties. The problem arises when the plans are abandoned with no substitute, which tends to make the project slide into code-and-fix mode. <b>Abandonar la planificación cuando hay dificultades o presión por terminar.</b>

Adding people to a late project	When a project is behind, adding people can take more productivity away from existing team members than it adds through new ones. Adding people to a late project has been likened to pouring gasoline on a fire. <a href="#">Aregar gente a un proyecto atrasado, saca productividad ya que necesitan que se les explique como funciona, o curva de aprendizaje de cómo trabaja el equipo/herramientas/negocio.</a> Para que sea efectivo agregarlos necesito que sea bajo tiempo de capacitación por experiencia u otra razón, que las tareas sean independientes/poca coordinación con otras personas y que las tareas que se den sean divisibles.
Assuming global development has a negligible impact on total effort	Multi-site development increases communication and coordination effort between sites. The greater the differences among the sites in terms of time zones, company cultures, and national cultures, the more the total project effort will increase. Some companies naively assume that changing from single-site development to multi-site development will have a negligible impact on effort, but studies have shown that international development will typically increase effort by about 40% compared to single-site development. <a href="#">Asumir que tener personas en distintos países afecta el impacto</a>
Code-like-hell programming	Some organizations think that fast, loose, all-as-you-go coding is a route to rapid development. If the developers are sufficiently motivated, they reason, they can overcome any obstacles. This is far from the truth. The entrepreneurial model is often a cover for the old code and-fix paradigm combined with an ambitious schedule, and that combination almost never works. <a href="#">Codear tan rápido como puedas comiéndote muchos bugs o pasando cosas.</a>
Confusing estimates with targets	Some organizations set schedules based purely on the desirability of business targets without also creating analytically-derived cost or schedule estimates. While target setting is not bad in and of itself, some organizations actually refer to the target as the 'estimate,' which lends it an unwarranted and misleading authenticity as a foundation for creating plans, schedules, and commitments. <a href="#">Confundir estimaciones con objetivos.</a>
Developer goldplating	Developers are fascinated by new technology and are sometimes anxious to try out new capabilities of their language or environment or to create their own implementation of a slick feature they saw in another product—whether or not it's required in their product. The effort required to design, implement, test, document, and support features that are not required adds cost and lengthens the schedule. <a href="#">Hacer funcionalidades que no son tan requeridas por el solo hecho de probar tecnología o cosas de la tecnología que se está usando.</a>
Excessive multitasking	When software developers are assigned to more than one project, they must 'task switch' as they change their focus from one project to another. They must get out of 'flow' on one project and into 'flow' on another. Task switching can be a significant factor—some studies have said that each task switch in software development can incur a 5-30 minute downtime as a developer works out of flow on one project and works into flow on the other. <a href="#">Cambiar o tener más de un proyecto trae problemas de productividad y cuesta seguir el hilo rápido.</a>
Feature creep	The average project experiences about a 25-percent change in requirements over its lifetime. Such a change produces at least a 25-percent addition to the software effort and schedule, which is often unaccounted for in the project's plans and unacknowledged in the project's status reports. <a href="#">No se contempló el 25% de porcentaje de adiciones de requerimientos/cambio de planificación en la planificación inicial por ende ante nuevos cambios se atrasó todo porque no hay tiempo contemplado desde el comienzo para eso.</a>
Friction between developers and customers	Friction between developers and customers can arise in several ways. Customers may feel that developers are not cooperative when they refuse to sign up for the development schedule that the customers want or when they fail to deliver on their promises. Developers may feel that customers are unreasonably insisting on unrealistic schedules or requirements changes after requirements have been baselined. There might simply be personality conflicts between the two groups. The primary effect of this friction is poor communication, and the secondary effects of poor communication include poorly understood requirements, poor user-interface design, and, in the worst case, customers' refusing to accept the completed product.
Heroics	Some project teams place a high emphasis on project heroics, thinking that the certain kinds of heroics can be beneficial. However, emphasizing heroics in any form usually does more harm than good. Sometimes there is a higher premium placed on a can-do attitudes than on steady and consistent progress and meaningful

	<p>progress reporting. By elevating can-do attitudes above accurate-and-sometimes-gloomy status reporting, such project managers undercut their ability to take corrective action. They don't even know they need to take corrective action until the damage has been done. Can-do attitudes can escalate minor setbacks into true disasters. An emphasis on heroics can encourage extreme risk taking and discourage cooperation among the many stakeholders in the software development process.</p> <p><b>El programador muy experimentado que salva las papas siempre.</b></p>
Inadequate design	A special case of shortchanging upstream activities is inadequate design. Rush projects undermine design by not allocating enough time for it and by creating a pressure-cooker environment that makes thoughtful consideration of design alternatives difficult. This results in going through several time-consuming design cycles before the system can be completed.
Insufficient planning	Planning can be done well, and planning can be done poorly. But some projects suffer from simply not doing enough planning at all, i.e., not prioritizing planning as an important activity.
Insufficient risk management	<p>Some mistakes have been made often enough to be considered classic mistakes. Other potential problems need to be identified project-by project through risk management. The most common problem with risk management is not doing any risk management at all. The second most common problem with risk management is not doing enough risk management.</p> <p><b>No se hace manejo de riesgos para riesgos ocurentes/classic mistakes que deben considerarse si o si. El problema es no hacer gestión de riesgos o que no sea suficiente.</b></p>
Lack of automated source-code control	<p>Failure to use automated source-code control exposes projects to needless risks. Without it, if two developers are working on the same part of the program, they have to coordinate their work manually and risk accidentally overwriting someone else's work. People develop new code to out-of-date interfaces and then have to redesign their code when they discover that they were using the wrong version of the interface. Users report defects that you can't reproduce because you have no way to recreate the build they were using.</p> <p><b>no usar sistemas como GIT de control de versiones entonces se pueden sobreescribir o pisar cosas del mismo archivo sin control.</b></p>
Lack of effective project sponsorship (PM)	<p>High-level project sponsorship is necessary to support many aspects of effective development including realistic estimates, adequate resource allocation, and achievable schedules, as well as helping to clear roadblocks once the project is underway. Without an effective project sponsor, other high-level personnel in your organization can force you to accept unrealistic deadlines or make changes that undermine your project.</p> <p><b>Sin un sponsor bien se pueden aceptar estimaciones o tiempos límites poco realistas, o aceptar cambios que traen problemas en el proyecto.</b></p>
Lack of stakeholder buy-in	All of the major players in a software-development effort must buy into the project. That includes the executive sponsor, team leader, team members, marketing, end-users, customers, and anyone else who has a stake in it. The close cooperation that occurs only when you have complete buy-in from all stakeholders allows for precise coordination of a software development effort that is impossible to attain without good buy-in. <p><b>PM, team, marketing, usuarios, clientes. No hay coordinación ni conformidad de todos o la mayoría.</b></p>
Lack of user involvement	User involvement is necessary for defining meaningful requirements. The degree of user involvement can affect how quickly or how slowly issues get resolved.
Letting a team go dark	<p>On some projects, management allows a team to work with little oversight and little visibility into the team's progress. This is known as "letting a team go dark." This practice restricts visibility into project status, and the project doesn't receive timely warnings of impending schedule slips. Before you can keep a project on track, you have to be able to tell whether it's on track, and letting a team go dark prevents that.</p> <p><b>Cuando trabajan medio a oscuras, no saben el estado del proyecto, o tiempos límites, impedimentos</b></p>
Noisy, crowded offices	About 60 percent of developers report that their work environments are neither sufficiently quiet nor sufficiently private. For many developers, this can prevent concentration and prevent achieving a state of 'flow' that is helpful in achieving high levels of productivity. Workers who occupy quiet, private offices tend to perform significantly better than workers who occupy noisy, crowded work bays or cubicles.

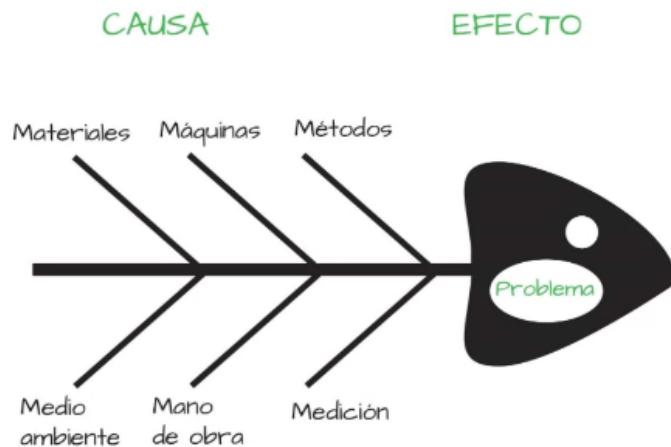
Omitting necessary tasks from estimates	If people don't keep careful records of previous projects, they forget about the less visible tasks, but those tasks add up. Forgotten activities can add 20 to 30 percent to a development schedule. <i>Si no contemplas las tareas necesarias y base pero que no son tan visibles en la descripción de la tarea trae problemas de tiempo.</i>
Outsourcing to reduce cost	Valid reasons to outsource include accessing capabilities that you don't have in house, diversifying your labor force, freeing up your in-house staff to focus on mission-critical or core-competency projects, adding "surge capacity" to your development staff, and supporting around-the-clock development. Many organizations that have outsourced for these reasons have accomplished their objectives. However, some organizations outsource primarily to reduce development costs, and historically those initiatives have not succeeded. Usually outsourcing motivated by cost savings results in higher costs and longer schedules. <i>Tercerizar para reducir costo trae problemas</i>
Overestimated savings from new tools or methods	Organizations often assume that first-time usage of a new tool or method will reduce costs and shorten schedules. In reality, first-time use of a new tool or method tends to be subject to a learning curve, and the safest planning assumption is to assume a short-term increase in cost and schedule before the benefits of the new tool or method kick in. <i>Subestimar el tiempo/curva de aprendizaje de una nueva herramienta o método. Los beneficios de la nueva herramienta se ven más adelante.</i>
Overly optimistic schedules	The challenges faced by someone building a three-month application are quite different than the challenges faced by someone building a one-year application. Setting an overly optimistic schedule sets a project up for failure by under-scoping the project, undermining effective planning, and abbreviating critical upstream development activities such as requirements analysis and design. It also puts excessive pressure on developers, which hurts developer morale and productivity. <i>Subestimar el tiempo, estimar con optimismo proyectos grandes, produce mucha presión en devs, reduce productividad, se subestima la planificación y se recorta en flujos de trabajo críticos</i>
Planning to catch up later	If you're working on a project and it takes you four weeks to meet your first two-week milestone, what's your status? You know that you're behind schedule, but will you stay behind schedule, or will you catch up later? Project planners commonly plan to catch up later, but they rarely do. Most projects that get behind schedule stay behind schedule. <i>Dejar para después cosas que se deberían planificar en el momento.</i>
Politics placed over substance	Larry Constantine reported on four teams that had four different kinds of political orientations. "Politicians" specialized in "managing up"—concentrating on relationships with their managers. "Researchers" concentrated on scouting out and gathering information. "Isolationists" kept to themselves, creating project boundaries that they kept closed to non team members. "Generalists" did a little bit of everything: they tended their relationships with their managers, performed research and scouting activities, and coordinated with other teams through the course of their normal workflow. Constantine reported that initially the political and generalist teams were both well regarded by top management. But after a year and a half, the political team was ranked dead last. Putting politics over results is fatal to software development effectiveness. <i>Usan políticas distintas pero nunca se establece un estándar. Focalizar en resultados y no en políticas.</i>
Premature or too frequent convergence	Shortly before a public software release there is a push to prepare the software for release—improve the product's performance, create final documentation, stub out functionality that's not going to be ready for the release, perform end-to-end testing including tests that can't be automated, test the setup program, and so on. On rush projects, there is a tendency to force convergence too early. If the software isn't close enough to a releasable state, the attempted convergence will fail, and the team will need to attempt to converge again later in the project. The extra convergence attempts waste time and prolong the schedule. <i>Sacar a funcionar un SW que no esté lo suficientemente maduro. Se va a necesitar arreglar muchas cosas y volver a "converger" y es pérdida de tiempo.</i>
Push me, pull me negotiation	One bizarre negotiating ploy occurs when a manager approves a schedule slip on a project that's progressing slower than expected and then adds completely new tasks after the schedule change. The underlying reason for this is hard to fathom because the manager who approves the schedule slip is implicitly acknowledging that the schedule was in error. But once the schedule has been corrected, the same person takes explicit action to make it wrong again.

	<p>Quien hace la planificación y tareas a seguir sabe que están mal pero las aprueba igual sin negociar, luego se tiene que replanificar.</p>
Requirements gold plating	<p>Requirements gold plating is the addition of requirements or the expansion of requirements without a clear business justification. Requirements gold plating can be done by end users who want the “system to end all systems” or it can be done by developers who are sometimes more interested in complex capabilities than real users are.</p> <p><a href="#">Aregar funcionalidades sin “justificación” razonable por parte del cliente</a></p>
Research-oriented development	<p>Some projects have goals that push the state of the art—algorithms, speed, memory usage, and so on. That's fine, but when those projects also have ambitious cost or schedule goals, the combination of advancing the state of the art with a tight budget on a short schedule isn't achievable.</p> <p><a href="#">Querer mejorar la performance al máximo, velocidad, buenos algoritmos, uso de memoria óptimo pero con poco tiempo (good enough)</a></p>
Shortchanged quality assurance	<p>Projects that are in a hurry often cut corners by eliminating design and code reviews, eliminating test planning, and performing only perfunctory testing. It is common for design reviews and code reviews to be given short shrift in order to achieve a perceived schedule advantage. This often results in the project reaching its feature-complete milestone but then still being too buggy to release.</p> <p><a href="#">Recortar en calidad. Ante falta de tiempo se reduce diseño o reviews de código nuevo, se elimina el testeо o se reduce. Produce varios bugs en el release</a></p>
Shortchanged upstream activities	<p>Projects sometimes cut out non-coding activities such as requirements, architecture, and design. Also known as “jumping into coding,” the results of this mistake are predictable. Projects that skimp on upstream activities typically have to do the same work downstream at anywhere from 10 to 100 times the cost of doing it earlier.</p> <p><a href="#">Recortar en tareas que no son código, como diseño, arquitectura o requerimientos. Si no analizas te trae problemas.</a></p>
Silver-bullet syndrome	<p>On some projects, there is an over reliance on the advertised benefits of previously unused technologies, tools, or 3rd party applications and too little information about how well they would do in the current development environment. When project teams latch onto a single new methodology or new technology and expect it to solve their cost, schedule, or quality problems, they are inevitably disappointed.</p> <p><a href="#">Implementar nuevas tecnologías o metodologías pensando que en corto tiempo va a funcionar y arreglar todo.</a></p>
Subcontractor failure	<p>Companies sometimes subcontract pieces of a project when they are too rushed to do the work in-house. (“Subcontractor” can refer either to an individual or to an outsourcing firm.) But subcontractors frequently deliver work that's late, that's of unacceptably low quality, or that fails to meet specifications. Risks such as unstable requirements or ill-defined interfaces can be magnified when you bring a subcontractor into the picture. If the subcontractor relationship isn't managed carefully, the use of subcontractors can undermine a project's goals. (Note: This question deals specifically with subcontracting part of a project—other items focus on outsourcing full projects.)</p> <p><a href="#">Subcontratar piezas o sistemas aparte con tu sistema in-house que trae problemas de calidad, especificaciones incorrectas, riesgos, etc.</a></p>
Switching development tools in the middle of a project	<p>This is an old standby that hardly ever works. Sometimes it can make sense to upgrade incrementally within the same product line, from version 3 to version 3.1 or sometimes even to version 4. But the learning curve, rework, and inevitable mistakes made with a totally new tool usually cancel out any benefit when you're in the middle of a project.</p>
Trusting the map more than the terrain	<p>Project teams sometimes invest more confidence in the plans they create than in the experience their project is giving them. Sometimes they will trust the delivery date written on a plan more than the delivery date implied by the project's track record. If the project reality and the project plans disagree, the project's reality is correct, and the plans must be wrong. The longer a project team trusts the plans rather than the project reality—i.e., trusts the map more than the terrain—the more difficulty they will have adapting their course successfully.</p>

Unclear project vision	The lack of clearly defined and communicated vision undermines the organization's ability to make and execute project-level plans that are consistent with organization-level goals. Without a clear understanding of the vision, people draw their own conclusions about the purpose of the project and how it relates to their day-to-day work, and they make decisions that run counter to the project's business objectives. The unclear vision contributes to changes in project direction, including detailed requirements; detailed plans that are misaligned with project priorities; and, ultimately, inability to meet schedule commitments.
Uncontrolled problem employees	Failure to deal with problem personnel (e.g., a prima donna programmer) can threaten development effectiveness. This is a common problem and has been well-understood at least since Gerald Weinberg published <i>Psychology of Computer Programming</i> in 1971. Failure to take action to deal with a problem employee is the most common complaint that team members have about their leaders.
Undermined motivation	Study after study has shown that motivation probably has a larger effect on productivity and quality than any other factor. On some projects, management can undermine morale throughout the project. Examples include giving a hokey pep talk, going on a long vacation while the team works through the holidays, and providing bonuses that work out to less than a dollar per overtime hour at the end.
Unrealistic expectations	One of the most common causes of friction between developers and their customers or managers is unrealistic expectations. Often customers simply start with unrealistic expectations (which is probably just human nature). Sometimes project managers or developers ask for trouble by getting project approval based on optimistic estimates. A Standish Group survey listed realistic expectations as one of the top five factors needed to ensure the success of an in-house business-software project.
Wasted time during the fuzzy front end	The “fuzzy front end” is the time before the project starts, the time normally spent in the approval and budgeting process. It's not uncommon for a project to spend months or years in the fuzzy front end and then to come out of the gates with an aggressive schedule. <i>Perder tiempo en el comienzo donde se esperan aprobaciones o definiciones de procesos.</i>
Weak personnel	After motivation, either the individual capabilities of the team members or their relationship as a team probably has the greatest influence on productivity. Hiring from the bottom of the barrel can threaten a development effort. On some projects, personnel selections were made with an eye toward who could be hired fastest instead of who would get the most work done over the life of the project. That practice gets the project off to a quick start but doesn't set it up for successful completion.
Wishful thinking	Wishful thinking isn't just optimism. It's closing your eyes and hoping something works when you have no reasonable basis for thinking it will. Wishful thinking at the beginning of a project leads to big blowups at the end of a project. It undermines meaningful planning and can be at the root of other problems.” <i>No optimismo sino pensar/esperar que funcione aunque no hiciste correctamente las cosas.</i>

30/8

**Diagrama causa-efecto:**



Identificando el problema (cabeza) identifica grandes causas de ese problema (Esqueleto). Identificamos la causa raíz.

#### Cinco por qués: Para identificar causa raíz.

Pregunto 5 veces por qué pasa una cosa, uno va descubriendo las diferentes causas raíces.

#### Calidad:

##### ¿Qué es la Calidad?

###### Crosby

- Cumplir con los requerimientos (requisitos)

###### Weinberg

- Cumplir con los requerimientos de alguna persona
  - Calidad es valor para alguna persona
  - Valor es aquello que está dispuesto a pagar para obtener sus requerimientos



###### Juran

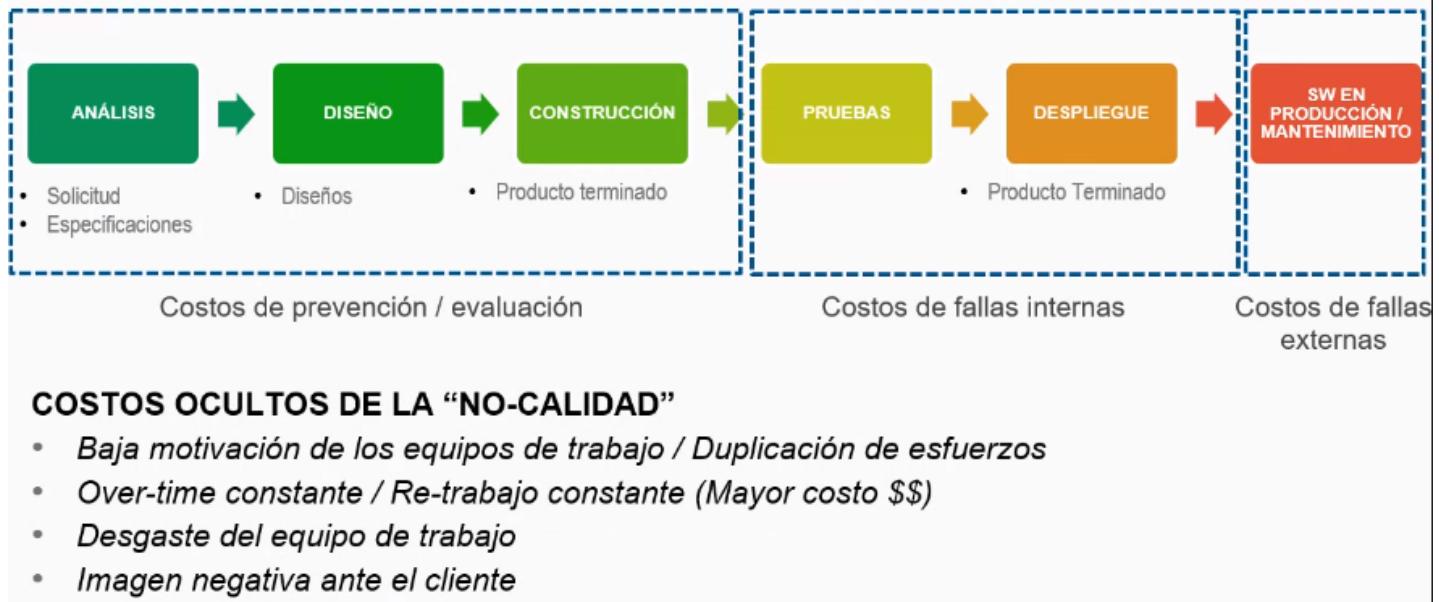
- Adecuación al uso
  - Satisfacción de las necesidades del cliente
  - Ausencia de deficiencias

###### ISO 8402-1986

- La totalidad de aspectos y características de un producto o servicio que se sustentan en su capacidad de cumplir las necesidades especificadas o implícitas

Definiciones distintas para autores distintos.

## Fundamentos de Calidad – Costo de la NO Calidad



### COSTOS OCULTOS DE LA “NO-CALIDAD”

- *Baja motivación de los equipos de trabajo / Duplicación de esfuerzos*
- *Over-time constante / Re-trabajo constante (Mayor costo \$\$)*
- *Desgaste del equipo de trabajo*
- *Imagen negativa ante el cliente*

Incrementando la primera categoría (prevención), se tiende a reducir la segunda categoría (re-trabajo). Cuando la cantidad de dinero ahorrado evitando defectos en el sistema aumenta más rápidamente que la cantidad invertida para evitar estos defectos, se dice “quality is free”.

Los costes de calidad se dividen en costes de prevención y de evaluación, mientras que los de no calidad se dividen en internos y externos.

- **Calidad Interna:** Es propio del software desarrollado. Comprende: Estructura del programa, reusabilidad, mantenibilidad, legibilidad, complejidad, flexibilidad, testeabilidad, etc.
- **Calidad Externa:** Es lo que se ve externo al software desarrollado. Comprende: Si funciona apropiadamente, costo de mantenimiento, eficiencia, fiabilidad.

### VOF:

- El costo de no calidad puede remediarlo utilizando timebox - **Falso** - Para implementar timebox necesito un equipo motivado, que no tengo en “no-calidad”, necesito alta participación del user que está desgastada por la “no-calidad”. Además el timebox lo usas en desarrollo y acá ya se contempla deploys, testeos, etc, etc.
- Los costos de calidad no se recuperan y deben minimizarse - **Falso** - Si aumentamos costo en prevención/evaluación podemos reducir los costos que re-trabajo

## Visiones de la Calidad

La calidad puede ser percibida desde cinco perspectivas:

### **VISIÓN TRASCENDENTAL**

*La calidad es algo que se puede reconocer pero no se puede definir*

### **VISIÓN DEL USUARIO**

*La calidad es adecuación al propósito*

### **VISIÓN DE LA MANUFACTURA**

*La calidad es conformidad con la especificación*

### **VISIÓN DEL PRODUCTO**

*La calidad está vinculada a las características inherentes del producto*

### **VISIÓN BASADA EN EL VALOR**

*La calidad depende de la cantidad de dinero que el usuario está dispuesto a pagar por el producto*

#### **Visión trascendental:**

Demasiado subjetivas e imposible de tomar decisiones.

Se basa en la percepción de la gente (lo que la sociedad dice)

Relacionada con el marketing.

El SW no se relaciona bien con esta visión sino a productos que apelan a los sentidos.

La menos objetiva de todas.

#### **Visión de usuario:**

Qué tan adecuado para el uso que se va a dar.

Si satisface las necesidades explícitas e implícitas.

Es una visión subjetiva.

#### **Visión de la manufactura:**

Podemos definir un proceso de buenas prácticas donde todos estamos de acuerdo a que si seguimos eso el software tiene calidad.

La calidad es conformidad con la especificación.

Compara el proceso

Si el producto cumple con todos los requerimientos hay "calidad" (no importan las necesidades del usuario ya que se asume que estas están expresadas en los requerimientos).

Busco optimizar mi proceso, por ejemplo, minimizar el desperdicio que se produce en las fábricas. En SW intento que cada paso sea consistente con los demás. Es objetiva.

Es objetivo porque son parámetros fijados para todos.

Mayor calidad => menor costo... "Quality is Free". Toda la calidad que yo inyerto se retribuye al final.

Ej.: ¿Un BMW tiene mayor calidad de manufactura que un Fiat Uno? => No hay respuesta.  
Si ambos productos siguen todos sus requerimientos de fabricación, ambos poseen "calidad", pero no es comparable.

### Visión del producto:

Pienso en los features del producto y sus atributos.

La calidad es en función de la cantidad de estos atributos.

Visión de las ISO 25010.

Visión objetiva porque lo evalúas por ciertos parámetros.

Más características => más calidad => más costo

Ej.: Calidad de una Cámara = Resolución + Sensibilidad a la luz + Capacidad de Almacenamiento.

### Visión basada en el valor:

Depende de la cantidad de dinero que el usuario está dispuesto a pagar por el producto.

"Calidad - precio"

Se busca un precio acorde al juego de la oferta y la demanda.

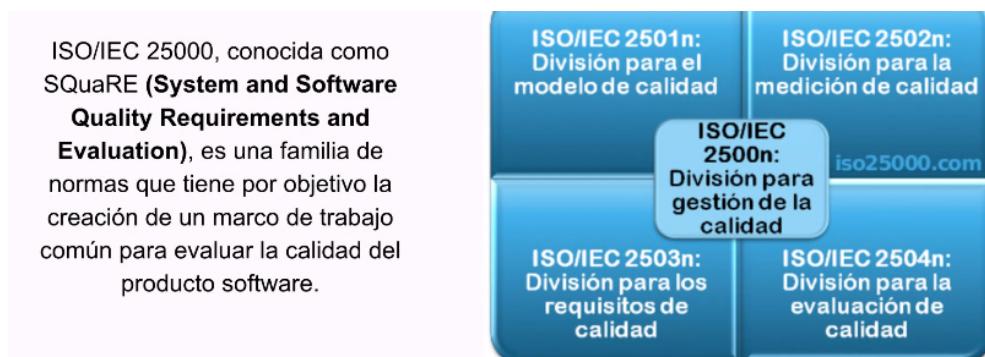
"Algo tiene buena calidad si alguien me compra algo a cierto precio".

Mientras mayor es el costo, su calidad percibida baja. ( Si es muy costoso para el valor que representa su calidad es baja).

Nos sirven para entablar un protocolo entre quienes discuten calidad y ver si hay o no.

## Visión de calidad de producto:

Iso 25010



# ISO 25010 – Calidad del producto de software



6/9

**Diagrama Causa - Efecto:** Permite encontrar las soluciones a las causas del problema.

- Standing On Principles (Karl Wiegers)

Separaciones de la ISO25000 para calidad de producto:

## A.Adecuación funcional:

Capacidad del producto software para proporcionar funciones que satisfacen las necesidades declaradas e implícitas, cuando el producto se usa en las condiciones especificadas (condiciones para las que fue creado).

Tendrá más calidad cuando se adecúe más funcionalmente a lo que se espera de ese producto, hay tres elementos que hacen esto:

1. Completitud funcional: Grado en el cual el conj. de funcionalidades cubre todas las tareas y los objetivos del usuario especificados. Cumpla de manera más abarcativa(todas las tareas y los objetivos del usuario especificados). las funcionalidades.
2. Corrección Funcional: Capacidad del prod. o sist. para proveer resultados correctos con el nivel de precisión requerido.
3. Pertinencia Funcional: Capacidad del prod. o sist. para proporcionar un conjunto apropiado de... .Cuan adecuado/apropiado es algo para lo que se supone que tiene que responder.

## B.Eficiencia de Desempeño:

Habla del comportamiento relativo a los recursos que en determinado contexto o condiciones usa el software que se está evaluando para tener ese comportamiento.

Podemos medir la eficiencia. Ej: eficiencia en tiempo de respuesta, memoria, etc.

3 contribuyentes:

1. Comportamiento temporal: Tiempos de respuesta y procesamiento y los ratios de throughput de un sist cuando lleva a cabo sus funciones bajo condiciones determinadas en relación con un banco de pruebas (benchmark) establecido. Es importante saber desde cuándo y hasta cuándo se debe medir y ver cómo medirlo.
2. Utilización de recursos: Cantidades y tipos de recursos utilizados (memoria, almacenamiento, AB, etc).
3. Capacidad: Grado en que los límites máximos de un parámetro de un prod. o sist. cumplen con los requisitos. Cuáles son los límites máximos que el producto permite soportar respecto a almacenamiento, memoria, AB, etc.

Ej: Cantidad media de rendimiento / tiempo de respuesta

## C.Compatibilidad:

Capacidad de 2 o + sist. o componentes para intercambiar info y/o llevar a cabo sus funciones cuando comparten el mismo entorno HW o SF.

1. Coexistencia: Capacidad del prod. para coexistir con otro SW independiente en un entorno común compartiendo recursos comunes sin detrimiento.
2. Interoperabilidad: Capacidad de 2 o + sist. o comp. para intercambiar información y utilizar la información intercambiada.

## D.Usabilidad:

Capacidad del producto SW para ser entendido, aprendido, usado y resultar atractivo para el usuario, cuando se usa bajo determinadas condiciones.

Que sea amigable al usuario pero que se puede medir en atributos, en esfuerzo,

1. Capacidad para reconocer su adecuación: Capacidad que permite al user entender si el SW es adecuado a sus necesidades. Que permita al usuario entender para qué es. Ej: Un botón con icon o letra poco intuitivo que no entendés para que funciona.
2. Capacidad de aprendizaje: Capacidad que permite al usuario aprender su aplicación. Cuán fácil es aprender o no eso lo podés medir de varias formas como por ejemplo para dos sistemas A y B cuánto tardan en realizar una acción, ver la cantidad de funcionalidades que consumen después de un tiempo, etc.
3. Capacidad para ser usado: Permite al usuario operarlo y controlarlo con facilidad.
4. Protección contra errores de usuario: Proteger a los usuarios de hacer errores.
5. Estética de la interfaz de usuario: Agradar y satisfacer la interacción con el usuario.
6. Accesibilidad: Permite que sea utilizado por usuarios con determinadas características y discapacidades.

## E.Fiabilidad:

Capacidad de un sistema o comp. para desempeñar las funciones especificadas, cuando se usa bajo unas condiciones y período de tiempo determinados.

1. Madurez: Capacidad para satisfacer las necesidades de fiabilidad en condiciones normales. Variable referida a unidad de tiempo y las fallas que se producen en ese tiempo.
2. Disponibilidad: Capacidad de estar operativo y accesible para su uso cuando se requiere.
3. Tolerancia a fallos
4. Capacidad de recuperación:

Ej: Tiempo medio entre fallas, si es corto es poco fiable, falta madurez.

#### **F.Seguridad:**

Capacidad de protección de la info y los datos de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos.

1. Confidencialidad: Proteger contra el acceso de datos e información no autorizados, ya sea accidental o deliberadamente. Que lo accedan los que deben nomas.
2. Integridad: Está debidamente protegida para que no sea alterada.
3. No repudio: Ej: poner logs. Todo queda registrado para no poder negar las acciones.
4. Responsabilidad: Rastrear de forma inequívoca el actor de la acción .
5. Autenticidad: Demostrar la identidad de un sujeto o recurso.

#### **G.Mantenibilidad:**

Capacidad para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas.

1. Modularidad: Capacidad que permite que un cambio en un componente tenga un impacto mínimo en los demás.
2. Reusabilidad:
3. Analizabilidad
4. Capacidad para ser modificado
5. Capacidad para ser probado.

#### **H.Portabilidad:**

Capacidad de ser transferido de forma efectiva y eficiente de un entorno HW, SW, operacional o de utilización a otro.

1. Adaptabilidad: Cuánta mano le tengo que meter para pasarlo de entorno A a B.
2. Capacidad para ser instalado
3. Capacidad para ser reemplazado:

Complejidad ciclomática:

Métrica del software en ingeniería del software que proporciona una medición cuantitativa de la complejidad lógica de un programa.

## Good Enough - conclusión:

La calidad es negociable en función de las necesidades del usuario.

## Standing on Principle - resumen:

No dejar que el PM o customer afecte tu trabajo.

### Integridad e inteligencia con customer:

- Buscar la necesidad real y práctica para lo que pide el cliente que suele estar sesgado por su conocimiento.
- Que no afecte tu forma de trabajar si te toca un customer con conocimientos en sistemas.
- Comunicar fallas o cambios en tiempo/planificación siempre. Si es buena o mala noticia decirla.

### Integridad e inteligencia con manager:

- Estimación, comunicación, responsabilidad y seguridad (si no estoy seguro veo de explotar más requerimientos o dar un rango de tiempo y no algo exacto) en las estimación que se haga.
- Si obligatoriamente tengo que estimar menos tiempo que el requerido, buscar más recursos (personas por ejemplo) o menor complejidad (quitar funcionalidades por ejemplo).

### Dimensiones 5:

Interrelacionadas entre sí.

1. Alcance
2. Tiempo
3. Costo
4. Calidad
5. staff

### Roles 3:

Para categorizar cada dimensión, dan distintas prioridades para que el PM pueda ver que es más o menos flexible.

1. Driver: Objetivo vital a lograr, tiene poco/nada de flexibilidad.
2. Constraint: Ej: Calidad para un software médico, costo para alguien con poca plata o gente si es una empresa con rotación alta.
3. Degree of freedom: Las restantes.

### Conclusión del paper:

- Comunicación y compromiso, siempre diálogo entre el dev + PM, PM + Customer.
- Como PM poder categorizar las dimensiones para ver donde tocar si surge una alteración en el proyecto.

### **Proceso:**

Definición de ciertos roles, herramientas y acciones para obtener una salida.

Conjunto de actividades que la gente usa para desarrollar y mantener software y sus productos asociados.

### **Para que un proceso esté definido:**

1. Guía de pasos
2. herramientas para seguir la guía
3. Roles que van a ejecutar las tareas de esa guía

### **Calidad de Proceso:**

Otra visión de la ing. en software además de la calidad del producto.

Establecer marcos de trabajo para construir software, establecemos pasos y garantizamos calidad si seguimos estos pasos de calidad.

Estamos todos de acuerdo que si seguimos esa forma obtenemos un producto acorde a lo esperado y que cumpla con calidad.

### **CMMI Capability Maturity Model Integrated:**

-Modelo para la mejora y evaluación de los procesos de desarrollo, mantenimiento y operación de SW.

-Es un marco que nos permite construir procesos, se basó en procesos anteriores con buenas prácticas, NO es un proceso para desarrollar SW (NO es una metodología).

-Comparamos nuestras prácticas con las del modelo para ver el nivel de madurez.

Determina la madurez de un proceso y organiza el esfuerzo para mejorarlo describiendo un camino incremental de mejoras

### **Madurez:**

Capacidad organizacional y no individual, de cumplir sistemáticamente con los objetivos. Es el grado en que un proceso está definido, documentado, administrado, controlado, medido y efectivo

Existen dos representaciones de CMMI. El contenido de ambas es el mismo pero su organización es diferente:

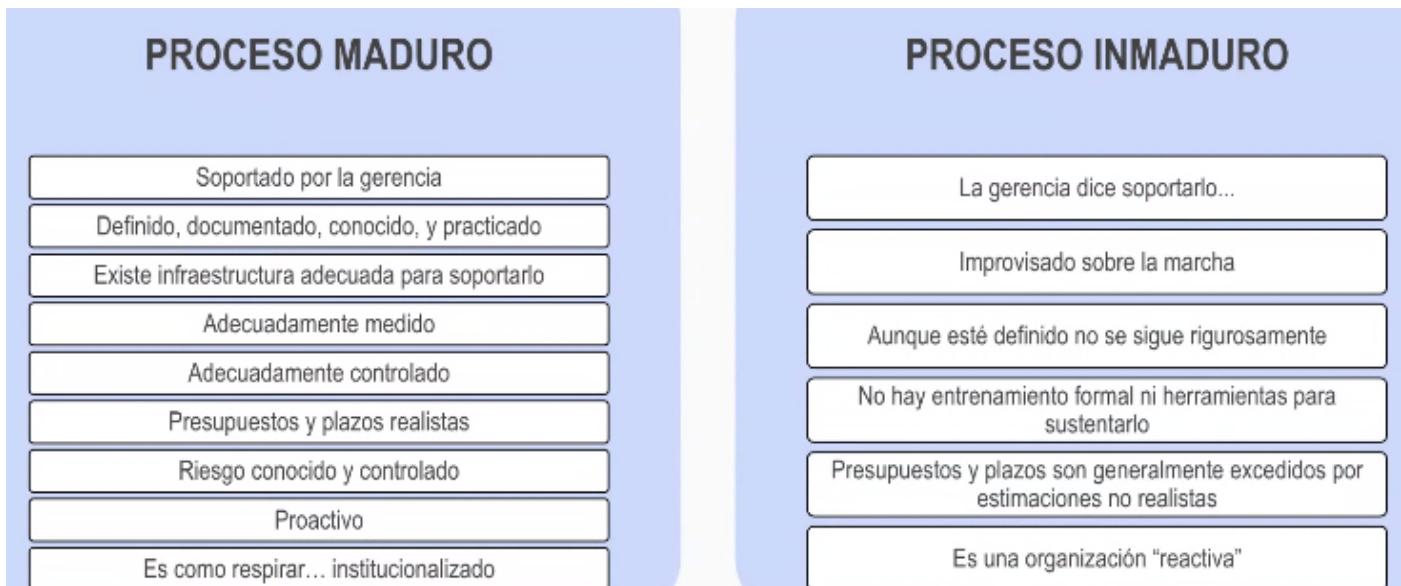
- Enfoque continuo: Provee máxima flexibilidad, se enfoca en las áreas específicas que le incumben a mi organización de acuerdo a los objetivos del negocio.
- Enfoque escalonado/por niveles: Marca un camino de Mejora. Estar en cierto nivel (subconjunto de procesos) en particular me dice cómo trabaja esa empresa.

### **-Cinco (5) niveles de madurez:**

1. **Caótico**: Todas las organizaciones asumen que están acá al empezar a evaluar cómo construir SW.
2. **Repetible**: Empieza a definir qué prácticas serían recomendables incluir en la construcción de SW correspondiente y para aprender. (Gestión de configuración, gestión de riesgos, gestión de configuración, gestión de planificación, etc.)  
Trabaja proyecto a proyecto con formas distintas de trabajar.
3. **Definido**: Los distintos proyectos trabajan de igual forma, con las mismas herramientas
4. **Administrado - cuantitativo**: Empieza a medir todos los trabajos que voy realizando, representado en mediciones, números, etc.
5. **Optimización**: Continuamente voy retroalimentando feedback con buenas prácticas, cómo innovar y agregar nuevas tecnologías para mantener el nivel de madurez.

Nivel	Características	Process Areas
Inicial - Nivel 1	<ul style="list-style-type: none"> <li>• Proyectos caóticos sin entorno estable, incapacidad de repetir sus éxitos, exceden presupuestos y no cumplen calendarios, sin procesos probados.</li> <li>• No dispone de un ambiente estable para el desarrollo y mantenimiento de software</li> <li>• Los procesos se ven minados por falta de planificación</li> <li>• El éxito de los proyectos se basa en el esfuerzo del personal</li> <li>• Se producen fracasos y, casi siempre, retrasos y sobrecostes</li> <li>• No implementa CMMI</li> </ul>	
Repetible - Nivel 2	<ul style="list-style-type: none"> <li>• Dispone de algunas prácticas institucionalizadas de gestión de proyectos</li> <li>• Existen métricas básicas y un razonable seguimiento de calidad</li> <li>• La relación con los clientes está gestionada sistemáticamente</li> <li>• Procesos planificados y realizados de acuerdo a políticas, monitorean, controlan y revisan, las prácticas se mantienen en tiempo de estrés, documentos visibles a la gerencia en hitos importantes.</li> </ul>	<ul style="list-style-type: none"> <li>• Admin. de Requerimientos (maneja el scope de los requerimientos)</li> <li>• Planeamiento de Proyectos</li> <li>• Monitoreo y Control de Proyectos</li> <li>• Administración de Acuerdo con Proveedores</li> <li>• Medición y Análisis</li> <li>• Aseguramiento de Calidad</li> <li>• Admin. de Configuración</li> </ul>
Definido - Nivel 3	<ul style="list-style-type: none"> <li>• Buena gestión de proyectos</li> <li>• Dispone de correctos procedimientos de coordinación entre grupos, formación del personal</li> <li>• Dispone de un nivel más avanzado de métricas en los procesos</li> <li>• Implementa técnicas de revisión por pares.</li> <li>• Procesos bien caracterizados y comprendidos, se describen estándares, se establecen y mejoran con el tiempo.</li> <li>• Diferencias con nivel 2: los procesos son consistentes en toda la organización [en el nivel dos los procesos pueden ser diferentes en cada instancia específica, por ejemplo, en cada proyecto]; los procesos se describen más rigurosamente en el nivel 3 que en el nivel 2. Hay mayor proactividad.</li> </ul>	<ul style="list-style-type: none"> <li>• Desarrollo de Requerimientos</li> <li>• Gestión de Proyectos integrada</li> <li>• Gestión de Riesgos</li> <li>• Integración de Producto</li> <li>• Validación (Cumple con las necesidades del usuario)</li> <li>• Verificación (Satisface el requerimiento)</li> <li>• Solución Técnica</li> <li>• Enfoque del proceso organizacional</li> <li>• Entrenamiento organizacional</li> <li>• Resolución y Análisis de Decisiones</li> <li>• Equipos Integrados</li> <li>• Ambiente Organizacional para la Integración</li> </ul>
Gestionado - Nivel 4	<ul style="list-style-type: none"> <li>• Dispone de un conjunto de métricas significativas de calidad y productividad que se usan de forma sistemática para la toma de decisiones y gestión de riesgos</li> <li>• El software resultante es de alta calidad. Se cuantifica la calidad.</li> </ul>	<ul style="list-style-type: none"> <li>• Desempeño del proceso organizacional</li> <li>• Gestión de Proyecto Cuantitativa.</li> </ul>

	<ul style="list-style-type: none"> <li>• Se establecen objetivos cuantitativos (mediciones). El rendimiento de calidad y del proceso se comprende en términos estadísticos. Hay repositorios de métricas.</li> </ul> <p>Básicamente para entrar en este nivel, la organización tiene que realizar un gran esfuerzo en diseñar e implementar métricas que permitan medir sus procesos, y es en base a estas, que dicha organización podrá tomar decisiones.</p> <ul style="list-style-type: none"> <li>• La diferencia entre el nivel 3 y 4 es la predictibilidad del rendimiento de los procesos.</li> </ul>	
Optimizado - Nivel 5	<ul style="list-style-type: none"> <li>• Es una organización en mejora continua. Hay mejoras incrementales e innovaciones de proceso y tecnológicas. Es el nivel de optimización.</li> </ul>	<ul style="list-style-type: none"> <li>• Innovación y desarrollo organizacional.</li> <li>• Resolución y análisis causal.</li> </ul>



## Otros modelos:

### ITIL:

- Conjunto de buenas prácticas utilizadas para la gestión de servicios de tecnologías de información
- Provisión y Soporte de servicios de IT es lo más importante

### ISO 15504 (SPICE):

- Estándar internacional de evaluación y determinación de la capacidad y mejora continua de procesos de ingeniería de SW
- Modela procesos para gestionar, controlar, guiar y monitorear el desarrollo del software

## Software Engineering Approaches:

### Proyecto:

Esfuerzo temporal emprendido para crear un producto o servicio único para lograr un objetivo.

Organizar los equipos en base a proyectos.

Características:

- Tiene un objetivo o beneficio
- Temporal porque empieza y termina
- Único porque no es recurrente, cada proyecto es diferente al otro.
- Posee recursos limitados. (5 dimensiones del paper leído).
- Consta de una sucesión de actividades o fases en las que se coordinan los distintos recursos.

### **Project Management:**

Planificar, organizar, obtener y controlar recursos, utilizando herramientas y técnicas para lograr que el proyecto logre sus objetivos en tiempo y forma.

### **Dimensiones de un proyecto de software:**

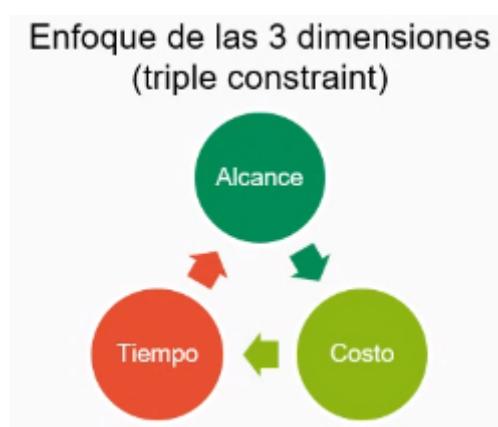
-Aspecto que el proyecto va a tener en el cual yo quiero prestarle particular atención.

-No se pueden cumplir todas a la vez. (relación con paper Good Enough)

-Cada una puede ser:

1. Driver: Objetivo vital a lograr, tiene poco/nada de flexibilidad. Por qué hacemos este proyecto, me cambia el negocio, hace que crezca, etc.
2. Restricción: No está bajo nuestro control directo y también tiene poco/nada de flexibilidad. Ej: Tiempo en timebox, es fijo.  
No tengo control sobre esta y es impuesta
3. Grado de libertad: Libertad para fijar objetivos. Existe flexibilidad para manejar esta variable. Tiene la mayor flexibilidad

Al haber algún cambio rotundo en el proyecto sabemos que dimensión podemos atacar, mejorar, darle más calidad, más control, etc.

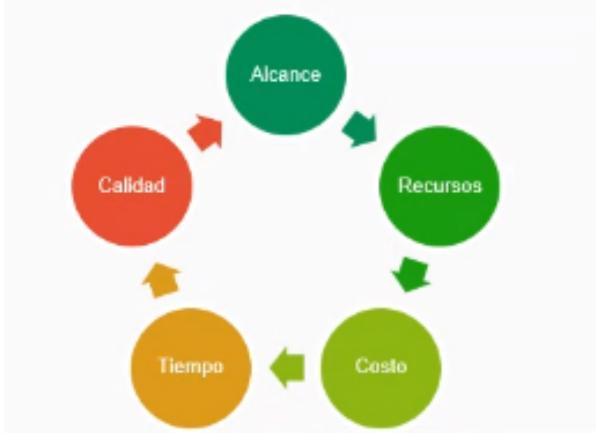


Estas 3 derivan a otra → Calidad dada por como voy a ir variando los 3 puntos.

Referido al paper de GoodEnough

Pero también existe:

## Enfoque de las 5 dimensiones



Al ser cinco ya no es una que no vas a poder manejar como el enfoque anterior sino que existe flexibilidad. Voy a poder ver cual de las dimensiones puedo maximizar o minimizar en un proyecto dado, vamos a poder calendarizar mejor, optimizar mejor los recursos, etc.

Los enfoques en alguno de los 5 o 3 niveles están presentes en los classic mistakes.

### Roles principales de un proyecto:

#### Roles principales de un proyecto

- Stakeholders
  - ✓ Son todos los involucrados por el proyecto
  - ✓ Tienen poder de decisión con capacidad de influir en la marcha del proyecto
- Sponsor
  - ✓ Es el “owner” del proyecto
  - ✓ Es el que tiene la autoridad para llevar adelante el proyecto
- Usuario Campeón
  - ✓ Experto en el dominio del problema del proyecto
  - ✓ Asegurar su capacidad para la función y su disponibilidad (son muy demandados)
- Usuarios Directos
  - ✓ Interactúan directamente con el sistema
- Usuario Indirecto
  - ✓ Hacen uso del sistema, aunque no necesariamente lo operan

Usuario campeón → Product owner en scrum. Entiende cómo trasladar los problemas de negocio al software que se está construyendo. Saben cómo opera el negocio y ayuda a varias áreas. Indican si cumple o no las necesidades.

Sponsor → defiende a capa y espada el proyecto, el que pelea con los inversores. Gestiona el presupuesto, la plata.

Usuario indirecto: Se beneficia o perjudica por lo que el software trae

Ej: El cajero usa un software bancario, es usuario directo, y la persona que solicita algo como por ejemplo un plazo fijo es indirecto.

## CYNEFIN:

Un marco para saber que tipo de framework me conviene dependiendo el contexto/dominio donde estoy. (toma de decisiones).

### Modelo que compara características de 5 dominios de complejidad diferentes:

- **Simple:**

Ya tenemos categorizadas las opciones y sé como resolverlo. Utilizamos las mejores prácticas. Se elige la alternativa que produce los mejores resultados.

Cómo actuar: se establecen los hechos, se categorizan y se responde con una regla o se aplica una mejor práctica, sense-analyze-respond.

Ej: modelo Waterfall y estilo de gestión “Command and Control”.

- **Complicado:**

Necesito un experto que me diga la mejor opción entre varias pero sé cual es mi objetivo final.

Requiere análisis o experiencia, existen múltiples respuestas correctas.

Se aplican buenas prácticas

Cómo actuar: Se evalúan los hechos, se analizan y se aplica una buena práctica (sense-analyze-respond)

Una práctica habitual de este dominio es el mantenimiento de sistemas y soporte técnico.

Ej: Se pueden aplicar metodologías como PMBOK, RUP o Prince2

1 y 2 suponen universo ordenado, donde las relaciones de causa y efecto son perceptibles, las respuestas correctas se pueden determinar en función de los hechos.

No hay incertidumbre, sino distintas opciones para tomar que sabes como van a terminar.

- **Complejo:**

No hay respuestas correctas. Soluciones adaptativas. (examinar resultados y adaptarnos) que no suelen volverse a repetir con igual resultados.

Requiere niveles altos de creatividad, innovación, interacción y comunicación.

Se aplican prácticas emergentes

Cómo actuar: Se exploran soluciones (prueban), se analizan y se responde al problema (probe-sense-respond).

No existe algún experto que pueda apoyar al equipo por que generalmente en este escenario se da la innovación, el conocimiento emerge mientras el proyecto avanza, muchas cosas no se han realizado antes

Ej: Scrum o metodologías ágiles para incrementales cortos, creo funcionalidad y pruebo constantemente.

- **Caótico**

No tengo idea como salir y voy probando por iteraciones hasta salir de ahí.

Cualquier acción es la respuesta apropiada.

Hay que actuar. Novel practice

Cómo actuar: Se actúa para establecer el orden, se analiza lo que tiene algo de estabilidad y se responde para transformar el caos en complejidad (act-sense-respond).

3 y 4 están desordenados, no hay relación inmediatamente aparente entre causa y efecto, y el camino a seguir se determina según los patrones emergentes.

Hay incertidumbre, distintas opciones que no sabés cómo van a salir hasta probar.

- **Desordenado:**

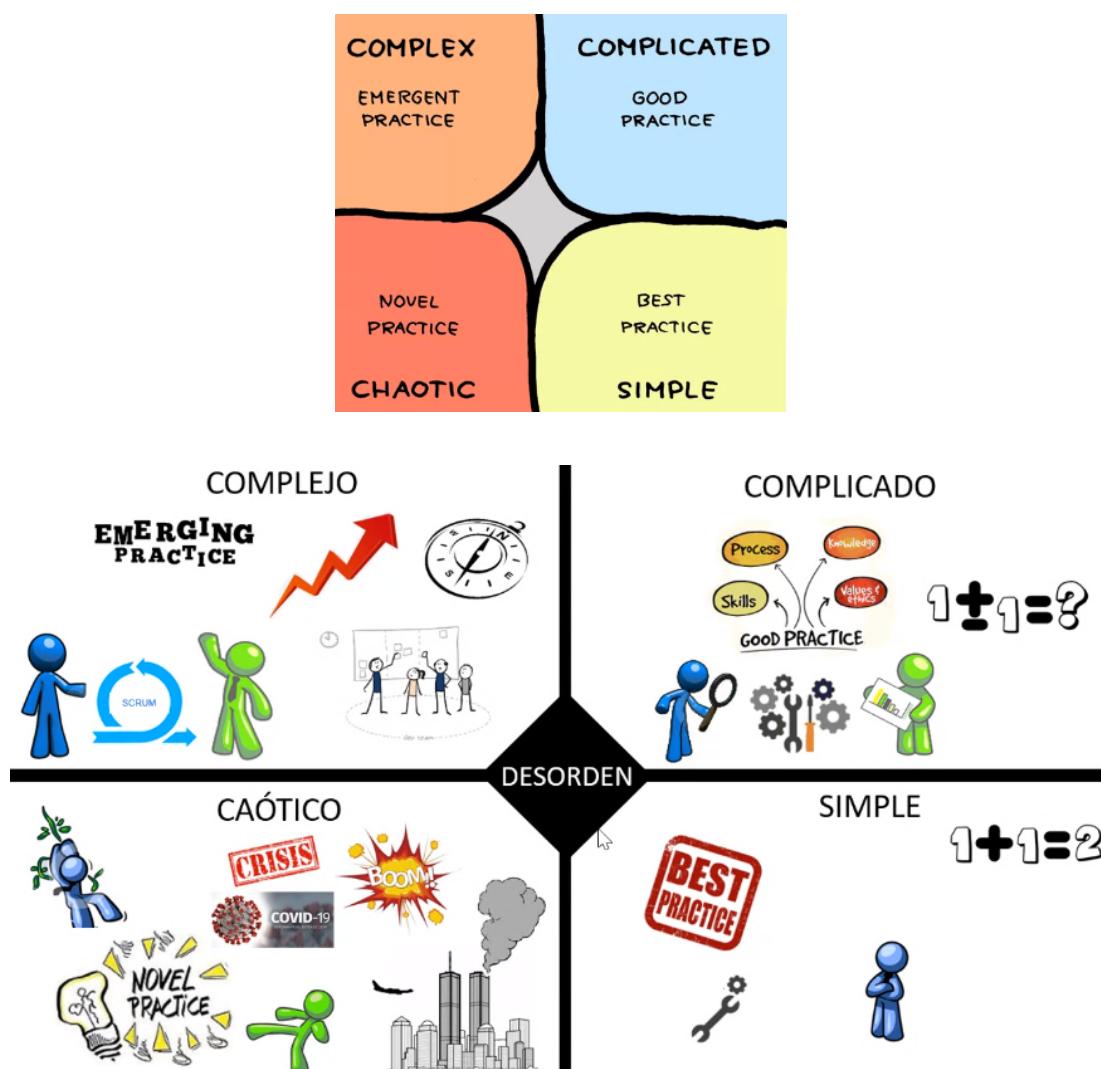
Mundo de la gestión basada en hechos, el mundo desordenado representa la gestión basada en patrones.

No sabemos en qué dominio estamos.

Zona peligrosa: no podemos medir las situaciones ni determinar la forma de actuar

Se suele interpretar la situación y actuar en base a preferencias personales aunque no se debería hacer.

Tenemos que salir sí o sí a una situación en la que podamos identificar cómo actuar.



## SCRUM:

Framework ágil para contextos complejos, donde partis de una visión del negocio donde hay incertidumbre, a través de los sprints construyo ese incremento de producto. Y veo momento a momento con los stakeholders si se adapta a los objetivos.

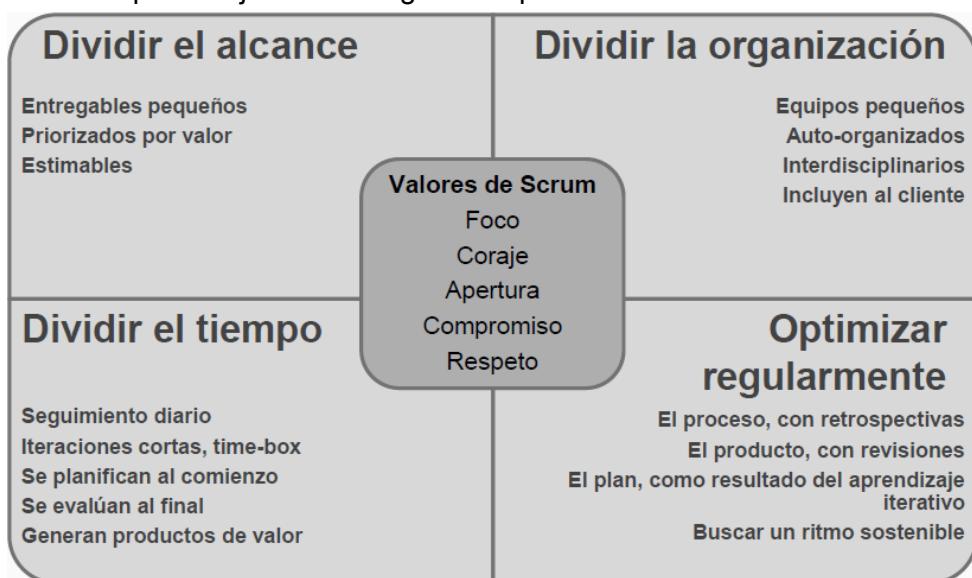
### Tiene las siguientes características:

- Roles - responsabilidades
  - Scrum master: facilitador, vela para que el framework se cumpla. Está entre el negocio y el equipo, facilita las ceremonias.
  - Product owner: Dueño del producto, el que sabe lo que tenemos que hacer, velar para que lo que se hace tenga valor. Define la visión. Tiene que tener mucho poder y conocimiento del negocio.  
Debe optimizar y maximizar el valor del producto.  
Se encarga de la planificación a largo plazo.
  - Development Team: Construyen el producto. El Scrum Master puede ser parte del equipo.  
El equipo determina que entregar en el siguiente sprint y estimar.



- Artefactos: Lo que genero
  - Product backlog: Lista de requerimientos a lo largo del proyecto, y es dinámica.
  - Sprint backlog: Conjunto de requerimientos de un sprint.
  - Incremento de producto: Requerimientos desarrollados completamente y el product owner me dio el okey.
- Ceremonias:

- Sprint: Siempre de tiempo fijo porque querés medir la velocidad del equipo, cuánto pueden construir por sprint, sacar métricas, si tomas duraciones distintas no sirve ninguna métrica.
- Planning: 1 vez x sprint al inicio. Dura 1-2 hs, se compromete el sprint backlog, se asignan user-stories, se generan tareas.
- Daily meeting: El equipo intenta remover los impedimentos. 1 vez x día al comenzar. de 10 a 15 minutos.
- Sprint review: 1hr, 1 vez x sprint al finalizar. Se aprueban las caract. presentadas. Con el cliente. (la demo)
- Retrospectiva: 1 vez x sprint al finalizar. 1-2 hs. Se obtienen las lecciones aprendidas y acciones para mejorar en el siguiente sprint. Puede estar o no el cliente.



## Paper – Scrum in a nutshell

### Definition of Scrum

**Scrum (n):** A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

**Scrum is simple to understand, but difficult to master.**

- The **Product Owner** owns the Product Backlog as he/she has to maximize Return of Investment.
- The Product Owner is responsible for expressing backlog items and prioritizing them.
- The Product Owner discusses and agrees with Stakeholders what to do, why and when to release.
- Collaborates with the Development Team on the details of the Product Backlog during the Sprint.



- The **Development Team** is a cross-functional team with an optimal size between 3 and 9 members.
- The Development Team delivers a Potentially Shippable Increment at the end of each Sprint.
- They are self-organizing. No one (not even the Scrum Master) tells the Development Team how to turn the Product Backlog into Increments of potentially releasable functionality.
- Scrum recognizes no titles for Development Team members other than Developer, regardless of the work being performed by each person.



- The **Scrum Master** is responsible for ensuring Scrum is understood and enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum theory, practices, and rules.
- The Scrum Master is a servant-leader for the Scrum Team, supporting the Product Owner with managing the Product Backlog and coaches the team in self-organization and cross-functional collaboration.
- Helps with removing impediments to the Development Team's progress.
- The Scrum Master leads and coaches the organization in its adoption of Scrum.



- The **Scrum Team** consists of a **Product Owner**, the **Development Team**, and a **Scrum Master**.
- The team model in Scrum is designed to optimize flexibility, creativity, and productivity.

### Scrum Values

**Openness:** Being open drives continuous improvement.

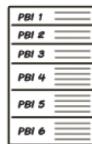
**Focus:** Teams focus to deliver customer value incrementally.

**Commitment:** Teams commit to goals and improvement.

**Respect:** Accept & respect the strengths of different people.

**Courage:** Be brave. Be honest. Embrace failure.

- The heart of Scrum is a **Sprint**, a time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created.
- No changes are made during the Sprint that would endanger the Sprint Goal.
- The Scope may be clarified and re-negotiated between the Product Owner and Development Team.



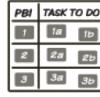
- The **Product Backlog** is an ordered list of Product Backlog Items (PBI's).
- Product Backlog Items are requirements for changes to the product (e.g: user stories...).

- The Product Backlog is ordered by priority, the highest items need to be more detailed.
- The team needs to be able to estimate and test all of the Product Backlog Items.

- The list of items is constantly evolving, changing and updating, the Product Owner is responsible for reflecting the changes in the Product Backlog.

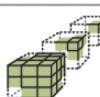


- The **Sprint Backlog** is list of Product Backlog Items selected by the Development Team for delivery in the sprint, plus a plan to deliver the increment.



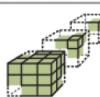
- Only the Development Team can change its Sprint Backlog during a Sprint. This happens throughout the Sprint and in particular in the Daily Scrum when necessary changes are understood.

- The Development Team tracks total work remaining in every Daily Scrum to forecast the likelihood of achieving the Sprint Goal.



- The **Increment** is the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints.

- An Increment must meet the Scrum Team's Definition of "Done".
- It must be in useable condition regardless of whether the Product Owner decides to actually release it.



- Artifact Transparency** is crucial in Scrum. Decisions to optimize value and control risk are made based on the perceived state of the artifacts.

- The Scrum Master must work with the Scrum Team, and other involved parties to make sure the artifacts are transparent and understood.



- The **Daily Scrum** is a daily 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours.



- This is done by inspecting the work since the last Daily Scrum and forecasting the work that could be done before the next one.

- The **Sprint Review** is held at the end of the Sprint to inspect the product increment.



- During the Sprint Review, the Scrum Team and stakeholders review what was done in the Sprint.

- The result of the Sprint Review is a revised Product Backlog that defines the probable Product Backlog items for the next Sprint.

- The **Sprint Retrospective** occurs after the Sprint Review and prior to the next Sprint Planning.



- The purpose is to inspect the process of the last Sprint and identify potential improvements.

- The Scrum Team creates a plan to incorporate these improvements into the way they do their work.

- The Scrum Team regularly does **Product Backlog Refinement** together with the stakeholders to add detail, estimates, and order to items in the Product Backlog.

- The Scrum Team decides how and when refinement is done.

- Product Backlog Refinement usually consumes no more than 10% of the capacity of the Development Team.

### Valores de scrum:

- Focus
- Commitment = Compromiso
- Respect = Respeto
- Courage = Coraje
- Apertura

## Paper – Scrum for Dummies Cheat Sheet

Scrum focuses on continuous improvement, scope flexibility, team input, and delivering quality products. Scrum adheres to the Agile Manifesto and the 12 Agile Principles, which focus on people, communications, the product, and flexibility.

- Product vision statement: An elevator pitch, or a quick summary, to communicate how your product supports the company's or organization's strategies. The vision

statement must articulate the goals for the product. The product vision statement is a common agile practice but is not a scrum artifact.

- Product roadmap: The product roadmap is a high-level view of the product requirements, with a loose time frame for when you will develop those requirements. The product roadmap is also a common agile practice but is not a scrum artifact.
- Product backlog: The full list of what is in the scope for your project, ordered by priority. After you have your first requirement, you have a product backlog.
- Release plan: A high-level timetable for the release of working software. The release plan is a common agile practice, although release planning is inherently part of scrum.
- Sprint backlog: The goal, user stories, and tasks associated with the current sprint.
- Increment: The working product functionality at the end of each sprint.

20/9

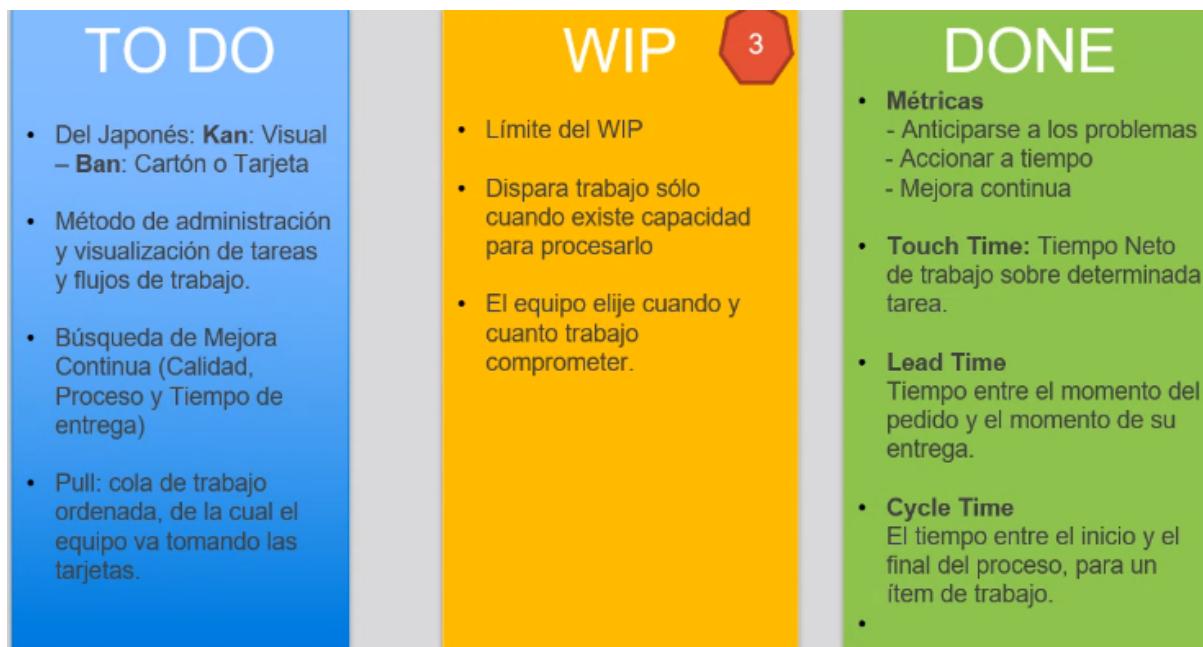
## KANBAN:

-Administración visual de flujo de trabajo que ayuda a realizar más con menos estrés.  
-Disciplina/forma de trabajo donde podés ver visualmente como el flujo de trabajo está funcionando.  
-Ver donde funciona bien, dónde mejorar y donde hay cuello de botellas.

## Filosofía:

Delivery continuo de mejoras al producto sin esperar una interacción. (Flujos continuos sin cortes y que no son fijos como los sprints).

-Mínimo de columnas: ToDo/Doing/Done.  
-La tarjeta es el trabajo que tenemos que hacer.



-WIP - Work in progress: es el límite de tareas que el equipo puede trabajar en paralelo. Podemos ver donde hacer mejoras.

La práctica de limitar WIP es lo que marca la diferencia entre tener una “lista visual de todo” y un sistema kanban.

#### Cuándo usamos kanban:

- Cuando el proyecto no es estable, las tareas van cambiando de prioridad
- los requerimientos van cambiando.
- En entornos de resolución de incidencias. (Mantenimiento - fallas de producción)

#### Tiene los siguientes conceptos básicos:

- Visualizar el flujo de trabajo
- Limitar el trabajo en progreso, es decir, cuánto trabajo podés hacer en forma simultánea
- Realizar mediciones y optimizar el flujo

#### LEAN:

-Filosofía y un enfoque que hace hincapié en la eliminación de residuos o de no valor añadido al trabajo, a través de un enfoque en la mejora continua para agilizar las operaciones.

-Se centra en ofrecer una mayor calidad, reducir el tiempo de ciclo y reducir los costos.

-Se aplica en varias áreas: Recursos humanos, sistemas, ventas, etc.



Ej: demasiadas reuniones

Ej: Diagrama de causa-efecto (5 por qué) es fomentado por lean.

-Modelo de gestión que busca guiar y reforzar los principios LEAN de manera efectiva en toda la organización a través del uso de distintas herramientas y sistemas con sus rutinas de trabajo generando los comportamientos necesarios que conduzcan a los resultados deseados creando así una cultura de mejora continua de creación de valor al cliente y desarrollo de las personas.

#### 4 pilares:

1. Principios guías: Me dirigen en los resultados.
2. Sistemas: Ayudar a lograr resultados y a conducir los principios
3. Herramientas: Idem 2
4. Resultados: Cumplí con mis principios

#### 4 ejes:

Buscamos mejora continua, lograr objetivos, aprendizaje constante, desarrollo y respeto de cada una de las personas, etc.

1. Resolución de problemas
2. Personas
3. Procesos
4. Filosofía





## 14 principios:

Relacionados a los ejes

1. Gestionar basado en una filosofía (propósito) de largo plazo.  
Si no tenés el propósito no llegás. Necesitás misión, visión y propósito.
2. Crear flujos de procesos continuos para evidenciar problemas
3. Usar sistemas de procesos “PULL” para evitar la sobreproducción.
4. Nivelar la carga de trabajo
5. Generar una práctica y cultura de “parar” para resolver problemas y asegurar la calidad
6. Estandarizar, base fundacional de la mejora continua y el empoderamiento de las personas.
7. Utilizar controles visuales para evidenciar los problemas. Ejemplo Kanban.
8. Recurrir únicamente a tecnología confiable y correctamente testeada para servir a las personas y a los procesos.
9. Desarrollar líderes que entiendan el trabajo, viven la filosofía y se la enseñen a otros.
10. Desarrollar personas y equipos excepcionales que sigan la filosofía de su organización.
11. Respetar a tu red de contratistas y proveedores desafiandolos y ayudándolos a mejorar.
12. Ve y observa por ti mismo para comprender profundamente la situación
13. Tomar decisiones por consenso, teniendo en cuenta todas las opciones, e implementarlas ágilmente.
14. Propiciar el proceso de convertirse en una organización que aprende a través de la reflexión y la mejora continua.

## Estimaciones:

Estimamos tamaño y luego determinamos el esfuerzo que nos toma.

Preguntas que se suelen hacer: Cuánto tiempo para desarrollar, cuantas hs/persona para desarrollar, y cuánto va a costar.

Pregunta que deberíamos hacer: Cuál es el tamaño de lo que tenemos que construir?

**Proceso de estimar:**

Requerimientos → tamaño (unidad rigurosa) → esfuerzo (horas/días hombre) → costo → duración.

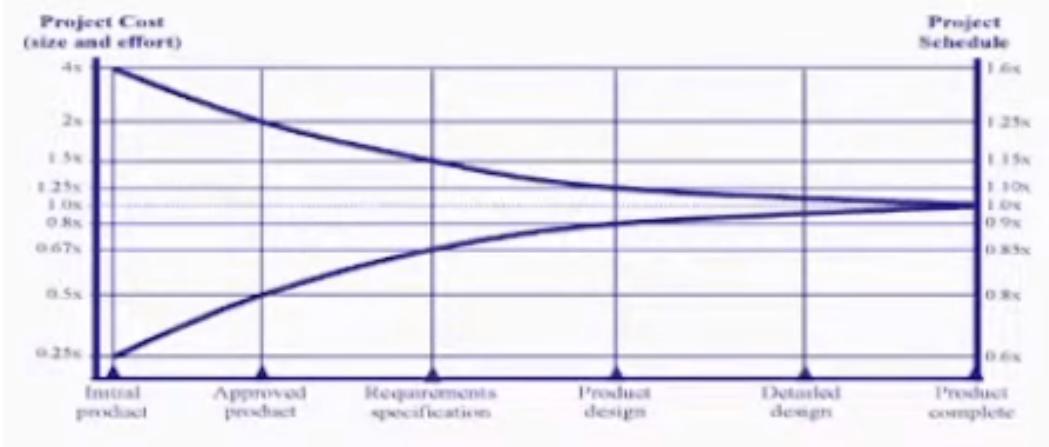
EL PROCESO DE ESTIMAR



**Cono de la incertidumbre:**

-Define niveles estadísticos predecibles de incertidumbre de las estimaciones en cada etapa del proyecto.

Estimaciones tempranas del proyecto suelen ser inexactas porque hay muchos supuestos. En la medida que tengo más información tengo menos incertidumbre.



Tips:

- Diferenciar estimaciones, objetivos y compromisos. Ej: un proyecto tiene tamaño x y requiere x cantidad de personas y compromiso del tipo esto tiene que estar hecho para tal fecha o tengo tal presupuesto.
- Asociar estimaciones con % de confiabilidad.
- Estimar en rangos y no un lugar.
- Presentar los supuestos tomados a la hora de estimar.
- Ley de Parkinson “Toda tarea se expande hasta ocupar el tiempo que tiene asignado”.

- Recolectar datos históricos para tener como referencia.
- Contemplar todas las actividades no solo código y testing.
- No asumir que sólo porque pasa el tiempo o fases del proyecto avanza con menos incertidumbre sino que yo tengo que ir finalizando tareas (cono de incertidumbre)

Estimaciones creíbles:

- Debe ser caja blanca
- Necesitamos historia de proyectos pasados para poder comparar
- Las estimaciones se basan en comparaciones de proyectos o tareas viejas.
- Las estimaciones las hacen las personas no herramientas ni modelos.

## Ciclo de estimación:

1. Estimar: A partir del tamaño para derivar el esfuerzo y el costo.
2. Medir: Mientras evoluciona el proyecto, medir el tamaño, el esfuerzo y costo incurrido.
3. Registrar: Dejar claras las mediciones tomadas, documentar lo que fui midiendo.
4. Analizar: Razones de desvíos, supuestos que quizás variaron, temas no contemplados, etc. Comparo lo que medí y lo que registré, me dio más o menos? Si me desvié por qué y cómo?. Analizo el resultado de la comparación.
5. Calibrar: Ajustar cada una de las variables y parámetros que intervienen en el proceso de estimación.
6. Volver a estimar:
  - a. El mismo proyecto pero ahora con más info que al empezar el mismo.
  - b. Nuevos proyectos con el proceso ajustado por la “calibración”.

Tamaño: Se usan los métodos de estimación (puntos de planning poker por ejemplo).

Esfuerzo: Trabajo en horas hombre que le llevaría a una persona con el conocimiento necesario ejecutar la totalidad de una tarea.

27/9

## Timebox Development:

- Práctica de construir en base al tiempo.
- Mantiene el foco del proyecto en las funcionalidades más importantes.
- Es recomendable aplicar en in-house business
- El éxito de esta práctica radica en aplicarla en los tipos apropiados de proyectos y planificaciones y usuarios finales que permitan cortar/dejar de lado funcionalidades en vez de ajustar la planificación.
- Se suele combinar con JAD, CASE tools, prototipos y evolución en proyectos RAD.
- Se utiliza en la etapa de desarrollo.
- Tiempo como *constraint*, features como *grados de libertad*.

## Riesgos de la práctica:

- Usarlo en proyectos que no encajan.
- Sacrificar calidad en vez de funcionalidades: Si el usuario pide planificación ajustada, alta calidad, muchas funcionalidades, no se pueden cumplir las 3 cosas. (Paper goodEnough).

#### Cómo funciona:

-Se indica un tiempo específico y máximo que se va a dedicar a construir el software. Podés desarrollar lo que quieras y de la forma que quieras siempre que ese desarrollo esté en el tiempo estipulado inicialmente y no se puede cambiar.

-Construcción como cebolla, las funcionalidades esenciales en el centro y las otras en las capas.

-Se desarrollan las funcionalidades más importantes y como permitidos para luego las otras.

-Desarrollar prototipos y envolverlos en un sistema funcionando.

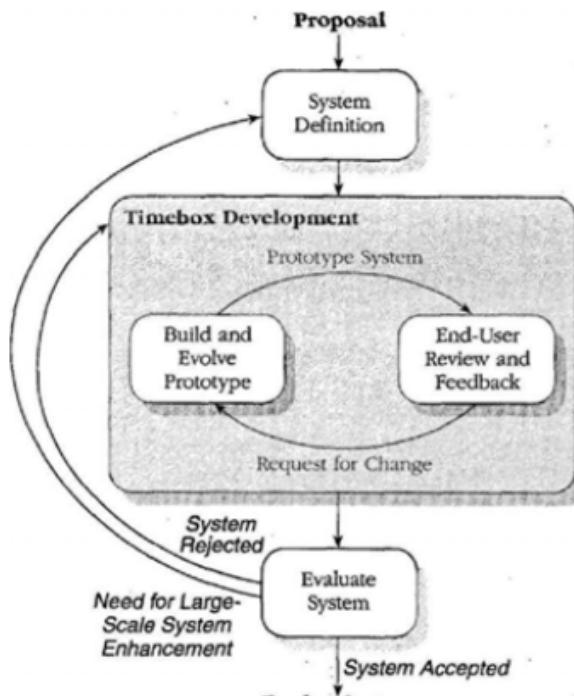
-Tiempo estimado de 60 a 120 días. Períodos más cortos no son suficientes para sistemas significativos. Tiempos más largos no generan ese sentimiento de urgencia para tener el objetivo claro. Si el proyecto es más grande que esos días se puede dividir en varios timebox projects.

-Después de la fase de construcción se puede aceptar y llevar a producción o rechazar para volver a hacer o no.

Time limit = timebox = deadline

#### Características:

1. Prioridad en la planificación: Que quepa en el tiempo estipulado ya que es sobre todo lo más importante. Si el alcance del proyecto tiene conflictos con el tiempo, se reduce el alcance.
2. Evita los problemas 90-90: Cuando el proyecto está 90% completado pero se queda ahí por meses o años
3. Deja en evidencia los funcionalidades principales
4. Limita la perfección del desarrollador: Si hay varias opciones para hacer una funcionalidad ( 2 días, 2 semanas o dos meses), timebox siempre sugiere que se debe elegir la más corta.
5. Controla las funcionalidades raíces.
6. Motiva devs y usuarios finales: A la gente le gusta sentir que hace algo importante, y el sentido de la urgencia lo hace.
7. Hay mucha participación del usuario final: Sponsor, usuarios clave, QA, soporte técnico, auditoría, etc.



### Ver si se adapta a tu proyecto:

1. Prioriza lista de funcionalidades: El framework y las func. deben estar bien definidas. Saber cuáles van si o si y cuales son opcionales. Si no se puede no encaja.
2. Estimar con el timebox team de forma realista
3. Tu proyecto debe tener el tipo correcto: Mejor encaja en in-house softwares, que usan CASE tools u otras que soportan desarrollo extremadamente rápido de código.
4. El usuario final tiene que tener la suficiente participación: Tener buenos feedbacks.

Tipos de equipo de timebox::

- De 1 a 5 devs.
- El equipo total incluye usuarios finales con dedicación full-time al rol del equipo.
- Altamente motivados.
- Tienen que tener mucho conocimiento, no hay tiempo para aprender un nuevo soft.

## Estimaciones:

Costo y duración → Se pueden hacer en paralelo, o la duración antes que el costo.

### Diferencia entre esfuerzo y duración:

Esfuerzo: trabajo en horas hombre que le llevaría a una persona con conocimiento ejecutar la totalidad de una tarea de forma ininterrumpida.

Duración: APPLICANDO REGLAS a ese esfuerzo, feriados, paralelización de tareas, etc etc. es la manera que obtengo el tiempo que se tardará.

## Métodos de estimación para medir tamaños:

Cada una tiene ventajas y desventajas, y las desventajas son respecto a cuánta incertidumbre no estoy contemplando

### 1. Rudimentarios:

- No tienen un proceso ni técnica, no usan historia en su concepción (recopilar, documentar casos anteriores de forma formal no solo experiencia).
- Terminan estimando esfuerzos (horas humanas).
- Casos:

- Juicio Experto:

Basados en experiencia define la cantidad de horas que necesita la tarea. Ej: Esto me tomará 4hs.

Muy fácil de estimar pero la incertidumbre es alta y no es una técnica repetible.

Depende de la persona y el contexto en el que se encuentre esa persona.

Lo hace una “sola” persona.

- Pert

Varios puntos de muestra, con la fórmula de la “media” obtenemos un esfuerzo consensuado.

Sirve cuando la estimación cambia si suceden distintas tareas previas, dependencias.

Tiene 3 puntos de estimación:

1. Estimación optimista: Todas las condiciones buenas pasan
2. Estimación pesimista: Todas las condiciones malas pasan
3. Estimación mejor: Fórmula entre estimación 1 y 2.

- Wideband Delphi

-Con un proceso establecido

-La estimación pasa por distintas etapas en un proceso.

-Se adquiere info del trabajo

-Se arman rondas de estimación con distintos equipos multidisciplinarios para armar la estimación, se elige un esfuerzo indicado y se dividen las tareas. (Kickoff meeting, group estimation, discussion meeting, repeat)

-El viejo planning poker

-basado en el juicio experto en grupos.

-Se puede usar en etapas tempranas del proyecto para proyectos poco conocidos sin historia.

- Planning Poker

Se estiman tamaños de tareas relativos entre sí (pivot), la estimación en Story Points puede variar entre proyectos pero el tiempo no.

Buena para scrum, no tan buena para primeras estimaciones.

### Reglas:

- Todos tienen que estimar

- Fijar cantidad de tiempo para estimar - timebox.
- Tener las historias de usuario priorizadas
- Tener el detalle/relevancia de la historia.
- Tener una escala (las dif tarjetas con puntos, sucesión de fibonacci).
- Usar tarea pivot: La que vamos a comparar contra algo.
- Definir la velocidad del equipo en base a sprints anteriores (para saber hasta cuántos puntos podemos seguir estimando). Ej: Cant. de puntos por sprint
- Definir el terminado: Hasta cuando es el desarrollo, testing, deploy, uat, producción, etc.
- Tamaños/valores de tarjetas: Fibonacci, fibonacci modificado, t-shirt sizes (x, xxs, xxl, etc), múltiplos de 2, etc.

#### Dinámica de la estimación:

- 1-Un moderador cuenta la historia en cuestión.
- 2-Se proponen estimaciones anónimas o todos muestran en paralelo
- 3-El mayor y menor valor justifican el por qué y se vuelve a estimar la historia hasta llegar a un consenso.

## 2. Paramétricos

- Determinar el tamaño de lo que tengo que construir.
- Se busca “contar” algo. Cada método cuenta cosas distintas.
- Hay experiencia e historia almacenada (promedios y sugerencias de distintas organizaciones para por ejemplo determinada cantidad de sprints, con tal método, etc).
- El resultado es una unidad de tamaño, las diferentes unidades de tamaño que hay:

- **Function points:**

Se basa/compara sistemas respecto a funciones independientes de la tecnología, metodología o HW.

Identificar qué funciones tiene que cumplir, que entrada y respuesta generan estas funciones.

Tengo que tener mucho detalle de cómo está construido el SW.

Existen tablas de complejidad que indican los valores/números respecto a las funciones que identifiqué.

Es objetivo, puedo comparar distintos SW respecto a los puntos de las tablas porque las funciones son funciones con entradas y salidas en cualquier proyecto.

-Sin ajustar → dan valor bruto pero indica cómo va a ser implementada la función  
-Primero contás con los puntos de las tablas, después lo ajustas en base factores de ajuste (ajuste de complejidad del software) y después lo traducís a esfuerzos.  
Ej: situaciones externas además de la mera separación de funciones.

-Factores: GSC's que son 14 características generales de los sistemas con una escala de 0-5 de grados de influencia (sin presencia, presencia incidental, moderada, promedio, significante, muy fuerte).

<b>General System Characteristic</b>		<b>Brief Description</b>
1.	Data communications	How many communication facilities are there to aid in the transfer or exchange of information with the application or system?
2.	Distributed data processing	How are distributed data and processing functions handled?
3.	Performance	Did the user require response time or throughput?
4.	Heavily used configuration	How heavily used is the current hardware platform where the application will be executed?
5.	Transaction rate	How frequently are transactions executed daily, weekly, monthly, etc.?
6.	On-Line data entry	What percentage of the information is entered On-Line?
7.	End-user efficiency	Was the application designed for end-user efficiency?
8.	On-Line update	How many ILF's are updated by On-Line transaction?
9.	Complex processing	Does the application have extensive logical or mathematical processing?
10.	Reusability	Was the application developed to meet one or many user's needs?
11.	Installation ease	How difficult is conversion and installation?
12.	Operational ease	How effective and/or automated are start-up, back up, and recovery procedures?
13.	Multiple sites	Was the application specifically designed, developed, and supported to be installed at multiple sites for multiple organizations?
14.	Facilitate change	Was the application specifically designed, developed, and supported to facilitate change?

Desventaja: No se puede hacer desde el día 0, sin desarrollo.

#### Análisis de function points:

-Los sistemas están divididos en 5 clases/características, 1) External Inputs, 2)External outputs, 3)External Inquires (consultas externas que transaccionan con archivos, se denominan transacciones, tratamiento de 1 y 2). 4) Internal Logical Files, 5) External interface files (data stored).

-Necesito gente entrenada y muy experimentada.

Ventajas:

- Podes determinar si un lenguaje, SW/HW es más productivo.
- Podes identificar y medir el scope base.
- Podes usar de métricas.
- Puede usarse por diferentes personas en diferentes momentos y da valores similares.
- Fácil de entender para usuarios no técnicos.

- **Use case points**

-Cómo están definidos los casos de uso (no el diagrama, sino el análisis), cuenta actores, pasos de los casos de uso, cantidad de interacciones entre actores y sistemas, a más interacciones, más pasos o más actores/tipos de actores es más complejo.

-Utiliza tablas similares a function points.

-Sin ajustar → dan valor bruto pero indica cómo va a ser implementado el caso de uso. Se deben ajustar con factores de ajuste con distintos niveles de influencia.

**Table 1. Technical complexity factors**

<b>Factor</b>	<b>Description</b>	<b>Wght</b>
T1	Distributed system	2
T2	Response or throughput performance objectives	2
T3	End-user efficiency	1
T4	Complex internal processin	1
T5	Reusable code	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Includes Security features	1
T12	Provides access for third parties	1
T13	Special user training facilities are required	1

**Table 2. Environmental factors**

<b>Factor</b>	<b>Description</b>	<b>Wght</b>
F1	Familiar with Rational Unified Process	1.5
F2	Application experience	0.5
F3	Object-oriented experience	1
F4	Lead analyst capability	0.5
F5	Motivation	1
F6	Stable requirements	2
F7	Part-time workers	-1
F8	Difficult programming language	-1

-Es más objetivo que un juicio de experto, se recomienda usar para mejorarlo si se combina.

Más características:

- Los actores tienen distintas categorías, el simple (comunicación con API), el promedio (sist. externo con protocolo TCP/IP), el complejo (persona que interactúa con interfaz gráfica). Se asigna un número de peso a cada categoría.
- Los casos de uso tienen las mismas categorías dependiendo la cantidad de transacciones que manejan y también se les asigna un peso.

- **Story points:**

Técnica de modelado donde uno escribe un requerimiento de una manera particular, a través de historias de usuario, y lo que cuenta son cuántas historias están incluidas en el sistema de información.

Se busca una historia pivote con un tiempo promedio, todas las nuevas historias se comparan con la pivote. No se estima con hr/hombre sino con puntos de historia con fibonacci, si toma demasiados puntos debería partirse.

Tener en cuenta cantidad de trabajo, complejidad, riesgo e incertidumbre para estimar.

- **Objects points:**

- Identifico los componentes del proyecto
- Es menos objetivo que function point porque depende de cómo está construido cada SW en específico.
- Los objetos contemplan pantallas, reportes, módulos, y otras implementaciones también contemplan clases, SP's, scripts, etc.
- Los objetos no son necesariamente OOP.
- Se suele usar en la etapa de mantenimiento.

-Se asigna a cada componente un peso de acuerdo a su clasificación de complejidad. Se considera factor de ajuste (precisión) el porcentaje de reuso de código.

-Cada objeto es clasificado como simple - medio - difícil.

Ej: Hacer algo en el service, mapear algo en el mapper, crear el método del repositorio, etc.

- Otros métodos usados antes: Líneas de código pero con un montón de problemas, es un método que varía según lenguaje, compañía, estándar, experiencia, etc.
- Object, Function e Story points establecieron un mecanismo de trabajo y cálculo que garantiza que la mayoría de las funciones son contables y comparables. La diferencia entre ambas es lo que cuentan.

### Papers de métodos de estimación:

[Paper - Fundamentals of Function Point Analysis](#)

[Paper - Comparing Effort Estimates Based on Use Case Points with Expert Estimates](#)

### Preguntas de parcial

¿Qué método de estimación utilizaría su equipo en etapas tempranas de un proyecto (product backlog o lista de requerimientos iniciales para definir si se hacen o no)?

Seleccione una o más de una:

- a. Use Case Point
- b. Function Point
- c. Object Point
- d. Juicio Experto ✓
- e. Planning Poker ✓

Nuestro proyecto está hace más de 3 semanas en etapa de pruebas de usuario. Esto sorprendió a todos ya que la mayoría del desarrollo se hace con herramientas de codificación automatizadas. Cada semana se resuelven algunas fallas, pero se insertan o descubren nuevas. El equipo está continuamente luchando por sobrevivir y no logra superar la situación. En las reuniones que se revisan las fallas detectadas, participa el equipo de desarrollo y se estima el esfuerzo para arreglarlas y el usuario le asigna una prioridad. El negocio, debido a la situación, puso un usuario dedicado a apoyarnos para solucionar esto.

¿Qué herramientas vistas en la materia pueden ayudar a **resolver** esta situación?

Seleccione una o más de una:

- a. Kanban ✓
- b. Scrum - hasta terminar el proyecto
- c. Timebox Development - hasta terminar las pruebas ✓
- d. CMMI
- e. Function Point

En cuál/es de las visiones de calidad, maximizar la calidad minimiza los costos

Seleccione una o más de una:

- a. Visión Basada en Valor
- b. Visión de la Manufactura
- c. Visión Trascendental
- d. Visión del Usuario
- e. Visión del Producto

Your answer is incorrect.

La respuesta correcta es: Visión de la Manufactura

Según qué vision/es de calidad, un Fiat Palio puede tener **más calidad** que un Mercedes Benz.

**NOTA:** un Fiat Palio es un auto barato y básico orientado al segmento de clase media. Un Mercedes Benz es un auto caro y sofisticado orientado al segmento de clase alta.

Seleccione una o más de una:

- a. Visión ISO 25000
- b. Visión de la Manufactura
- c. Visión del Usuario
- d. Visión de Producto
- e. Visión Basada en Valor
- f. Vision CMMI

Your answer is incorrect.

La respuesta correcta es: Visión de la Manufactura, Visión Basada en Valor, Visión del Usuario

¿Qué método de estimación utilizaría su equipo en etapas tempranas de un proyecto (product backlog o lista de requerimientos iniciales para definir si se hacen o no)?

Seleccione una o más de una:

- a. Planning Poker
- b. Juicio Experto
- c. Use Case Point
- d. Object Point
- e. Function Point

---

Your answer is incorrect.

La respuesta correcta es: Planning Poker, Juicio Experto

---

Marque la/s declaración/es incorrecta/s con respecto a las responsabilidades del rol Scrum Master

Seleccione una o más de una:

- a. Facilitador con el equipo de trabajo
- b. Proteje y cuida al equipo
- c. Quita obstáculos del camino
- d. Dueño del Proceso
- e. Está pendiente del cumplimiento del proceso

---

Respuesta incorrecta.

La respuesta correcta es: Dueño del Proceso

---

Cuál/es de las respuestas son apropiadas para Kanban

Seleccione una o más de una:

- a. El bajar el número de WIP me ayuda a que haya menos bloqueos
- b. Si bajo el Lead Time podré aumentar la capacidad de producción
- c. Puedo agregar nuevas tareas a un board de Kanban en cualquier momento
- d. Es el menos adaptativo de los SE Approaches vistos en clase
- e. Se adapta bien a dominios de Cynefin caóticos

---

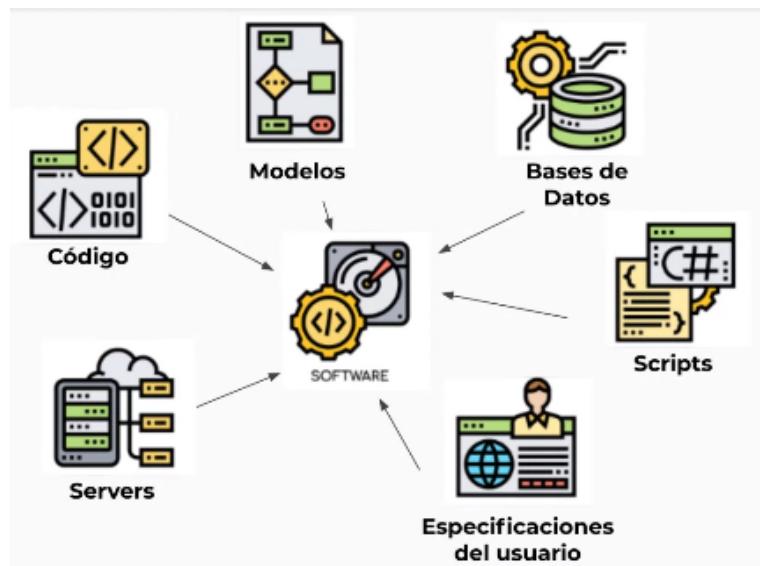
Respuesta incorrecta.

La respuesta correcta es: Puedo agregar nuevas tareas a un board de Kanban en cualquier momento, El bajar el número de WIP me ayuda a que haya menos bloqueos, Si bajo el Lead Time podré aumentar la capacidad de producción, Se adapta bien a dominios de Cynefin caóticos

## Software Configuration Management

Todo software tiene estos elementos/ítems:

- Código
- Documentación
- Scripts
- Bases de datos
- Modelos
- Servers
- Especificaciones del usuario



### Ítem de configuración:

Elementos de cualquier tipo involucrados en la construcción, desarrollo y ejecución de un producto software. Está bajo el control de la gestión de configuración.

Para cada ítem se conoce información sobre su configuración: nombre, versión, fecha de creación, autor, etc.

La gestión de configuración configura ítems de configuración.

### Configuración:

Conjunto de todos los componentes fuentes (ítems de configuración) que son compilados en un ejecutable consistente / software.

Todos los componentes, documentos e información de su estructura que definen una versión determinada del producto a entregar

Ej: Sistemas operativos compuesto por makefile, máquinas virtuales, especificación de entregas, scripts de pruebas y ejecución, etc.

#### Línea Base:

Conjunto de ítems de configuración que forman/generan una configuración a partir de la cual puedo tomar decisiones y puedo establecer un determinado acuerdo en algún momento. Es decir ponerse de acuerdo que esta es la versión que funciona y qué tiene por componentes. Estos componentes me son significativos por algún motivo y me generan una configuración que me sirve de referencia. Ej: configuración inicial.

## SCM - Gestión de la configuración de software:

Disciplina ingenieril orientada a administrar la evolución de:

- Productos
- Procesos: Que me permiten la construcción de ese producto
- Ambientes: Donde el SW va a ser ejecutado

**Propósito:** Establecer y mantener la integridad de los productos del proyecto de software a lo largo del ciclo de vida del mismo.

Trabaja los ítems de configuración y cómo los voy a gestionar.

Hace una serie de tareas, varias, para garantizar que en cualquier momento donde esté parado o haya pasado mucho tiempo, si yo trabajo sobre el software tengo las herramientas para hacerlo.

Involucra para una configuración:

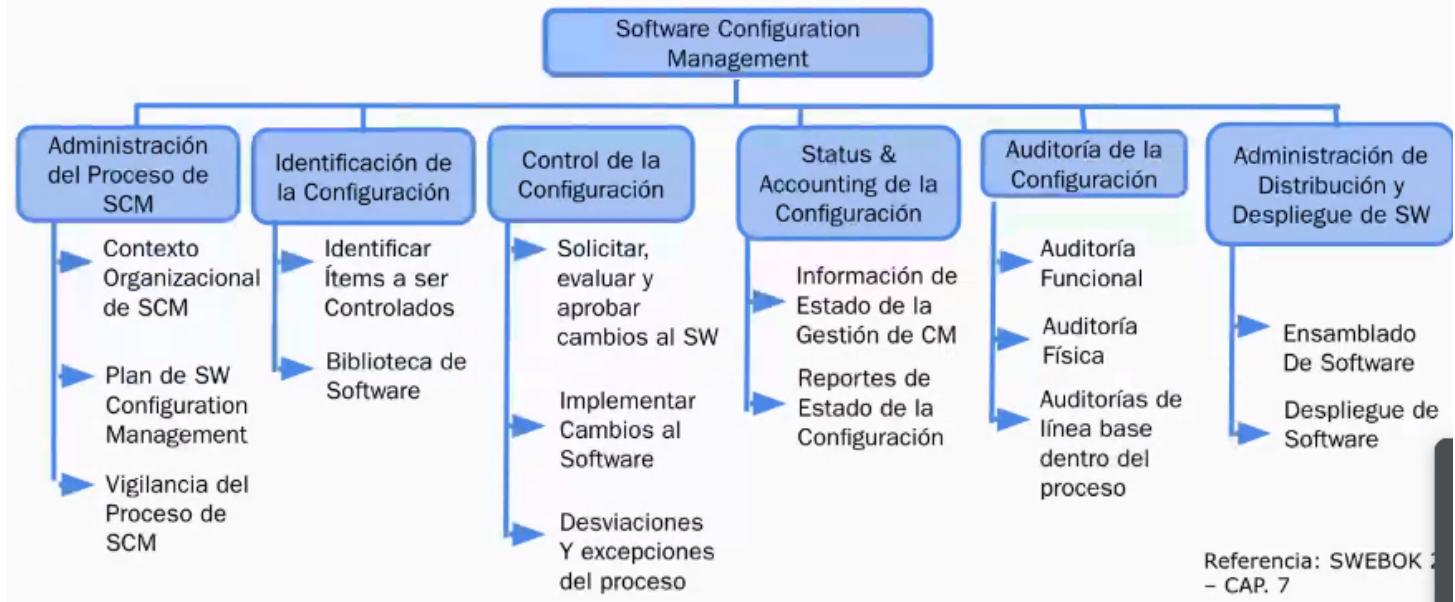
- Identificarla en un momento dado
- Controlar cambios sistemáticamente
- Mantener su integridad y origen.

Acompaña la actividad de cambio con actividades de control.

Tiene su fuente teórica en SWEBOK con un capítulo específico para gestión de la configuración donde define las disciplinas que forman parte de la configuración.

Trabaja las siguientes etapas y tareas:

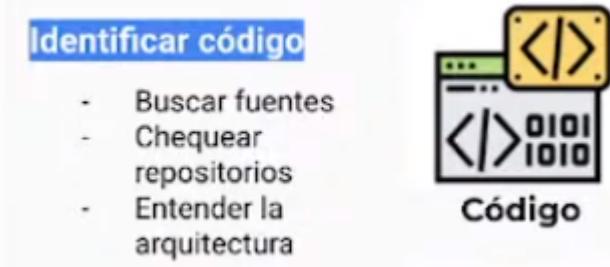
## Software Configuration Management de acuerdo a SWEBOk



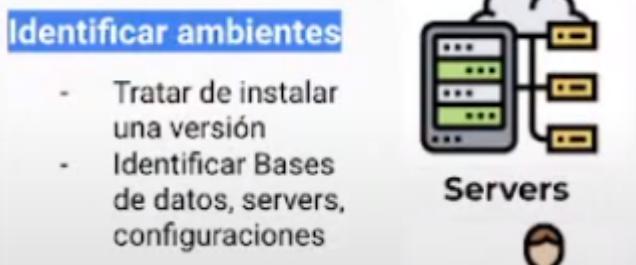
### Caso real de SCM:

Estado caótico, se toma proyecto con todos los devs despedidos.

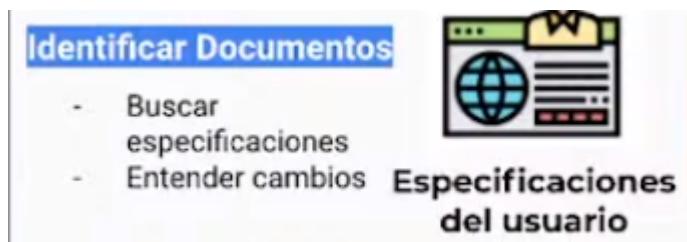
#### 1. Identificar código



#### 2. Identificar ambientes



#### 3. Identificar documentos



Se va a buscar encontrar, de diferentes maneras, la **configuración** del SW que ayude a replicar la instalación de producción. → Primera actividad de SCM. Esto produce el primer paso:

## Pasos/áreas de SWEBOK

Las 4 del medio son las más importantes.

### 1. 1 Identificación de la configuración

Definir qué elementos (ICs) estarán controlados por la gestión de configuración.

- No todo necesita estar bajo SCM: El ítem de configuración tiene que tener un uso, un valor para ser gestionado
- Es necesario contar o definir con guías para qué elementos son parte o no.

Define momentos o condiciones para establecer una línea de base o bien para liberarla (es decir, llevarla a otro ambiente, también conocido como “release”).

Ítem de configuración:

#### Ítem de Configuración

Deberá hacerse seguimiento de atributos importantes de un IC:

- Nombre
- Versión
- Autor / Revisores
- Módulos o ítems relacionados
- Última modificación
- Tipo de IC (código, scripts, diagramas, requerimientos, etc.)

Actividades: Para cada ítem le asigno los datos de arriba, lo identifican únicamente.

#### Clasificación de ítems de configuración:

- **Proceso:** los ítems de proceso se gestionan a lo largo del proyecto o del ciclo/sprint, tienen una vida útil de gestión que equivale a una unidad fija de tiempo. Una vez que termina ya no los gestiona más porque no me interesan. Nacen y mueren con el proyecto/sprint/marco de trabajo que defina.  
Ej: riesgos, plan de calidad, plan de proyecto, plan de configuración, documentos de requerimientos, etc. En sisop la consigna es un ítem de proceso.

- **Producto:** los ítems de configuración de producto definen al producto de software en sí. Trascienden el tiempo del proyecto porque me sirven para seguir trabajando. Viven por fuera del tiempo de proyecto.  
Si hoy queremos recuperar la configuración de un proyecto tendemos a ir a estos ítems de configuración.  
Ej: Requerimientos, planes de instalación, de despliegue, manual de usuario, código fuente, scripts de base de datos, etc.

### Clasificación de ICs de Proceso o de Producto

Proceso	Producto
<ul style="list-style-type: none"><li>- Plan de CM</li><li>- Propuestas de Cambio</li><li>- 10 Riesgos principales</li><li>- Plan de desarrollo</li><li>- Plan de Calidad</li><li>- Lista de Control de entrega</li><li>- Formulario de aceptación</li><li>- Planes de fases</li><li>- Registro del proyecto</li></ul>	<ul style="list-style-type: none"><li>- Requerimientos</li><li>- Plan de Integración</li><li>- Prototipo de Interface</li><li>- Manual de Usuario</li><li>- Arquitectura del Software</li><li>- Estándares de codificación</li><li>- Casos de prueba</li><li>- Código fuente</li><li>- Gráficos, íconos, ...</li><li>- Instructivo de ensamble</li><li>- Programa de instalación</li><li>- Documento de despliegue</li></ul>

¿La minuta de reunión es un ítem de configuración?

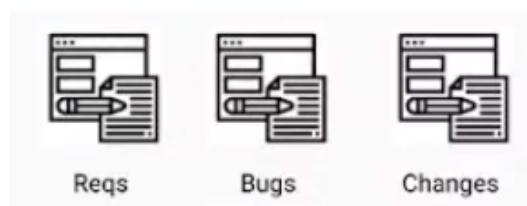
Minuta de reunión: Si es definida por una persona y se envía por “formalidad” o registro no es un ítem de configuración. Pero, si es colaborativa y cada uno va aportando la parte, podría ser un ítem de configuración porque a medida que se cambia hay que ir guardando los cambios, pero depende si es de interés registrar los cambios a lo largo del proyecto.

## 2. 2 Control de la configuración

Ya pasé la etapa de código, tengo todo el código y creo que funciona. Debemos establecer un orden para hacer los cambios en el software.

(Cómo se solicita y quien acepta o rechaza).

Ahora tengo que operar el SW con las situaciones tradicionales que va a tener, cambios por:



Objetivo de esta etapa: Asegurar que los ítems de configuración mantienen su integridad ante los cambios a través de:

- La identificación del propósito del cambio
- La evaluación del impacto y aprobación del cambio
- La planificación de la incorporación del cambio
- El control de la implementación del cambio y su verificación

y en base a eso lo apruebo o rechazo.

Es la etapa MÁS importante porque los cambios pueden romper o dejar deuda técnica.

#### **Change Management Flow (CMF)**

Debo establecer un flujo/forma en la cual a mi me van a solicitar cambios a esta configuración.

Forma común en que se implementa el flujo: Kanban, GERA, TFS, tickets, etc.

#### **SCCB - SW configuration control board:**

Rol/figura o conjunto de ellas definido por el CMF. Comité de control de cambios a la configuración

Personas que tienen poder de aprobar o rechazar cambios a la configuración. Evalúan el impacto del cambio, cómo se modificaría la asignación de personas, etc.

Las personas pueden ser, el líder, el product owner con el líder, el analista funcional y muchos más.

Actividades:

Establecer un procedimiento de control de cambios y controlar el cambio y la liberación de ICs a lo largo del ciclo de vida.

### **3. 3 Status & Accounting de la configuración**

Seguimiento y control de la configuración.

Ahora comienza la ejecución (los devs programan, se opera el SW, PRs, branches, etc).

Se implementa siempre con una herramienta (git) si no es muy complicado.

Registrar y reportar la información necesaria para administrar la configuración de manera efectiva.

- Listar los ICs aprobados
- Mostrar el estado de los cambios que fueron aprobados

- Reportar la trazabilidad de todos los cambios efectuados al baseline.

Se debe informar periódicamente con reportes a todos los grupos afectados.

Debe poder contestar “¿Qué cambios se realizaron al sistema?”

- Cuándo cambió?
- Quién lo cambió?
- Qué cambió?
- Alcance del cambio
- Quién aprobó el cambio?
- Quién solicitó el cambio?

No importa el tiempo que pase desde el desarrollo/implementación, es decir, no importa el distanciamiento de tiempo entre causa y efecto yo conozco quién, cuándo y por qué lo cambió entonces puedo revertirlo o arreglarlo.

Esta etapa busca que sin importar el tiempo que pase entre falla y defecto la herramienta (por ejemplo git) me permita retroceder x pasos necesarios para encontrar el problema.

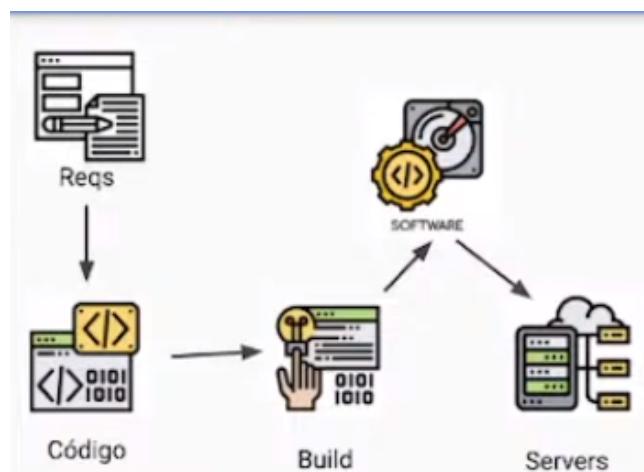
Trazabilidad de cambios: El feature desde la definición en un “papel” hasta que llegó a producción, tengo 100% confianza en el seguimiento de control de cambios.

#### 4. 4 Administración de distribución y despliegue de SW

Tenemos el código, hicimos los cambios pedidos, debemos construir nuestra configuración y habilitarla para que pueda ser utilizada por los usuarios (A.K.A producción).

Con el software que construimos necesitamos desplegarlo en algún ambiente

Desplegar SW: Cualquier software o build que sale del ambiente que fue desarrollado se dice que está siendo desplegado.



- Asegurar la construcción exitosa del paquete de SW, basada en los ICs requeridos para la funcionalidad a entregar, para luego liberarlo en forma controlada a otros entornos ya sea de pruebas, producción, usuario final, etc.
- Se divide en 2 partes:
  - Software building
  - Release Management

Además del ejecutable o paquete de software, comprende la administración, identificación y distribución de un producto.

La funcionalidad de esta etapa la resuelve muy bien Docker, Kubernetes, Jenkins.

Estas 4 etapas son las cores de SCM y luego están estas dos:

## 1- Administración del proceso de SCM:

Es la etapa previa al arranque del proyecto, donde se debe comprender el contexto organizacional. Cómo el equipo va a gestionar todo.

### Algunas preguntas a realizar:

- ¿Existe ya algún proceso de SCM en la organización? ¿Qué establece?
- ¿Qué herramientas hay disponibles? Por ejemplo JIRA, GitLab, GitHub, etc.
- ¿Qué cosas deberemos incluir o cambiar en el plan de SCM?

Actividades que incluye:

Realización del plan de SCM con sus distintas etapas:

- Identificación de la configuración
- Procesos de control de cambios
- Auditorías de software
- Release management, etc.
- Políticas de manejo de ramificaciones o branches para que no crezcan.

Ej: Definir que ante cada sprint que comenzamos, cada tarjeta tiene que tener su propio branch de desarrollo. Cuándo cambio la versión, como vinculo pull request a un cambio, etc.

Puede ser que ese trabajo ya esté definido o tengamos que construir como equipo y consensuar. Se suele hacer una vez sola fuerte y luego se va manteniendo pero se va adaptando y no cambia.

## 2- Auditoría de la configuración:

Objetivo: Realizar una verificación del estado de la configuración a fin de determinar si se están cumpliendo los requerimientos especificados.

Actividad independiente que verifica o valida aspectos de la configuración de software con la que se trabaja.

La auditoría puede ser ejecutada con diferentes niveles de formalidad:

- Revisiones informales basadas en checklists
- Pruebas exhaustivas de la configuración que son planificadas

### Tipos de auditorías:

-Funcional: Verifica el cumplimiento de requerimientos. Validar que lo que se subió a producción cumpla con lo que se indicó que se iba a subir.

Verifica que los requerimientos que se implementaron en una configuración son los que fueron solicitados.

-Física: Verifica la configuración del producto en cuanto a la estructura especificada. Verifica que los cambios al código se condicen con el cambio que yo estaba solicitando. Se busca que no se haya introducido código que haga funciones que no debería hacer. Ej: Tarjetas hardcodeadas.

-De proceso: Verifica que se haya cumplido el proceso de SCM. Ej: si dijimos que para que un cambio llegue a producción pase por un ticket de JIRA, lo tiene que incorporar a un sprint backlog y deployar. La auditoría va a chequear que se pase por todos esos lados.

Las auditorías no forman parte del proceso sino que son un control posterior.

[Paper – But I only changed one line of code.](#)

SCM es una buena práctica de software, pero es usualmente ignorada y produce impactos negativos serios.

SCM es el arte de identificar, organizar y controlar cambios de software. SCM tiene que ser implementado en toda la vida útil y ciclo de vida incluso la mantenibilidad del producto software.

También se puede definir como partes de software bien definidas y los procesos y herramientas exactas para construir el producto desde esos elementos. Incluye el manejo de versiones y releases.

Nombra las 4 áreas funcionales.

La identificación de la configuración es lo más importante de SCM porque los ítems que no identificas no los podés manejar.

**Cambiar una línea de código puede explotar todo:** El problema está en que ni la organización ni los desarrolladores estuvieron controlando las funcionalidades. Hay poco testing unitario y escaso test de integración. Si se usara correctamente SCM esos cambios como el de una línea sola sólo estarían permitidos de “deployar” o implementar luego de analizar, evaluar, revisar y aprobar por uno o más personas autorizadas.

Following is a list of sample reports that the status accounting function should provide.

- Transaction log.
- Change log.
- SCI delta report.
- Resource usage report.
- SCI status report.
- Changes in progress report.
- Change request status report.
- Change completion report by developer, application, etc.

Buenas prácticas: La gente sigue, implementa y fomenta las buenas prácticas si sabe que alguien o algo está chequeando esos resultados o esas implementaciones. Si nunca se miden las métricas obtenidas entonces los desarrolladores también las van a ignorar. Si nadie controla las políticas, procesos y procedimientos, los desarrolladores también las van a ignorar.

La auditoría (área del SWEBOk) debe ser continua y con incrementos de frecuencia y profundidad a lo largo del ciclo de vida.

Ejemplos de problemas si no se implementa bien o directamente no se implementa SCM:

- A software bug that was fixed six months ago has suddenly reappeared.
- A programmer just spent 12 hours making a change to the wrong version of the software.
- There is no way to trace requirements from the requirements document to the user documentation and source code.
- Two programmers working on a project have overwritten each other's code, rendering the last 40 hours of each of their work useless.
- No one can find the latest version of the source code.
- Fielded software that was working fine yesterday mysteriously will not work today.
- An application installed at various locations is running fine at some locations, but not at other locations.
- There is no way to evaluate impact on requirements from proposed changes.

### Conclusión:

Si lo que se definió, analizó e implementó como pasos durante un proyecto no se ejecuta, se saltan pasos o cosas de ese estilo. Hasta una insignificante línea de código puede romper todo en un ambiente productivo porque no se corrigió bien, porque no se validó bien, porque no hubo una segunda opinión, etc, etc.

A supposed one-line change requires proper identification of what actually needs to change impact a change to the code will have another sections of the code. It requires review and approval by individuals who know and understand the application and can determine the extended impact of the requested change. It also requires documentation and traceability into what change is really required, who approved the change, when the change was made, and why it was made. It also assumes that someone is watching the process, that audits are being conducted ensuring that the approved process for making changes is being followed, and that the software product matches the documentation.

25/10

### SCM - Resumen:

1. Disciplina que mantiene la integridad del producto software: Desde que se construye hasta que se sustituye.
2. Tiene definidas actividades
3. Podemos basarnos en el SWEBOk para definir que aplicar en nuestro proceso

Depende del software aproache que definieron en el equipo

### Paso a Paso : Planificar el proceso de SCM

#### Un plan de SCM define principalmente:



Lista ítems de configuración (IC), y en qué momento ingresan al sistema de CM



Define reglas de uso de la herramienta de CM y el rol del administrador de la configuración



Define estándares de nombres, jerarquías de directorios, estándares de versionamiento



Define el contenido de los reportes de auditoría y los momentos en que estas se ejecutan



Define políticas de branching y de merging



Define los procedimientos para crear "builds" y "releases"

Puede ser definido por proyecto o a nivel organizacional (y luego realizar una adaptación al proyecto)

Ej: En SISOP cuando había cambios en el documento del enunciado nosotros no teníamos un BCCN (comité de cambios) que impidiera o ayudara a rechazar esos cambios.

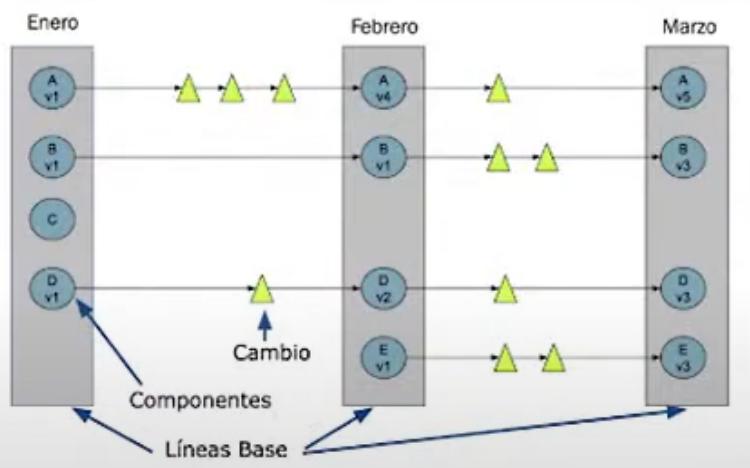
### Definiciones básicas:

- Línea de base (baseline):

#### Línea de Base (Baseline)

Representa un estado de la configuración de un conjunto de ítems en el ciclo de desarrollo, que puede tomarse como punto de referencia para una siguiente etapa del ciclo.

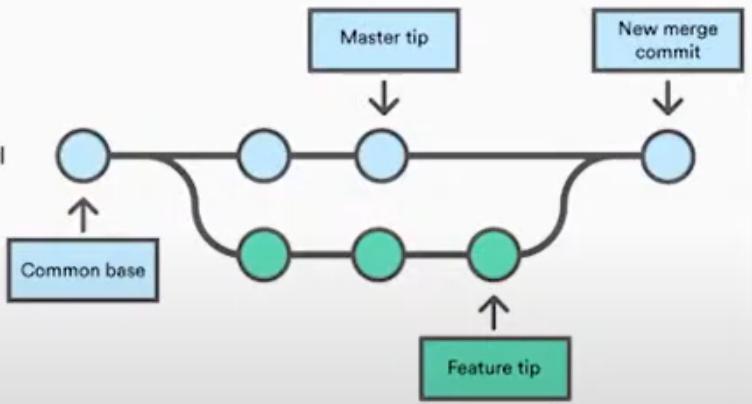
Se establece porque se verifica que esta configuración del ítem o conjunto de ítems satisface (n) algunos requerimientos funcionales o técnicos.



- Rama (branch)

#### Rama (Branch)

- Es la acción de crear líneas de desarrollo separadas.
- Estas líneas utilizan las líneas base de un repositorio existente como punto de partida
- Permite a los miembros del equipo trabajar en múltiples versiones de un producto, utilizando el mismo set de ítems de Configuración



### Trazabilidad entre ítems de configuración:

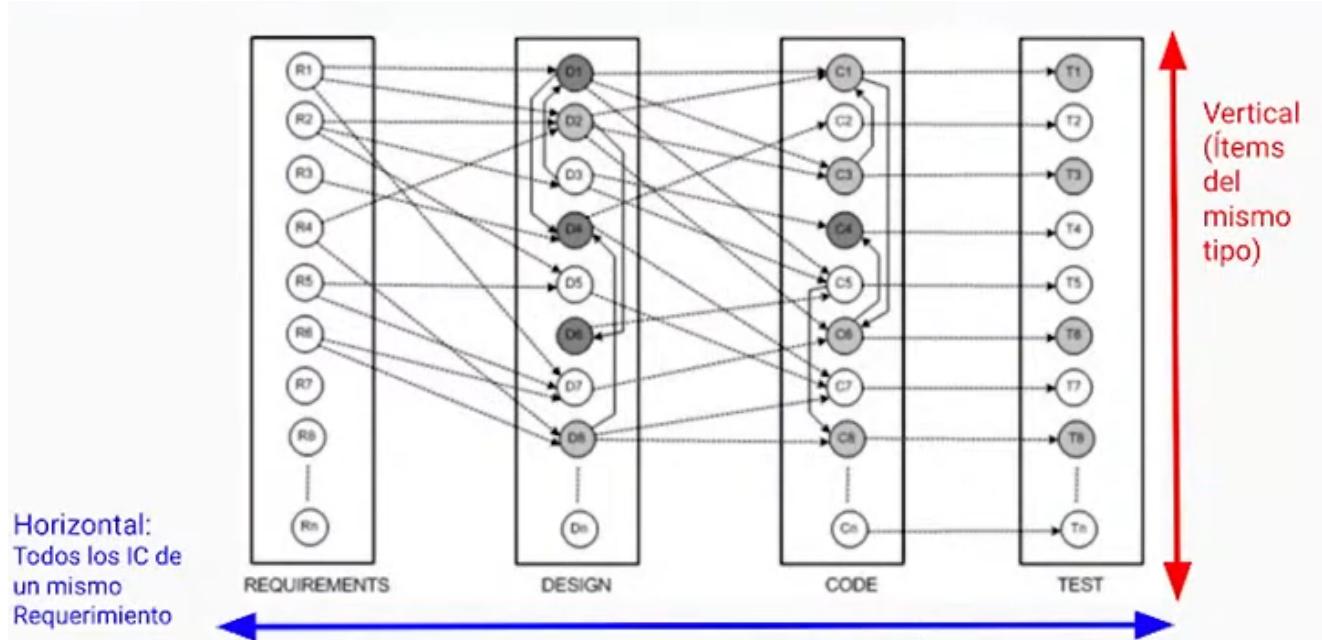
#### Trazabilidad horizontal:

Cuando podemos llevar un ítem de configuración o un documento desde el momento que se originó hasta el momento que está implementado en el producto software.

Puedo rastrear el origen.

### Trazabilidad vertical:

Si todos los ítem de configuración son del mismo tipo, por ejemplo todos los códigos fuentes. En la imagen se muestran todos los ítems de: Testing, todos los de code, todos los de design, etc.



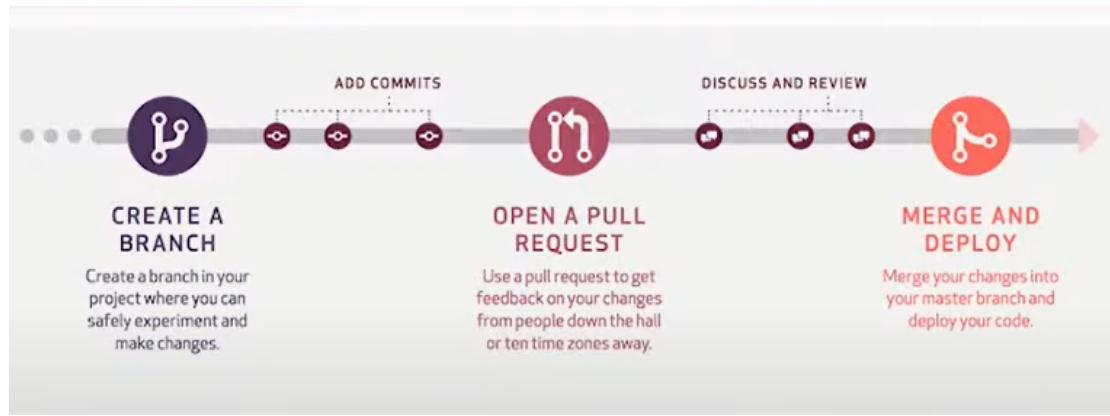
### Proceso de SCM definido

Podemos decir que tenemos un proceso definido cuando (por ejemplo):

- Está definido cómo se registran los cambios de software  
Ej: JIRA, TFS.
- Se establecieron los roles que aprueban o rechazan el cambio.
- Están identificados los repositorios de los artefactos.  
Ej: GitHub (como repositorio de código), npm (como gestor de paquetes), sonatype (como repositorio de paquetes)
- Hay establecida una política de trabajo sobre esos repositorios: El equipo conoce las reglas de desarrollo, quién puede escribir, quién hace y aprueba un review, etc.
- Están identificados los momentos en los que se realizan las líneas base: Si hay cambio va a una nueva rama, cuándo se mergea, etc. Gitflow como flujo de trabajo.
- Los cambios son gestionados completamente, desde el ingreso hasta su puesta en producción: Se identifica cuándo y quién hizo qué paquete.

Disclaimer: Los puntos anteriores no son todos, sino la base mínima necesaria para garantizar la gestión.

### Ejemplo de gestión de cambios en el código:



### Distribución y Despliegue:

Tomo el código, scripts, modelos, documentación, etc de un lugar identificado, una rama trabajada y aplicando reglas y herramientas de construcción genero la construcción del software. Genero un binario por ejemplo.

### Software Building:

- Desde tomar el código de una rama y pasarlo a otra, hasta que genero el binario
- Es un proceso muy repetitivo donde se aplican las mismas reglas
- Garantizo que todos construyamos lo mismo

### Release management:

- Ya construido el binario lo distribuyo o envío a un lugar distinto a donde fue concebido. De forma controlada
- Ya sea para que se pruebe, para una demo, un usuario final, etc.
- Ej: Ambiente de desarrollo, testing, producción y si es móvil google play store o app store.
- Cuándo se habla de release management se suele hablar de las dos partes.

## Release Management: Objetivo

Asegurar la **construcción exitosa** del paquete de software, basada en los ICs requeridos para la funcionalidad a entregar, para luego **liberarlo** en forma controlada a otros entornos ya sea de pruebas, producción, usuario final, etc.

Se divide en 2 partes:

- Software Building
- Release Management

## Release Management

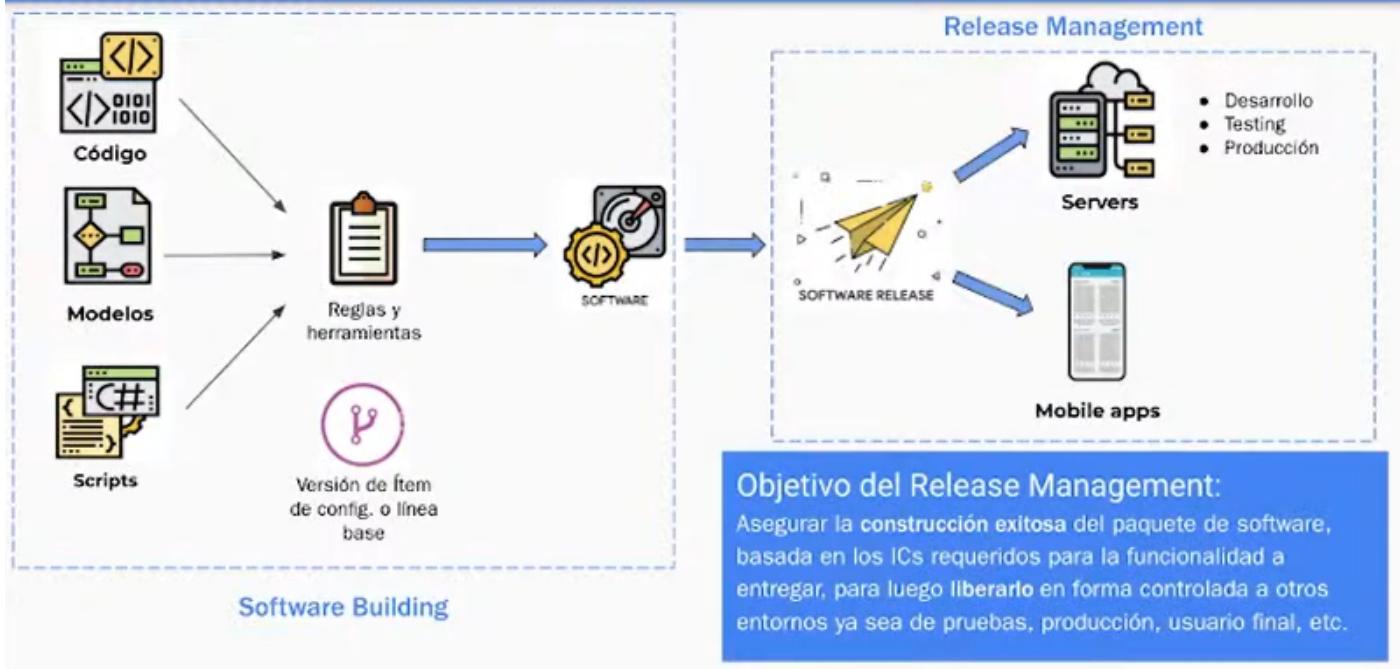
Además del ejecutable o paquete de software, comprende la administración, identificación y distribución de un producto. Esto incluye, por ejemplo, el ejecutable, la documentación, notas de la versión, etc.

Los "Releases" deben incorporar cambios al sistema, producto de errores en el mismo o para incorporar nuevas funcionalidades en el sistema.

Estos cambios deben haber sido aprobados por control de Cambios.

Un release seguramente es una línea base, pero una línea base no necesariamente sea un release. Ya que puedo tener una línea base que no genere un release por ejemplo la inicial

## Distribución y Despliegue (Release Management) del Software



Un release necesariamente requiere una versión (previa o nueva).

**Conceptos de release management:**

- **Versión:** Instancia de un sistema que es funcionalmente distinta en algún aspecto de otras instancias. Dos instancias son distintas en su versión si cambia algo funcionalmente.

**Tipos:**

- Del ítem de configuración: Distintos estadíos por los que ese ítem fue pasando o cambiando. Ej: versión 1 cuando lo cree, 2 cuando hice el cambio, 3 cuando lo pasé a producción.
- Del release o producto software: En dos versiones distintas de release puedo tener la misma versión de un ítem porque no fue modificado.
- Variante: Una instancia de un sistema que es funcionalmente igual a otras instancias pero difiere a niveles no funcionales. Referido a arquitecturas o plataformas diferentes independientemente del ambiente en el que se esté ejecutando.  
Ej: Mismo producto para linux o windows, app que para android te deja cargar sube por NFS y para IOS no.
- Release: Según IEEE refiere a la distribución de SW fuera del entorno de desarrollo.  
Tag == release.

**Tipos de builds:**

Que atacan problemáticas diferentes y tienen objetivos diferentes

1. Local builds:

El developer lo realiza localmente en su entorno de desarrollo, corre pruebas unitarias. En su propia máquina.

2. Integration builds

Objetivo de generar el entorno completo para pruebas de integración. Construyo para poder probar los módulos de software que estoy trabajando.

3. Nightly builds:

Su objetivo es ejecutar la construcción en forma diaria y generar reportes con información sobre estabilidad, tiempo de build, etc

4. Release builds:

Se disparan cuando bien un admin decide crear una nueva versión a ser liberada o por el mismo sistema de integración si se utiliza el modo de deployment continuo.  
De cara al despliegue para el usuario final.

Todos los tipos buscan automatizarse.

**Builds:**

- Deben ser automáticos: Porque son tareas repetitivas y debemos minimizar los problemas que pueden surgir de construir el build
- Deben permitir la generación de reportes: Si algo falló, si algo no funcionó, qué commit rompió, etc.

- Beneficios para el negocio: Reducen la cantidad de defectos, mejora la producción de problemas y trazabilidad y mejora la performance del equipo de desarrollo.

## Deploy:

Se debe evaluar según qué estrategias se usen, qué tecnologías, etc. Dependen del producto software y del lugar.

Se debe evaluar:

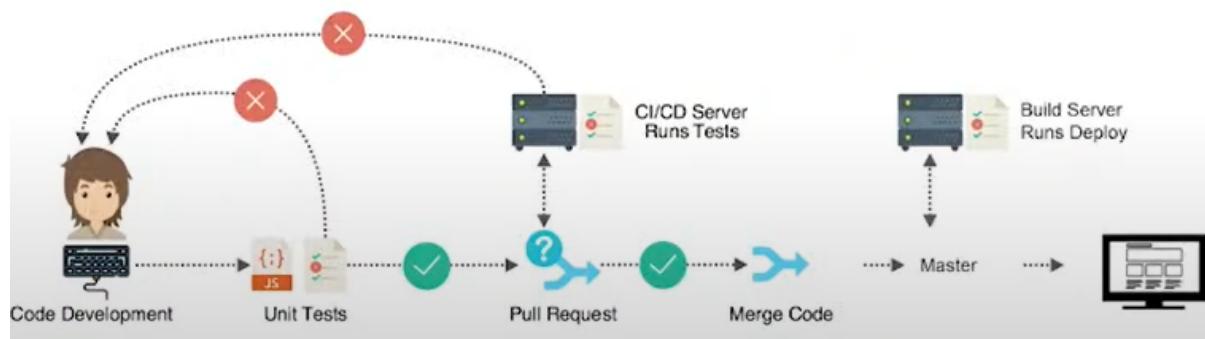
- Qué usuarios deberán recibir los cambios (geográfico, premium, por segmento)
- En qué forma se debe hacer el deploy
  - Entornos: Testing, prod, etc.
  - Procesos de roll forward: Siempre pasando a una versión que incrementa valor. Si podés volver a una versión anterior ante un error o si seguís hacia adelante, desplegando una nueva.
  - Procesos de Rollback
  - Validación de despliegue correcto / incorrecto.
- Riesgos del despliegue y cómo minimizan
- Aprobación por el negocio y/o área de QA
- Quienes realizarán el deployment de la versión definida

## Pipeline:

-Asociado a los deployments.

-Manifestación automatizada del proceso para llevar el software desde la aprobación del cambio de SCM hasta que llega a manos de los usuarios.

-Incluye el camino complejo desde que se compila y construye el SW seguido del progreso a través de varias etapas de testing y deployment. Requiere colaboración entre individuos y equipos.



-Es netamente la parte de construcción del producto software, no incluye lo que sucedió antes con el requerimiento, cómo llegó y demás. Ej: Jira incluye todo.

## Integración continua y despliegue continuo (CI/CD)

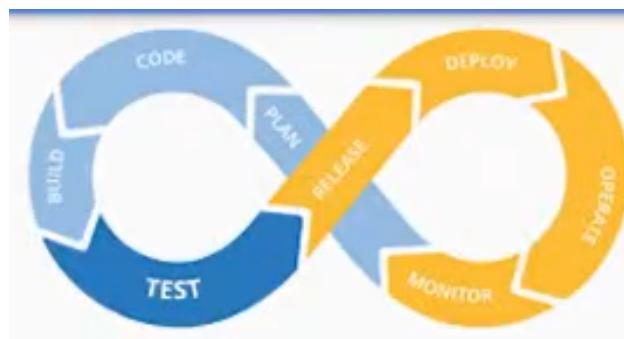
### Integración continua:

Forma de trabajo en la cual se trata de construir, compilar y generar una versión estable del código al menos una vez al día o más para evitar los problemas de pull request grandes y los merges (trabajo en paralelo)

Realizar integración de código al menos 1 o más veces por día, para minimizar problemas de código.

#### Prácticas de la Integración Continua

- Repositorio de código único.
- Automatizar el proceso de build
- Hacer el build testeable automáticamente.
- Todo commit debe construirse por una herramienta de integración - no por el dev
- El build debe ser rápido
- entre otras ....



Necesito scriptear el build, tener un único repositorio, usar siempre la misma herramienta y testearlo. Si cumple las prácticas puedo decir que uso el concepto de integración continua.

### Herramientas de integración continua:

#### Herramientas de CI



Jenkins



Travis CI



SEMAPHORE



Bamboo



GitLab-CI



CODESHIP  
by CloudBees



circleci



TFS

### Responsabilidades del equipo en IC:

- Hacer check-in de código frecuentemente
- No subir código roto
- No subir código no testeado
- No subir código cuando el build no pasa
- No irse a casa hasta que el build central compile.

#### Ventajas del IC:

- ✓ No hay integraciones de días de trabajo
- ✓ Mejora la visibilidad y la comunicación
- ✓ Atrapar errores de integración complejos en forma temprana
- ✓ Mayor rapidez para lanzar software al reducir problemas de integración
- ✓ Fin del "en mi máquina funciona"

#### Desventajas del IC:

- Todas las reglas/configuraciones de deployment tienen que tenerse en cuenta si se modifican. Si modificas la configuración también debés modificar la forma en que se construye.
- Si no se hacen bien los reportes encontrar una falla puede ser muy difícil.

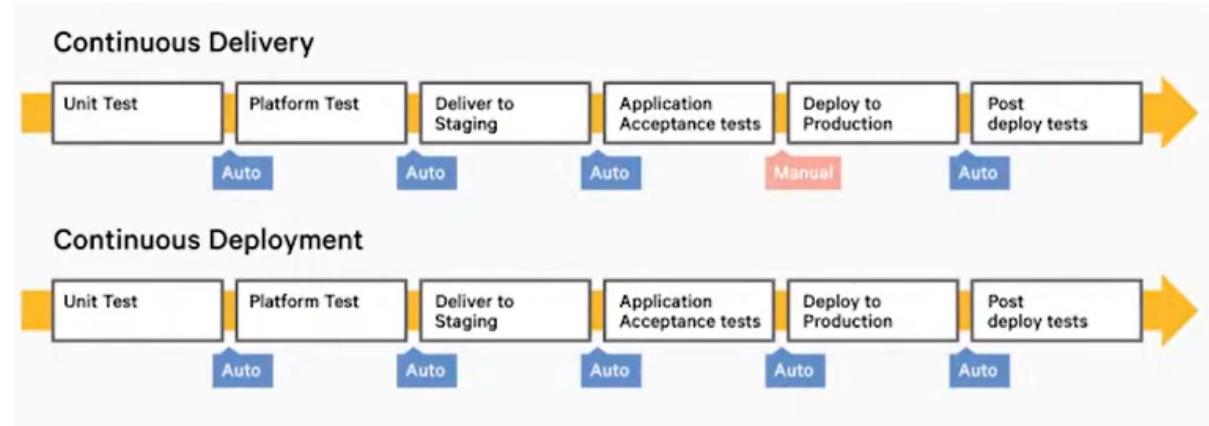
#### Delivery continuo:

Conjunto de prácticas que me garantiza que el SW que tengo construido y automatizado lo puedo desplegar rápidamente en producción cuando quiera.

Trabaja de forma automática hasta un ambiente semi-productivo, y de ahí manualmente. Cada cambio llega a un entorno de staging o semi productivo donde se corren exhaustivas pruebas de sistema. Luego se puede pasar a producción manualmente cuando alguien apruebe el cambio con solo apretar un botón.

Es todo el proceso de automatización salvo la puesta en producción.

#### Continuous Delivery VS continuous Deployment:



C. Delivery: Automatización del despliegue del producto SW que construí hasta el punto anterior al despliegue a producción que se hace manual.

C. Deploy: Todo el proceso hasta el despliegue a producción está automatizado. Paso siguiente a c. delivery. Se complejiza muchísimo más.

## Gestión de configuración - herramientas:



## DevOps

Development and operations

Resuelve la pared que se genera entre los devs y la gente de IT - infraestructura.

Se indican prácticas en común entre el equipo de desarrollo y el equipo de operaciones, desde la planificación del cambio, desde la construcción y codificación, el testing con ambos grupos, se usan herramientas de integración continua o delivery y se tiene todo monitorizado.

### Origen y Definición

Se acuña el término en 2009, en una conferencia que era de infraestructura pero quería atraer desarrolladores. [DevOps days](#) en bélgica.

**DevOps** es un conjunto de prácticas destinadas a reducir el tiempo entre el cambio en un sistema y su pasaje a producción, garantizando la calidad y minimizando el esfuerzo.

Es la combinación de desarrollo y operaciones, normalmente en un equipo

### Prácticas de DevOps

- Planificación del Cambio
  - Conversación y colaboración
- Coding & Building
  - Automated Build Pipelines
  - Infrastructure as Code
- Testing
  - Automated Testing
  - Chaos Testing / Fault Injection
- Release & Deployment
  - Continuous Integration
  - Continuous Delivery
- Operation & Monitoring
  - Virtualization & Containers
  - Monitoring & Alerting tools

### Filosofía CALMS:

Cambio o acrónimo que habla de la cultura del equipo para que trabaje en la unificación y colaboración de ambos sectores. Que haya automatización, los equipos más de infraestructura ayudan a automatizar. Utilizan LEAN para remover desperdicios de esos procesos, y se mide todo lo que se hace para entender el impacto de los cambios que hacemos porque se termina colaborando para que otros puedan hacerlo.

**Cultura:** Ser dueños del cambio para mejorar la colaboración y comunicación.

**Automatización:** Eliminar el trabajo manual y repetitivo lleva a procesos repetibles y sistemas confiables, reduce error humano.

**Lean:** Remover la burocracia para tener ciclos más cortos y menos desperdicio

**Métricas:** Medir todo, usar datos para refinar los ciclos.

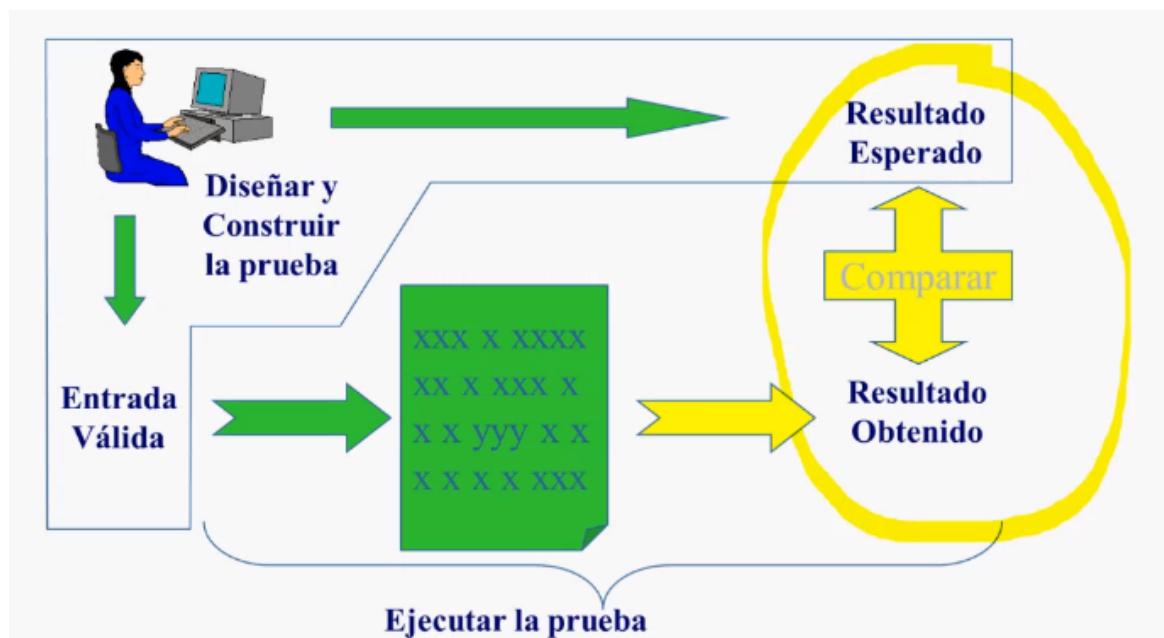
**Sharing:** Compartir experiencias de éxito y falla para que otros puedan aprender.

### Testing de software:

**Es una actividad dinámica:** Necesito ejecutar algo, algo que haya compilado y que pueda correr. No es una revisión de código, ver una porción de código o un script.

**Prueba exitosa:** Si encuentra fallas. Si queremos asegurar que funciona inconscientemente vamos a estar generando cosas para demostrar que funciona lo que tiene que funcionar y no buscar fallas.

Probar es ejecutar un componente con el objetivo de producir fallas.



Si no tengo los requerimientos, no sé los resultados que debo esperar, no sé qué probar, y al final del día los termina pensando el tester.



SW de calidad:

- Aquel que cumpla con los requisitos
- Aquel que al ser usado no tenga fallas.

### Aseguramiento de la calidad - QA:

- Patrón planificado y sistemático de todas las acciones necesarias para brindar una adecuada confianza que un producto o componente cumple con los requerimientos técnicos establecidos.
- Prueba de software es una de las actividades involucradas en el aseguramiento de calidad.
- Implica la revisión y auditoría de los productos y actividades para verificar que cumplen con los procedimientos y estándares aplicables y suministrando a los gerentes de proyecto y otros del resultado de estas revisiones y auditorías.
- Testing es una parte de QA. QA es mucho más que testing. Puede implicar también peer reviewing o auditorías.

### Asegurar la calidad (QA) VS controlar la calidad (QC):

Aseguramiento de la calidad está en cada momento del ciclo de vida independientemente de la forma que tenga tu equipo y tu sistema, busca prevenir defectos, orientado al proceso y refiere a:

- La calidad no se puede injectar al final
- La calidad del producto depende de las tareas realizadas durante el proceso.
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos.

QC entra en un momento del ciclo de vida y si tiene que ver con testing orientado al producto.

Busca encontrar defectos, orientado al producto final

<https://novanotio.es/diferencias-entre-garantia-de-calidad-qa-y-control-de-calidad-qc/> tabla comparativa

## Objetivo del testing:

Encontrar fallas en el producto.

- Hacerlo lo más eficiente posible (más rápido y más barato posible)
- Hacerlo lo más eficazmente posible
  - Encontrar la mayor cantidad de fallas: A más fallas mejor es el testing.
  - No detectar fallas que no son: Menor cantidad de reportes de falso positivo. Al encontrar errores que no son, eso es un “error” no hiciste bien el testing, no fuiste eficaz.
  - Encontrar las más importantes.

Reportar un falso positivo:

- Todo aquello que yo reporto para que sea corregido pero que alguien lo descarta porque funcionaba bien.
- Es contraproducente para la eficacia y para la eficiencia porque te hace perder tiempo y por ende dinero entonces no es más barato.

## Según IEEE:

- Una actividad en la cual un sistema o componente es **ejecutado** bajo condiciones específicas, los resultados de dicha ejecución son observados o registrados y, a partir de los mismos, se realiza una evaluación de algún aspecto del sistema o componente

**Deducimos que se trata de una actividad/proceso *dinámico***

- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

## Conceptos relacionados:

### Falla:

- Resultado de ejecución incorrecto. Es el producido por el SW distinto al resultado esperado. Por la ejecución dinámica.
- Resultado esperado != resultado obtenido.
- Ej: Que tire una excepción, que vuele por los aires, que calcule mal un valor sin explotar.

### Defecto:

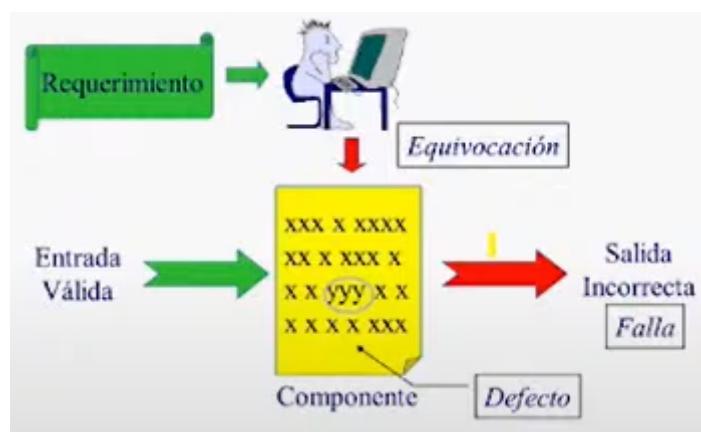
- Paso, proceso o definición de dato incorrecto.
- Ausencia de cierta característica.
- Es el origen de la falla. Qué me generó que el resultado esperado sea distinto al obtenido.
- Ej: un < y > invertido que produce un resultado inverso.

### Equivocación:

- Acción humana que produce un resultado incorrecto.
- En la equivocación se cierra el ciclo de mejora continua.
- Ej: El programador no se dio cuenta o pensó que el signo iba invertido y por eso codeó mal.

### Aclaraciones:

- Una equivocación lleva a uno o más defectos que están presentes en el código
- Un defecto lleva a cero, una o más fallas: Puede explotar muchísimo tiempo después o nunca explotar porque no se pasa por esa sección de código.  
Puede haber un defecto en una función que se use en muchas features entonces ese defecto produce “muchas” fallas.
- La falla es la manifestación del defecto.
- Una falla tiene que ver con uno o más defectos: A veces varios defectos confluyen en una falla.



### Incidente de testing:

Un issue no es una falla, la falla es cuando se ratifica como tal, antes de eso podría ser un falso positivo. Es la ocurrencia de un evento que requiere investigación

#### Según IEEE:

- Toda ocurrencia de un evento que sucede durante la ejecución de una prueba de software que requiere investigación

No toda incidencia es una falla

#### EJEMPLOS:

- Defectos en los casos
- Equivocaciones al ejecutar las pruebas
- Interpretaciones erróneas
- Dudas

### **Condiciones de prueba:**

Descripciones de situaciones que quieren probarse ante las que el sistema debe responder.

Crear condiciones es un proceso creativo, porque desafía a la regla de negocio para ver si se cumple.

Es la situación a probar.

Es la mejor manera para cotejar la completitud, correctitud, consistencia y la no ambigüedad de los requerimientos. Pensar en la debilidad de los requerimientos pensando cómo lo vamos a probar.

Ej: Mayor o igual y menor a 10.000 para un contexto de una feature donde no cobrás un envío si la compra es mayor a 10.000. Son 2 condiciones.

### **Casos de prueba:**

Son lotes de datos necesarios para que se dé una determinada condición de prueba.

Crear casos es un proceso “laborioso” .

Los datos instancian a la condición.

Ej: [5400, 10000, 12000]. Me son suficientes 2 casos para probar esas 2 condiciones.

### **Criterio de Selección:**

Condición para seleccionar un conjunto de casos de prueba

De todas las combinaciones posibles solo seleccionaremos algunas, la menor cantidad de aquellas que tengan mayor probabilidad de encontrar un defecto no encontrado por otra prueba.

Tengo que buscar la menor cantidad de esas agrupaciones (particiones) que a mi me representen todas las condiciones, para probar la totalidad de la funcionalidad.

### **Partición:**

Todos los posibles casos de prueba los dividimos en clases

Todos los casos de una clase son equivalentes entre sí: Detectan los mismos defectos.

Con solo ejemplos de cada clase cubrimos todas las pruebas.

El éxito está en la selección de la partición.

Clases de equivalencia, agrupar valores para casos posibles y no probar 5000 veces algo que con 1 o 2 casos es suficiente.

Asociado a tener el mínimo conjunto de casos que me representen la totalidad de la funcionalidad.

Ej: clase menores a 10.000 todos los números menores, con que pruebes 1 de ellos es suficiente.

### **Depuración:**

Depurar es eliminar un defecto que posee el SW

La depuración NO es una tarea de prueba aunque es consecuencia de ella

La prueba detecta la falla-manifestación (Efecto) de un defecto (causa)

La prueba me detecta la falla pero no el origen de ella, el defecto lo debo buscar en la depuración.

La depuración puede ser fuente de introducción de nuevos defectos: Rompemos más

### **Fases de la depuración:**

En la depuración debemos

- **DETECTOR**
  - Dada la falla debemos hallar el defecto (dado el efecto debemos encontrar la causa)
- **DEPURAR**
  - Encontrado el defecto debemos eliminarlo
  - Debemos encontrar la razón del defecto
  - Debemos encontrar una solución
  - Debemos aplicarla
- **VOLVER A PROBAR**
  - Asegurar que sacamos el defecto
  - Asegurar que no hemos introducido otros (regresión)
- **APRENDER PARA EL FUTURO**
  - Lecciones Aprendidas

### **¿Hasta dónde pruebo?:**

Hasta que el costo beneficio no cierre. Si es más costoso probarlo a que suceda la falla ya no sirve.

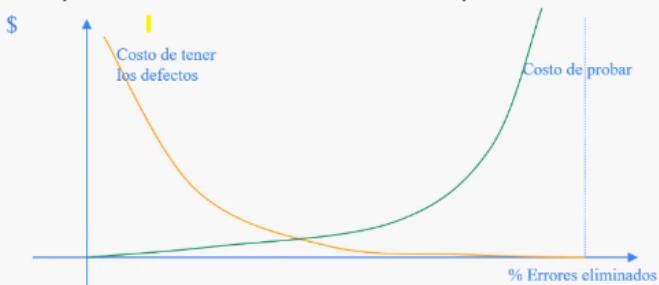
## Economía del Testing

### ¿Hasta cuándo tengo que probar?

Se puede invertir mucho esfuerzo y tiempo en probar

PROBAR en definitiva es el proceso de establecer confianza en que un SW hace lo que se supone que tiene que hacer

Y ya que nunca se va a poder demostrar que un SW es correcto, continuar probando es una decisión económica



### ¿Cuándo detengo la prueba?:

Depende de cada situación en particular. Negocio, criticidad, riesgos, etc.

Algunos criterios:

- Pasa exitosamente el conjunto de pruebas que fue diseñado: Cree condiciones y lotes de prueba y si pasan consideran que está okey.
- "Good enough" cierta cantidad de fallas no críticas es aceptable: Errores conocidos pero se aprovecha el tiempo de salir a producción en ese momento.
- La cantidad de fallas detectadas es similar a la cantidad de fallas estimadas.

### ¿Cómo abaratar el testing?

Armados de ambientes fáciles, diseño modular, pruebas unitarias hechas, etc.

### ¿Cómo abarato la prueba?

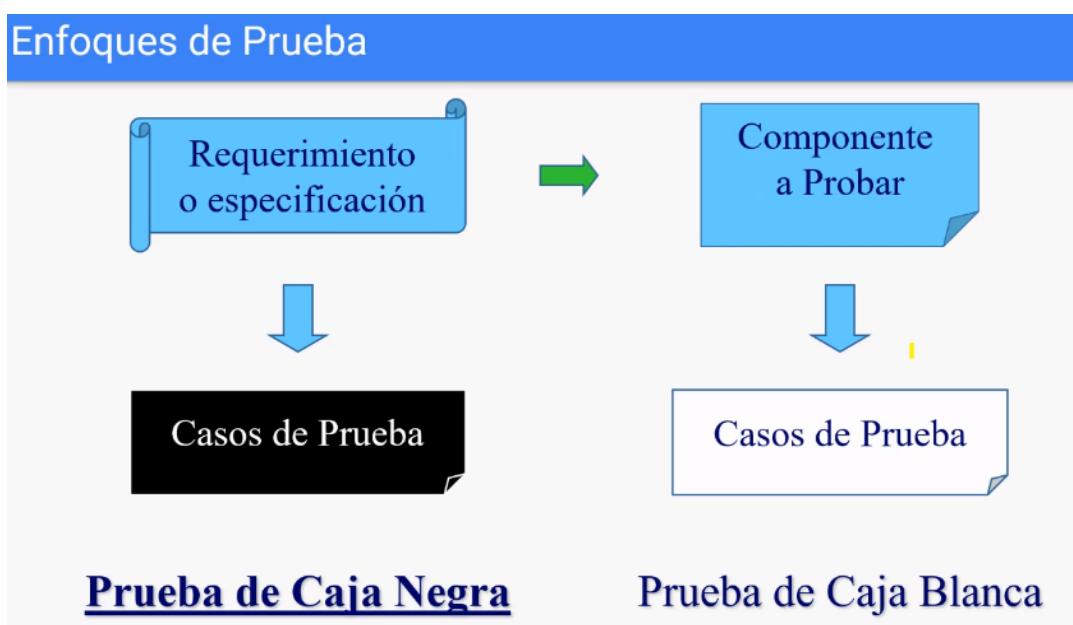
- Hacer pruebas es caro y trabajoso
- La forma de abaratarlas y acelerarlas –sin degradar su utilidad- es:
  - DISEÑANDO EL SW PARA SER TESTEADO
- Algunas herramientas son:
  - Diseño Modular
  - Ocultamiento de Información
  - Uso de Puntos de Control
  - Programación NO egoísta

Ocultamiento de información: El que codea es críptico, no brinda detalles de la implementación.

### Conclusiones:

- No mejoran el SW las pruebas, solo muestran fallas por varios defectos.
- Buen diseño y construcción benefician a las pruebas y la corrección de los componentes y su mantenimiento.
- El no probar no elimina errores, ni acorta tiempos ni abarata el proyecto.
- Lo más barato para encontrar y eliminar defectos es NO introducirlos.
- Probar SW es una actividad CREATIVA e INTELECTUALMENTE desafiante.

## Prueba funcional



### Prueba caja negra:

Prueba funcional, producida por los datos o producida por la entrada/salida.

Partimos del requerimiento y no nos importa el componente, la implementación

Debería conocer la entrada y el comportamiento esperado de lo que ocurre adentro, no sé cómo lo hace pero sé que me tiene que devolver.

Prueba lo que el software debería hacer:

- Se basa en la definición del módulo a probar (definición necesaria para construir el módulo).
- Nos desentendemos completamente del comportamiento y estructura interna del componente.

La prueba de caja negra exhaustiva es imposible de realizar. (tendría que probar todos los valores posibles de todos los datos de entrada a veces son infinitos). Ej: probar TODA la lista de votantes y que me de bien la mesa donde deben votar.

**Técnicas para probar algo que tenemos que reducir** (porque no puedo probar todos los casos):

- Seleccionamos subconjuntos de los datos de entrada posibles, esperando que cubran un conjunto extenso de otros casos de pruebas posibles.
- Podemos suponer que la prueba de un valor representativo de cada clase es equivalente a la prueba de cualquier otro valor cada subconjunto es una clase de equivalencia. Si pruebo para cualquier valor < a 10.000 es el mismo comportamiento, solo pruebo 1 y es suficiente.
- Estas pruebas son llamadas “pruebas por partición de equivalencia” o “pruebas basadas en subdominios”.

**Algunos criterios:**

Que utilizo a la hora de construir estas condiciones de prueba que me agrupan los casos que son infinitos.

El más usado es la partición en clase de equivalencia.

- Variaciones de Eventos
- Clase de Equivalencia
  - *De entrada*
  - *De salida*
- Condiciones de Borde
- Ingreso de valores de otro tipo
- Integridad del Modelo de datos
  - *De dominio*
  - *De entidad*
  - *De relación*

**1 Partición en clase de equivalencia:**

La más usada.

Incluye dos pasos:

- Identificar las clases de equivalencia
- Definir casos de prueba: Poblarla

Se hace dividiendo la condición de entrada en dos grupos independientemente del tipo:

- Clase válida
- Clase inválida

### Por c/condición de entrada

- *Rango de valores. Ej.: 100 < Nro. Sucursal < 200*
  - Una válida y dos inválidas
- *Conjunto de valores. Ej.: DNI, CI, PAS*
  - Una válida y una inválida
- *"Debe ser". Ej.: Primera letra = "A"*
  - Una válida y una inválida
- *Si creemos que los elementos de una clase de equivalencia no son tratados en forma idéntica, debemos dividir la clase en clases menores*
  - Ej.: Las suc. de Cap. Fed. son de la 100 a la 130

#### Por rango de valores:

Hay una clase válida que es la que esté dentro de ese rango y clases inválidas las que estén fuera del rango.

#### Conjunto de valores:

Es válido si pertenece a ese conjunto. Es inválido si no pertenece.

#### 1. 2 Casos de borde:

No son condiciones sino casos.

La experiencia demuestra que los casos de prueba que exploran las condiciones de borde producen mejor resultado que aquellas que no lo hacen.

Los casos de borde NO generan nuevas condiciones a las que identifiqué previamente. Sino que exploran a esas condiciones ya dadas.

Ej: caso del delivery gratis a partir de 10.000. Pruebo con 5.000, con 12.000 y casos de borde como 9999 y 10001. El 9999 es tan chico como el 5000. Pruebo con más casos pero no pruebo con la totalidad, me deja más tranquilo sin dejar de ser eficiente.

El código postal es un número entre 1000 y 8000	<table border="1"><tr><td>998</td><td>7998</td></tr><tr><td>999</td><td>7999</td></tr><tr><td>1000</td><td>8000</td></tr><tr><td>1001</td><td>8001</td></tr><tr><td>1002</td><td>8002</td></tr></table>	998	7998	999	7999	1000	8000	1001	8001	1002	8002
998	7998										
999	7999										
1000	8000										
1001	8001										
1002	8002										

## CONDICIONES DE BORDE:

¿Cómo hacemos?

- *Rango de Valores*
  - Casos válidos para los extremos del rango y casos inválidos para los valores siguientes a los extremos
- *Aplicar lo mismo para los datos de salida*
- *Si la entrada o salida es un conjunto ordenado, enfocar la atención en el primero y en el último de los elementos del conjunto*
  - Prestar especial atención a los archivos/tablas vacíos, primer registro / fila, último registro / fila, fin del archivo / tabla.

## 2. 3 Valores de otro tipo - clases inválidas:

Clases inválidas pero de otro tipo que la clase válida.

En algunos casos ya son resueltas por el entorno de desarrollo (elementos que no te permitan llenar más del límite, o radio buttons, etc). En ese caso los casos de prueba no son necesarios.

- *Números en vez de alfabéticos*
- *Alfabéticos en vez de numéricos*
- *Combinaciones de ambos*
- *Fechas erróneas*

La combinación de datos de entrada puede producir una clase válida o inválida. Ej: divorciado con datos de cónyuge, cuit sin documento.

## Paper - Testing without a map

Afirmación falsa decir que para testear bien tenés que tener una especificación completa y escrita de lo que se debe testear. Y que sin definición no se puede testear. No siempre, las justificaciones:

- Hay muchos casos que te piden testear sin darte una especificación formal.
- “Completamente” depende de la perspectiva y el contexto.
- Mientras más detallado, largo y explícito sea menos lo van a leer entero.
- Estas especificaciones te las pueden dar por mail, call, conversación, por inferencias propias del tester, etc.
- La capacidad del tester y una reunión con el líder a veces son suficientes.

## Testers exploradores:

Tipos de testers donde el conocimiento se da por:

- **Oracles:** Principio o mecánica por el que sabes que algo anda acorde según el criterio de alguien, es decir, nos dice si está bien hecho respecto al punto de vista de alguien.
- **Heuristics:** Es una guía falible y provisional para investigar y solventar un problema. Encontrar respuestas a través de la exploración.

Un producto debe ser consistente en:

1. **Historia:** La funcionalidad o función debe ser consistente con el comportamiento anterior en el caso que no sea necesario cambiarlo.
2. **Imagen:** La apariencia y comportamiento debe ser consistente con lo que la organización quiere proyectar a los clientes o a los usuarios internos.
3. **Comparación con productos:** Deberíamos poder usar otros productos como guías.
4. **Afirmaciones:** Debe ser consistente con lo que digan los documentos, artefactos o personas a través de especificaciones, un archivo de ayuda, advertencias, un mail, etc. y esas afirmaciones las deben reclamar o pedir alguien con autoridad
5. **Expectativas de usuarios:** Debe ser consistente con lo que los usuarios entienden que debe hacer o lo que esperan.
6. **El producto en sí:** Consistente respecto a lo que debería hacer o patrones funcionales de estos tipos de productos a menos que haya una buena razón para que no sea igual.
7. **Propósito:** Debe ser consistente con lo que aparenta ser de propósito.
8. **Estatutos:** Relación con lo legal o regulatorio.

### Conclusión:

En contextos en los que no tenés definido algo como lo esperarías, para no cruzarte de brazos lo que podés hacer como tester es todo lo que nombra el paper. Además tenés tu experiencia de donde agarrar.

No es necesario siempre tener super definido algo para testearlo, hay testers exploratorios que con solo mirar y usar un poco un sistema pueden darse cuenta de fallos. A veces es contraproducente esperar esa definición. Depende siempre del contexto. No es lo óptimo.

Para estos casos sin contexto es esencial que estén definidos algunos puntos con los que debe ser consistente lo que se testea.

La regla de negocio no hay heurística que la pueda reemplazar.

Si hay que testear cálculos, porcentajes, reglas muy complejas necesito si o si la definición sino es muy difícil.

### Heurística “HICCUP”

El producto (desde el punto de vista del tester) debe tener consistencia en su:

- History: ¿Algo cambió sin aviso?
- Image: ¿Se ve profesional?
- Comparable Product: ¿El producto similar tiene este comportamiento?
- Claims: ¿Coincide con lo que se habla sobre producto?
- User Expectation: ¿El usuario espera este comportamiento?
- Product: ¿Hay cambios en la terminología / cómo funciona / look and feel dentro del mismo producto?
- Purpose: ¿Es consistente con el propósito del software en sí? Por ejemplo: valores negativos en el tamaño de letra en Word

Usando estos puntos podemos expresar por qué pensamos que algo es un bug.

08/11

### 3. 4 Conjetura de errores:

- Heurísticas
- Llamada prueba de sospechas
- “Sospechamos” que algo puede andar mal basado en experiencia o suposición (parte de código tocada por muchos, por historial, por regla de negocio cambiante etc).
  - Enumeramos una lista de errores posibles o de situaciones propensas a tener errores.
  - Creamos casos de prueba basados en esas situaciones
- No hay un respaldo atrás más que lo que piensa una persona
- Es un proceso muy efectivo. Formalizado a partir del análisis de las fallas.
- El programador es quien puede darnos información más relevante.

Orígenes:

- Partes complejas de un componente
- Circunstancias del desarrollo

Creatividad juega un papel clave:

- No hay una técnica para la conjectura de errores
- Es un proceso intuitivo y ad hoc
- Se basa mucho en la experiencia

Cuándo hacerlo:

- Un componente o parte de él, está hecho a las apuradas
- Un componente modificado por varias personas en distintos momentos.
- Un componente con estructuras anidadas, condiciones compuestas, etc.
- Un componente que sospechamos fue armado por copy paste de varios componentes.

### 4. 5.1 Tomando una “Wish List” - Lista de deseos

Partimos de componentes generados por la etapa de requerimientos.

No es tan precisa como clases de equivalencia pero

- Cada statement (declaración) es un requerimiento funcional
- Un requerimiento no es testeable no es implementable: Si no podemos definir cómo va a ser testeado tenemos un problema. Pensar en cómo testear para solventar la incompletitud de los requerimientos.
- Pensar en “variaciones” de las declaraciones funciona muy bien.
  - Revisar sustantivos / verbos.

Ej: “Debe permitir que los clientes compren cosas”.

Variación: Todos los clientes? ¿Qué tipo de clientes?

Ej: “Debe permitir la inscripción de alumnos...”

Variación: Invertir el verbo, y si me quiero desinscribir?

## 5. 5.2 Tomando “use cases”

Describo como una secuencia de eventos. Cada evento es susceptible a las variaciones del punto anterior.

- La secuencia de eventos dentro de un caso de uso tienen variaciones indicadas por su texto.
- Cada variación de un evento constituye una “condición de prueba”
- Cada condición debe ser ejercitada por al menos un caso de prueba.
- Cada caso ejercitará uno o varios componentes involucrados.
- El conjunto de condiciones y casos constituye la base de la prueba de aceptación funcional.
- Este trabajo no puede hacerse si no se ha realizado el análisis de requerimientos.

## 6. 5.3 Tomando un “Modelo de datos”

La integridad referencial entre tablas y la cardinalidad de las relaciones definen reglas de negocio que deben ser probadas.

Ejemplo:

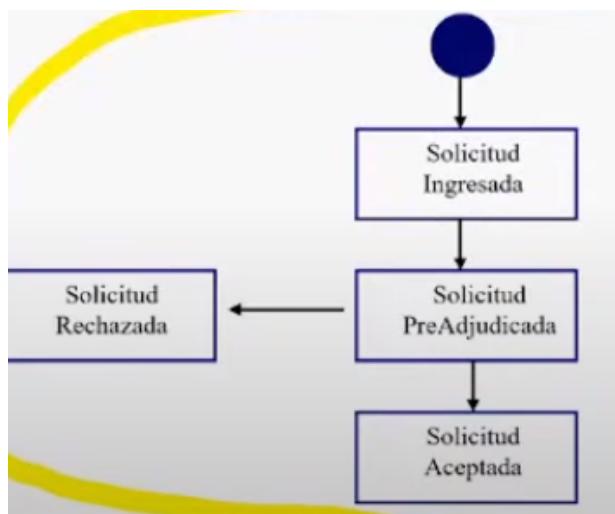
- Una mesa de examen sin alumnos.
- Una mesa de Examen con un Alumno
- Una mesa de examen con muchos alumnos.

Porque hay entidades que no pueden quedar “huérfanas” por ejemplo

## 7. 5.4 Tomando un “Diagrama de transición de estados”

El ciclo de vida de un objeto define reglas de negocio que deben ser probadas.

- Transiciones válidas
- Transiciones inválidas.



Las cajitas y flechitas son reglas de negocio.

#### Conclusiones:

- Ninguna técnica es completa: La combinación de ellas es recomendable.
- Las técnicas atacan distintos problemas.
- Debemos tener muy en cuenta la conjectura de errores (cualquier caso de uso, diagrama, lista para testear reglas de negocio sin saber cómo están implementados)
- Sin especificaciones de requerimientos todo es mucho más difícil.

### Prueba estructural - Caja blanca

Nos interesa cómo se implementó algo, queremos saber cómo se implementó una regla de negocio.

Con estas pruebas puedo medir cuánto de la lógica interna se ejecutó y cuánto no.

- Clear box o glass box también llamada.
- Prueba lo que el SW hace
  - Se basa en cómo está estructurado el componente internamente y su definición.
  - Usada para incrementar el grado de cobertura de la lógica interna del componente. Busco que no me quede parte de la lógica sin probar

Caja negra: pruebo la definición

Caja blanca: pruebo cómo se implementó esa definición.

No es probar lo que dice el código, porque sino siempre daría bien. Es probar contrastando con lo que sabemos de la implementación.

#### Grados de cobertura:

Busca que toda la lógica interna de un componente tenga el mayor grado de cobertura, o sea, que esa lógica interna sea ejecutada, procesada, instanciada, etc. Que no me quede parte de la lógica sin ejecutar.

- **Cobertura de Sentencias**  
Prueba cada instrucción, cada línea de código.  
Ej: Pintadas 100% con una herramienta de análisis
- **Cobertura de decisiones**  
Pruebas con salida de un “IF” o “WHILE”  
Si tenés 100% de cobertura de decisiones tenés también de sentencias.  
Ej: Pintadas 100% todas las decisiones, pueden pasar por un condicional pero nunca irse por el sí, que pase por todos los caminos.
- **Cobertura de condiciones**  
Prueba cada expresión lógica (A and B) de los IF, WHILE.  
Decisiones encadenadas como una “mega” decisión, tengo que buscar todas las combinaciones posibles.
- **Prueba del camino básico**  
Prueba todos los caminos independientes.  
Es calculable, complejidad ciclomática.  
Es la más completa.

#### Aclaraciones:

- Puedo tener 100% de cobertura de sentencias pero no de decisiones.
- Más abajo cumple al 100% lo de arriba. Ejemplo: Si tengo cobertura de decisiones tengo 100% cobertura de sentencias. Si tengo cobertura de condiciones tengo cobertura de decisiones.

#### **Complejidad ciclomática:**

Número de caminos independientes que se necesitan para testear todas las situaciones de lógica que tiene metida una pieza de código.

Métrica de software que proporciona una medición cuantitativa de la complejidad lógica de un programa.

- Cantidad de caminos independientes
- Camino independiente: Agrega un nuevo conjunto de sentencias de procesamiento o una nueva condición.

Relacionado con la mantenibilidad, complejidad alta y mantenibilidad baja.  
es directamente proporcional al esfuerzo de mantenibilidad

Formas de calcularla:

- Número de regiones del grafo
- $V(g) = A - N + 2$  (A = aristas, N = nodos)

## Prueba de caja gris:

Conoces los requerimientos (caja negra) y una parte de la implementación (caja blanca) que te permite generar condiciones adicionales que nunca podrías poner solo con caja negra.

Prueba que combina elementos de la caja negra y caja blanca.

No es negra porque se conoce parte de la implementación o estructura interna y se aprovecha ese conocimiento para generar condiciones y casos que no se generarían naturalmente en una prueba de caja negra.

Conocimiento parcial no total. Total es caja blanca.

Es más completa que caja negra porque agregas condiciones.

### Conclusiones:

Caja blanca es un importante complemento a las pruebas de caja negra, me agrega valor.

- Hay defectos que serían casi imposibles detectarlos a través de la caja negra.
- Los defectos que originan las fallas son encontrados más rápidamente bajo caja blanca lo que deriva a una prueba más económica.

Es caja negra + caja blanca.

Las pruebas de caja gris prueban el SW como si fuera caja negra, pero suman condiciones y casos adicionales derivados del conocimiento de la operación e interacción de ciertos componentes de SW que componen la solución

## Tipos de pruebas:

### 1. Prueba unitaria:

- Se realiza sobre una unidad de código claramente definida
- Generalmente lo realiza el área que construyó el módulo: Pruebo lo que acabo de codear.
- Se basa en el diseño detallado
- Comienza una vez codificado, compilado y revisado el módulo
- Los módulos altamente cohesivos son los más sencillos de probar: No tienen que estar conectados a muchas cosas para poder probarlos, mientras más independiente es mejor para probar.

### 2. Prueba de integración:

Orientada a verificar que las partes de un sistema que funcionan bien aisladamente también lo hacen en su conjunto.

Depende de la arquitectura que tengas y la complejidad de interacción con otros sistemas.

Tipos:

- No incrementales: Big bang
- Incrementales: Bottom-up, top-down, sandwich.

Puntos claves:

- Conectar de a poco las partes más complejas
- Minimizar la necesidad de programas auxiliares

### 3. Prueba de aceptación de usuario:

La debe hacer el usuario no nosotros, debe comprometerse.

Prueba de caja negra pura

Prueba realizada por los usuarios para verificar que el sistema se ajusta a sus requerimientos.

- Las condiciones de pruebas están basadas en el documento de requerimientos
- Es una prueba de “caja negra”

### 4. Pruebas no funcionales:

Buscan comprobar la satisfacción de los requerimientos no funcionales del SW

No prueba el qué sino el cómo. (rápido, seguro, amigable, etc).

**Se clasifican de acuerdo al req. no funcional bajo testeо:**

- *Volumen*
- *Perfomance*
- *Stress*
- *Seguridad*
- *Usabilidad*
- *Otras pruebas:*
  - *Recuperación*
  - *Portabilidad*
  - *Escalabilidad*
  - *Etc ...*

#### 4.1 Volumen:

Orientada a verificar que el sistema soporta los volúmenes máximos definidos en la cuantificación de requerimientos.

- Capacidad de almacenamiento
- Capacidad de procesamiento
- Capacidad de transmisión.

#### 4.2 Performance:

Orientada a verificar que el sistema soporta los tiempos de respuesta definidos en la cuantificación de requerimientos en las condiciones establecidas.

Se evalúa la capacidad de respuesta con diferentes volúmenes de carga.  
Ayudan a identificar “cuellos de botella” y causas de degradación  
Se realizan de la mano de las de “volumen”.

#### 4.3 Estrés:

Excede los límites de capacidad definidos en la cuantificación de requerimientos.. ¿Hasta dónde se la banca? Cuándo se degrada/rompe.

- Capacidad de almacenamiento
- Capacidad de procesamiento
- Capacidad de transmisión.

Se busca el punto de ruptura

Busca ver el comportamiento en términos de estabilidad, disponibilidad, manejo de errores.

#### 4.4 Seguridad:

Orientada a probar los atributos requerimientos de seguridad del sistema (si puede ser vulnerado, si el control de acceso es adecuado, etc).

Necesitas mucho skill para realizarlas. Son difíciles.

##### *– Ejemplo: Penetration Test*

- Simula el accionar que puede realizar un intruso.
- Persigue conocer el nivel de seguridad y exposición de los sistemas ante la posibilidad de ataques
- Se basa en un conjunto de técnicas que permite realizar una evaluación integral de las debilidades de las aplicaciones
- Se practica desde diferentes puntos de entrada:
  - *Internos*
  - *Externos*



#### 4.5 Usabilidad:

Orientada a probar los atributos de usabilidad definidos en los requerimientos del sistema.

##### *Las pruebas de usabilidad persiguen:*

- ¿Cuáles son los principales problemas que evitan que el usuario complete su objetivo?
- ¿Cómo la gente usa o usaría el producto para un fin determinado?
- ¿Qué elementos o aspectos hacen sentir frustrado al usuario?
- ¿Cuáles son los errores más frecuentes?
- Se suele medir:
  - *Éxito en la tarea*
  - *Tiempo en la tarea*
  - *Errores en la tarea*
  - *Satisfacción “subjetiva”*



## 5. Pruebas de regresión:

Orientada a verificar que luego de introducido un cambio en el código la funcionalidad original no ha sido alterada y se obtengan comportamientos no deseados o fallas en módulos no modificados.

Probar lo viejo

La automatización es una buena opción.

## 6. Prueba de humo (smoke test):

Orientada a verificar de una manera muy rápida que en la funcionalidad del sistema no hay ninguna falla que interrumpa el funcionamiento básico del mismo.

- No tiene mucha preparación ni es muy sofisticada.
- Se busca ver que algo que compiló al menos corre. Muy alto nivel, sin detalles.
- Es rápida, para ver si algo se rompió o dejó de compilar.

## 7. Pruebas alfa y beta:

Se entrega una primera versión al usuario que se considera está lista para ser probada por ellos.

- Normalmente plagada de defectos
- Una forma económica de identificarlos (ya que otro hace el trabajo).
- En muchos casos no puede hacerse: No puedo lanzar cierta aplicación de facturación con bugs.
- ALFA: La hace el usuario en mis instalaciones (en un entorno controlado y suele estar el developer)
- BETA: La hace el usuario en sus instalaciones. (El entorno no es controlado por el desarrollador y el usuario es quien registra las fallas y reporta regularmente).

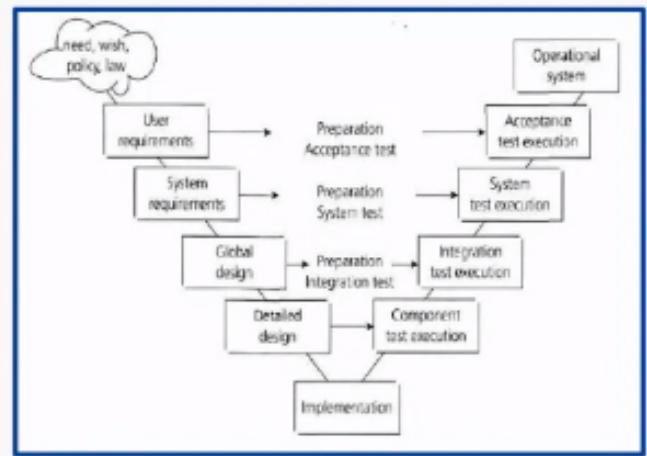
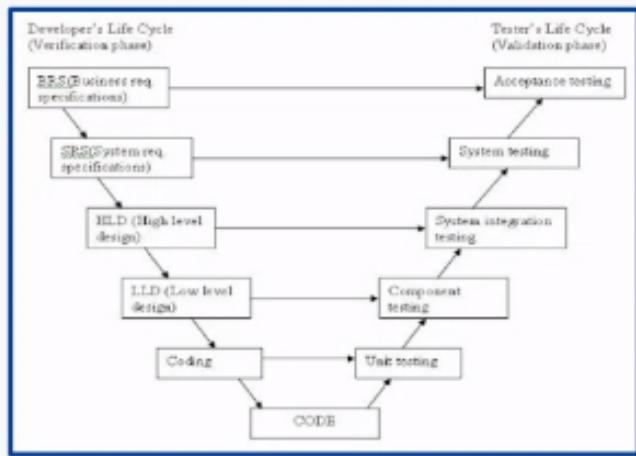
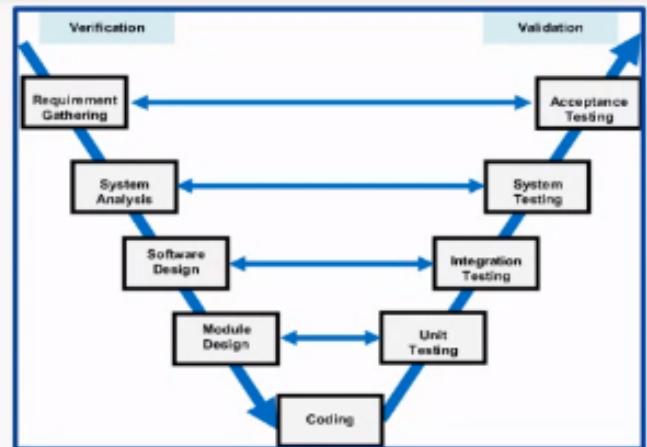
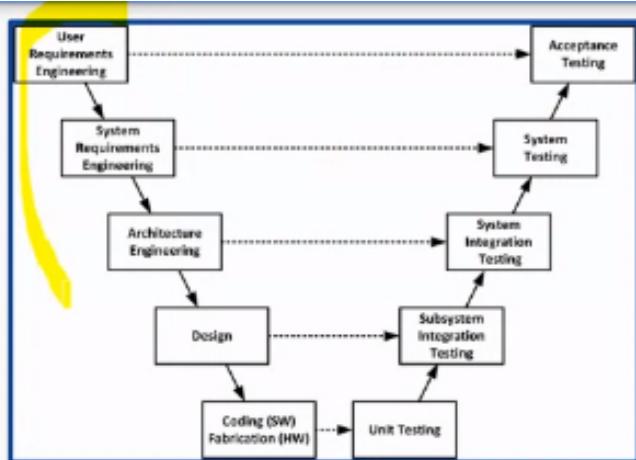
## V-Model:

-Muestra para cada parte del ciclo de vida que hay un tipo de testing asociado.

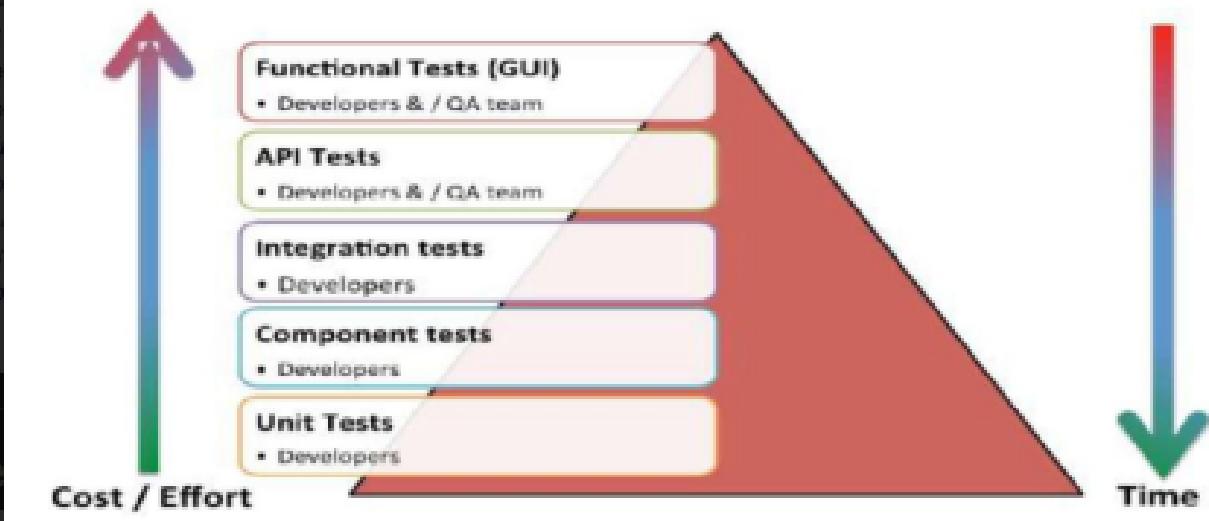
-Sugiere que cuando tengo definidos los requerimientos ya puedo probar las pruebas de aceptación, si ya tengo la arquitectura puede pensar los test de integración, etc.

-Cuando estoy en la etapa o la estoy pasando ya puedo ir pensando en los tests.

No alienta que hagas el camino de V (primero todo el desarrollo y al final de todo testing)



## Pirámide del testing



## Testing guionado - script driven testing:

Dado estos requerimientos genero las condiciones, hago los casos y lo pruebo.  
De antemano defino todo.

Se confeccionan las condiciones y casos tempranamente para luego ser ejecutados. Muchas veces, en cada ciclo de prueba se vuelven a repetir las condiciones y casos y no se retroalimentan. Cada ciclo miro lo mismo no puedo experimentar.

Desarrollo de condiciones dedicado a profesionales con más skills y ejecución y reporte para personas con menos skills.

Ventajas:

- Puede ser objeto de revisión entre pares (Testers y usuarios finales)
- Puede ser reusado para una nueva ejecución fácilmente
- Puede ser medible (condiciones y casos).

Desventajas:

- Laborioso
- La etapa más creativa se da en el diseño de condiciones y casos y no en la etapa de ejecución.

### **Testing exploratorio - unscripted:**

Voy generando las situaciones y condiciones a medida que ejecuto las pruebas.

El trabajo creativo se hace durante la ejecución (no antes).

También conocido como “AdHoc testing”

Es el aprendizaje, diseño y ejecución de la prueba en forma simultánea. El tester va decidiendo tácticamente qué es lo mejor en cada paso de acuerdo al conocimiento que va adquiriendo de la ejecución de un test.

Apunta a tener más variantes y no tener restricciones del “script driven” no que sea sin preparación.

### **Etapas básicas recomendadas:**

1. Reconocimiento y aprendizaje:  
Identificar toda la info que nos permita conocer qué es lo más importante a probar y cómo hacerlo. Depende de la habilidad del tester para identificar los riesgos tanto de la aplicación como de la plataforma.
2. Diseño:  
Crear una guía draft para probar
3. Ejecución  
Ejecutar los casos y registrar los resultados.
4. Interpretación:  
Obtener conclusiones de lo probado.

Ventajas:

- Se pueden variar los test sobre la marcha de acuerdo a lo que se considere más apropiado.
- Permite mayor cobertura de situaciones sobre posibilidades difíciles de anticipar: Porque puedo ir variando los tests y ejecutar distintos.

Desventajas:

- Puede perderse la capacidad de “reproducir” sino se sigue un orden, plan o charter, a la hora de probar (plan no es script). No poder volverlo a replicar.
- Muy dependiente de las personas: Su calidad, memoria, intuición, conocimiento del negocio para explorar.

**Cómo empezar con exploratory test:**

- ¿Cómo empezar? Hay distintos enfoques:
  - *Haciendo un draft de la arquitectura*
    - Distinto tipo de modelos
  - *Brainstorming the tipos de condiciones/casos a ejecutar*
  - *Imaginando todas las posibles formas de “fallar” que puede experimentar la aplicación*
  - *Haciendo preguntas “context-free”*
    - Quién? / Cuándo? / Qué? / Por qué? / Cómo / Dónde?
  - *Revisando especificaciones / manuales de usuario*
  - *Uso de heurísticas*
  - *Etc . . . . .*

**Armado de un charter:**

Armado del “charter”:

- *Un charter define la “misión” de la sesión de testing*
  - Qué se debería testear
  - Cómo se debería testear
  - Qué tipo de defectos buscar
- *Un charter NO es un plan detallado*
- *Ejemplo*
  - Opción “high level”: Analizar la función “Insertar Gráfico”
  - Opción “detailed level”: Ver el comportamiento al insertar varios tipos de gráficos en distintos docs. Poner foco en el consumo de recursos y el tiempo de respuesta

## El Testing & su Contexto

- Hay numerosos y variados aspectos en c/proyecto que influyen a la hora de testear:
  - Calendario / Presupuesto
  - Skill Testers
  - Herramientas / Ambientes
  - El tipo de producto a testear
    - Conocimiento previo del dominio
    - Conocimiento previo del producto
  - Objetivos de “calidad” (ponderación del cliente)
  - Etc . . . . .
- Cada una de las consideraciones influyen a la hora de seleccionar cómo testear

### Cuándo es apropiado aplicar exploratory testing:

- Se necesita conocer el producto rápidamente
- Se demanda feedback en poco tiempo
- Se ejecutó “scripted testing” y se quiere diversificar la prueba
- Se pretende chequear el testing realizado por un tercero a través de una breve prueba independiente
- Hay que atacar un riesgo en particular
- Hay que buscar un bug puntual previamente reportado

### Conclusión:

- Exploratory testing es una forma de encarar el testing
- ET es dependiente del tester (skill - experiencia - personalidad)
- ET es dependiente del conocimiento que se va obteniendo a medida que se ejecuta la prueba.
- No se trata de “script based testing” vs “unscripted based testing”. Deben convivir. El scripted se puede automatizar el otro no.

### Testing metrics (Métricas de testeo):

- Productividad diseño:  
Ej: Condiciones construidas por unidad de tiempo  
Ej: Medir productividad de un equipo en cuanto a qué capacidad de construcción de condiciones tiene en un tiempo.
- Productividad ejecución:  
Ej: Casos ejecutados por unidad de tiempo.

Ej: Capacidad operativa de correr los casos que pueblan las condiciones en un tiempo.

- Eficacia pre release:

Qué tan buenos o no somos en lo que hacemos.

Ej: Incidentes reportados aceptados / total incidentes reportados x 100

Es interesante medir también la eficacia en la resolución de defectos.

Se mide de todo lo que reporté cuantas son fallas realmente.

- Eficacia post release:

Ej: Fallas reportadas / fallas reportadas + fallas reportadas por user \* 100

Cuánto evité que se filtrara al usuario.

- Calidad del desarrollo:

Ej: índice de severidad por defectos reportados

- Release readiness:

Ej: Índice de severidad por defectos abiertos.

Saber si podés salir a un ambiente según los defectos que tenés abiertos.

## **Testing automatics:**

Las pruebas automáticas tienen que ser un complemento de los manuales:

- no buscan eliminar o bajar recursos de testing
- no buscan suplantar testers
- no es una meta sino una alternativa.

Las automatizaciones abarcan un amplio espectro del proceso de testing (desde gestión de incidentes, prueba de regresión y generación de casos de uso)

Automatizar, como cualquier proceso de desarrollo de SW, lleva tiempo y esfuerzo y un mantenimiento a medida que nuestro producto cambie.

Hay ciertos tipos de testing que por su magnitud/complejidad son excesivamente costosos (hasta imposibles de realizar) sin la ayuda de herramientas (inviabilidad técnica).

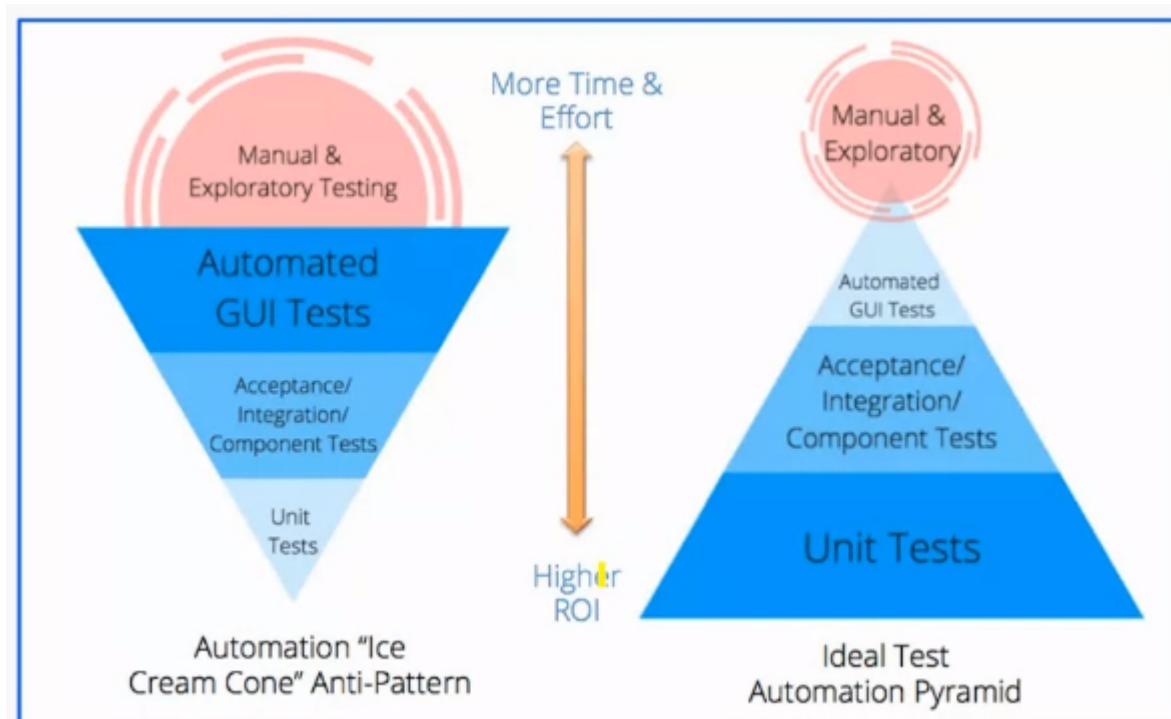
## **Clasificación herramientas de testing:**

## Clasificación Herramientas Testing \*

- Herramientas para gestionar las pruebas y los test cases
  - Gestión de pruebas
  - Gestión de requerimientos
  - Gestión de incidentes
  - Gestión de configuración
- Herramientas para testing estático
  - Revisión
  - Análisis estático
  - Modelado
- Herramientas de soporte para la especificación de pruebas
  - Diseño de pruebas
  - Preparación de datos de prueba
- Herramientas para soporte de ejecución y registración
  - Ejecución de pruebas
  - Pruebas Unitarias
  - Comparadores de pruebas
  - Medición de cobertura
  - Pruebas de seguridad
- Herramientas de soporte de Performance y monitoreo
  - Análisis dinámico
  - Pruebas de Performance / Carga / Stress
  - Monitoreo
- Herramientas para soporte a fines específicos
  - Evaluación de calidad de datos
  - Pruebas de Usabilidad

### Pirámide de Cohn: Qué automatizar

Si tengo que probar algo que tiene fuerte dependencia con cosas que cambian en la parte de presentación, es lo menos deseable para automatizar. Debo automatizar los cambios de bajo nivel.



### Ventajas de automatizar:

- Cobertura
- Bajo costo de ejecución
- Multiplicabilidad: Escalar muy rápido.
- Independencia del tester: Si se equivocó o faltó testear algo.
- Consistencia: Si lo hace bien o mal siempre de la misma manera
- Reuso
- Rapidez

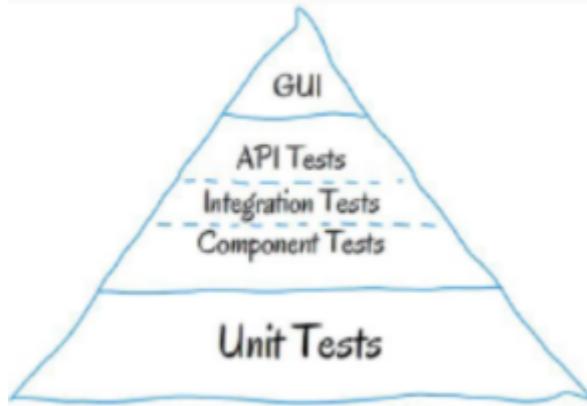
### Desventajas:

- En algunos casos el costo de licenciamiento o alquiler es elevado
- Costo de desarrollo y mantenimiento: Como a cualquier sistema.
- Las fallas pueden deberse a la automatización.

Ver si aplica para tu aplicación:

## La Aplicación

- *Criticidad del producto / aplicación*
  - Relevancia para el negocio
- *¿Producto maduro/estable?*
- *"Volatilidad" de cambios*
  - ¿Qué parte de la aplicación es la que más cambia?
- *Rol de la interfaz de usuario*
- *Frecuencia del testing de regresión*
- *Importancia de los Reqs No Funcionales*
- *Desarrollo In House / Third party Dev / COTS*
  - ¿Tecnología propietaria?
- *Plataformas: OS / DB / Browsers / Devices / Integración / Cloud / ...*
  - ¿Una o varias?
- *Frecuencia de upgrades de las partes*



De arriba para abajo lo más caro.

## El Caso de Negocio

### *La presión por el ROI*

- Lo usual: Ahorro en esfuerzo = Ahorro en dinero ... pero NO es todo

### **De la Prueba Manual:**

- Costo del Diseño/Construcción de Condiciones & Casos de prueba
- Costo de un Ciclo de Prueba

### **De la Prueba Automática:**

- Licencias Herramienta + HW + SW de Base + Training + Mantenimiento
- Costo de Diseño/Construcción de Scripts
- Costo de prueba de los scripts / Change Mgmt & Versionado / BackUp / Etc ....
- Costo de un Ciclo de Prueba

*Por costo tener en cuenta esfuerzo y \$\$\$*

### *La automatización como soporte y liberar recursos (NO reemplazar)*

- Nueva funcionalidad (crecimiento) = Mayor ejecución

### **Visión del "Costo/Beneficio"**

### **Conclusiones de automatización:**

- Acumular primero experiencia en testing manual
- Setear expectativas
- No esperar resultados de manera inmediata
- No pretender automatizar todo
- No perder vista que las condiciones y casos son el activo
- Independizarse lo más posible de una herramienta en particular.
- Separar el equipo de automatización del equipo de ejecución: El de automatización es más perfil developer y el de ejecución perfil operativo.

- Analizar proyecto a proyecto: Su situación en particular.
- Esperar cualquier evaluación de automatizar a contar con un producto aplicación con cierta estabilidad: Ver la volatilidad y el peso de la interfaz de usuario.
- Automatizar es un proyecto más y su entregables es una aplicación más.
- Analizar debe seguir un proceso de medición y evaluación.

**Pregunta 42**

Sin contestar

Puntúa como 1,00

Una "con flechas" las etapas del ciclo de vida con las pruebas respectivas según V-Model

Especificación	Elegir...
Diseño Arquitectónico	Elegir...
Requerimientos	Elegir...
Diseño componente	Elegir...

Respuesta incorrecta.

La respuesta correcta es: Especificación – Prueba de Sistema, Diseño Arquitectónico – Prueba de Integración, Requerimientos – Test de Aceptación de Usuario, Diseño componente – Prueba Unitaria

**Pregunta 89**

Sin contestar

Puntúa como 1,00

Cuando se utiliza Integración Continua (CI), ¿qué tipos de Test son los que normalmente se automatizan?

Seleccione una:

- a. Test de Integración solamente
- b. Test de Integración y regresión
- c. Test Unitario, de integración, y de sistemas
- d. Test Unitarios y de Integración
- e. Test Unitarios solamente

Respuesta incorrecta.

La respuesta correcta es: Test Unitarios y de Integración