

Ingeniería de Software

Resumen

1C-2024 - UTN - FRBA

Curso: K4152 (Lunes 19 a 21)

Docentes: Marcelo Dalceggio, Diego Mansilla, Cecilia Collazo

Índice:

U1. Introducción a la Ingeniería de Software	9
Materiales:	9
Definición de Ing. de software	9
Body of Knowledge	9
SWEBoK (Software Engineering Body of Knowledge)	9
Problemas habituales en el desarrollo y mantenimiento de SW	10
Diagrama CAUSA-EFECTO	11
Paper: Classic Mistakes	11
Según Otero:	20
Paper: ISO 25000 - The Applicability of ISO/IEC 25023 Measures in the Integration of Agents and Automation Systems	21
Adecuación Funcional	21
Eficiencia de Desempeño	21
Compatibilidad	22

Usabilidad	22
Fiabilidad (reliability)	23
Seguridad	23
Mantenibilidad	24
Portabilidad	25
Misc: Menti de atributos y subatributos	29
U2: Modelos de calidad de software	32
Materiales	32
PPT U2: Modelos de Calidad de SW	32
Qué es la calidad	32
Costos asociados a la calidad y a la no-calidad	32
Visiones de la Calidad	33
Visión de producto: ISO 25000 - Calidad del producto de SW	34
Visión manufactura/proceso: Calidad del proceso	34
CMMI (Capability Maturity Model Integrated)	34
ITIL	36
ISO 15504 (SPICE)	36
Paper: When good enough software is best	36
Casos de ejemplo	37
Paper: Standing on principle	37
Con clientes	37
Con managers:	38
Las cinco dimensiones de un proyecto de software	38
U3: Software Engineering Approaches	39
Materiales	39
¿Qué es un proyecto?	39
¿Qué es el “Project Management”?	39
Dimensiones de un proyecto de software	39
Roles principales de un proyecto	40
CYNEFIN	41
Contexto Simple	42
Contexto Complicado	42
Contexto Complejo	43
Contexto Caótico	43
SCRUM	44
Valores de Scrum	45
Roles	46
Ceremonias	47
Sprint planning	47

Daily Meeting	47
Sprint review	48
Retrospective session	49
Información adicional (de otros años)	49
KANBAN	50
Conceptos básicos:	50
¿Cuándo uso Kanban?	50
Principios	51
Trabajo dividido en	51
Métricas	52
LEAN	52
Principios Lean	52
Modelo de gestión Lean	52
Filosofía Lean / 14 Principios	52
Paper: Timebox development	54
U4: Estimaciones de software	56
Materiales	56
★ ¿Qué nos preguntamos para estimar?	56
Mail del profesor Diego Mansilla	57
★ Distinciones útiles	58
Por qué fallan las estimaciones	58
Nivel de incertidumbre	59
Tips para estimar	60
ESTIMACIONES CREÍBLES	60
★ Ciclo de estimación (“dorado”)	61
Métodos de estimación	62
Métodos rudimentarios	62
Métodos paramétricos	63
Notas generales sobre métodos	64
Método: Planning Poker --> TÉCNICA PARA LLEGAR AL ESFUERZO, no para estimar completo	64
Reglas	65
Dinámica	65
Finalización	66
Paper: Fundamentals of function point analysis	66
Componentes	67
Pasos	67
Notas de clase	68
Paper: Use case points	68

Proceso	68
Story points (resumen ajeno)	70
Object points	70
Proceso	70
Temas nombrados para el primer parcial	71
U5: Software Configuration Management	74
Materiales	74
Notas adicionales	74
Overview en clase	74
¿Qué es la Gestión de Configuración de software?	75
¿Qué es el software?	75
Conceptos clave	76
Item de configuración (IC)	76
ICs de Proceso	76
ICs de Producto	77
Configuración de software	77
Línea Base	77
Trazabilidad	78
Branching	80
Ambiente (Environment)	80
Funciones de SCM (según SWEBOK)	81
F1. Administración del proceso de Gestión de Configuración del Software (SCM)	82
2. Plan de gestión de configuración (CM Plan)	82
Ejemplo de proceso de SCM definido	83
Estrategias de branching	83
Gitflow	84
Github flow	85
Trunk based flow (1:25:00)	87
F2: Identificación de la configuración	88
1. Identificación de la configuración	88
Qué elementos se registran de un IC	89
F3: Control de Cambios de la configuración	89
Software Configuration Control Board (SCCB)	90
Change Management Process	90
Mediciones relacionadas a Change Request	91
F4: Status Accounting de la Configuración	92
F5: Auditorías de la Configuración de Software	93
Tipos de Auditorías	93
F6: Release Management & Delivery	94

1. Software Building	95
Build	95
Integración Continua (CI)	95
2. Software Release Management	97
Definiciones	97
Consideraciones para release management	98
Continuous Delivery (CD) = Entrega continua	98
Etapas habituales de las pipelines:	99
Continuous Deployment/Despliegue continuo	99
Comparación CI/CD/CD	100
CD vs CD	100
Comparaciones misc de internet	101
Deployment Strategies	102
Release Management Metrics	104
F7: Herramientas de SCM	105
Toolings de automatización de entornos	105
DevOps - fusión entre desarrollo y operaciones	106
El problema original	106
Definición de DevOps	107
Filosofía de trabajo: CALMS	107
Prácticas de DevOps	107
Paper: But I only changed 1 line of code	108
Configuration Identification	108
Configuration Control	109
Configuration Status Accounting	109
Auditoría de la configuración	109
Tema 6: Testing de SW	111
Materiales	111
¿Qué es un SW de “calidad”?	111
Aseguramiento de la calidad (QA -> Quality Assurement)	111
Asegurar la Calidad vs. Controlar la Calidad (QA vs QC)	111
Testing	112
Objetivo	112
Definición	112
El proceso del testing	112
Conceptos clave	113
Incidente de testing	113
Equivocación vs. defecto vs. falla	113
Otros conceptos relacionados	114

Condiciones de Prueba	114
Casos de Prueba	114
Criterio de selección (de casos de prueba)	114
Depuración	114
Partición!	115
¿Cómo se parte en clases de equivalencia?	115
Economía del Testing	117
Final de las pruebas	117
Abaratamiento	117
Primeras conclusiones	117
Enfoques de prueba (caja negra/blanca/gris)	118
Prueba funcional - de caja negra	118
Definir condiciones y casos	118
Conclusiones	119
Prueba estructural: de caja blanca	119
Grados de cobertura	121
Prueba de caja gris	122
Conclusiones de enfoques de prueba	122
Tipos de prueba	123
Prueba unitaria	123
Prueba de integración	123
Prueba de aceptación de usuario	124
Pruebas No Funcionales	124
Pruebas de Regresión	126
Prueba de humo / Smoke Test	126
Pruebas alfa y beta	126
V-Model	127
Scripted/exploratory and unscripted testing	129
Exploratory Testing / Ad hoc testing	129
Scripted vs unscripted	129
¿Cuándo es apropiado el ET?	131
Etapas básicas de la Exploration Testing	131
¿Cómo empezar?	131
Armado del “charter”	132
Conclusión	132
Métricas del testing	132
Testing Automation	133
Clasificación de Herramientas de Testing	133
¿Qué automatizar?	134

Pirámide de testing de Kohn (de otros años)	135
Ventajas y desventajas	135
Proceso	135
Objetivo	136
La Aplicación	136
El Caso de Negocio	137
Conclusiones	137
Paper: Testing without a Map, de Michael Bolton (obligatorio) + Using Heuristics Test	
Oracles de Michael Kelly.	138
Paper: Exploratory Testing Explained, de James Bach (Obligatorio?)	139
Funciones de SCM (según SWEBOk)	143
Branching strategies	144
Métricas	145
SCM	
Change Request	145
SCM	
Release Management	145
Mantenibilidad	146
Testing	146
Tipos de Auditorías	147
Tipos de Build	147
CD vs CD	148
Filosofía de trabajo de devops: CALMS	148
HICCUPS.	148
Testing - Conceptos a diferenciar	149
Testing de caja blanca: Grados de cobertura	149

1er parcial

U1. Introducción a la Ingeniería de Software

Materiales:

- PPT Unidad 1.
- Paper: Classic mistakes
- Paper: Métricas ISO 25023 → Aplica a calidad, U2. De hecho se trata en la PPT de la U2.

Nota: Aquellos ítems que están tildados, están completamente resumidos en este archivo y no hace falta releerlos.

Definición de Ing. de software

- La ingeniería en software es aquella que aplica los principios de las ciencias de la computación y las matemáticas para alcanzar soluciones “cost-effective” a problemas de software. (Del Software Engineering Institute)
- La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software. (IEEE)
- **Es la disciplina tecnológica y de administración que se ocupa de la producción sistemática y mantenimiento de productos de software que son desarrollados en tiempo y costo estimados. (IEEE)**

Body of Knowledge

Describe el conocimiento relevante para una disciplina, a fin de que califique como una profesión.

SWEBoK (Software Engineering Body of Knowledge)

De 1998, de la IEEE. Describe el conocimiento que existe de la ingeniería de software. Está conformado por 15 áreas de conocimiento, entre las cuales se encuentran:

- | | |
|---------------------------------|---|
| 1. Software Requirements | 5. Software Maintenance |
| 2. Software Design | 6. Software Configuration Management |
| 3. Software Construction | 7. Software Engineering Management |
| 4. Software Testing | |

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

Problemas habituales en el desarrollo y mantenimiento de SW

- Tiempo: Los proyectos se terminan con mucho atraso
- Costo: Se exceden en el costo estimado
- Alcance: El producto final del proyecto no cumple con las expectativas del cliente
- Calidad: El producto final no cumple con los requerimientos de calidad mínimos

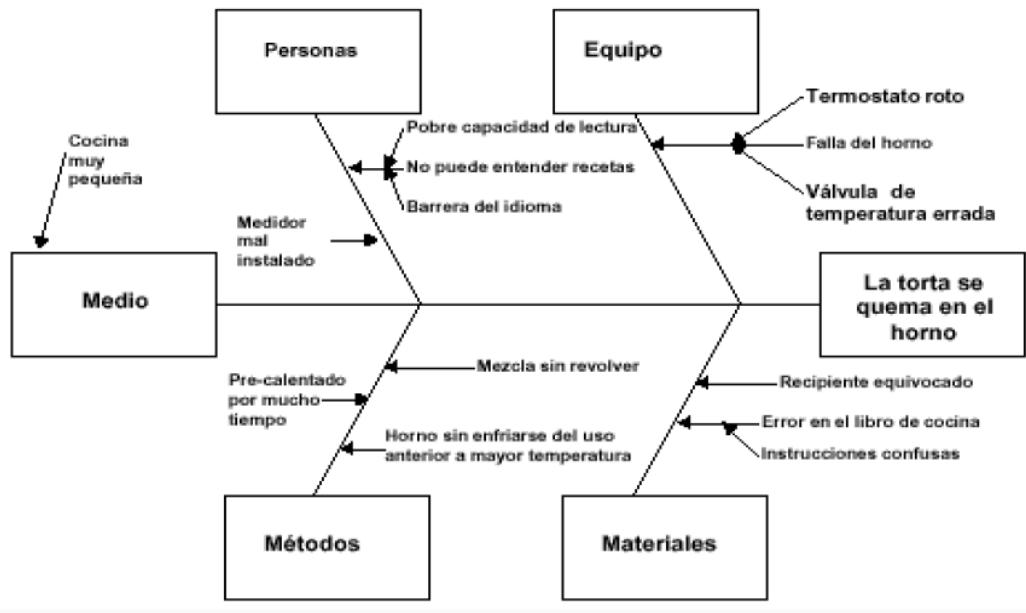
Causas comunes	Reacciones comunes
<ul style="list-style-type: none"> - No hay administración y control de proyectos - El éxito depende de los gurúes - Se confunde la solución con los requerimientos - Las estimaciones son empíricas y no hay historia sobre proyectos realizados - La calidad es una noción netamente subjetiva que no se puede medir - No se consideran los riesgos - Se asumen compromisos imposibles de cumplir - Las actividades de mantenimiento van degradando el software - Se comienzan los proyectos con requerimientos no claros ni estables - Existe una tendencia a ser optimista (simplificando, subestimando, ...) <p>Etcétera...</p>	<ul style="list-style-type: none"> - Buscar un gurú que nos saque del problema - Agregar gente a un proyecto atrasado - Recortar documentación - Recortar la etapa de Testing - Recortar cualquier actividad que no sea codificar - Asumir definiciones en lugar del usuario - Recurrir a una "bala de plata" - Justificar con frases típicas

Diagrama CAUSA-EFECTO

Representación de varios elementos (causas) de un sistema que pueden contribuir a un problema (efecto).

Usado para identificar las posibles causas de un problema específico.

Es útil para la recolección de datos y efectivo para estudiar procesos y situaciones complejas.



Ejemplo de diagrama de causa-efecto, Ishikawa o Espina de pescado

Paper: Classic Mistakes

Son los problemas recurrentes en todos los proyectos de desarrollo y mantenimiento (D&M) de software. Suelen tener un gran impacto negativo en el desarrollo.

Dada la alta probabilidad de que se cometan estos errores durante un proyecto, es importante conocerlos, junto con sus causas y consecuencias para tratar de evitarlos o reaccionar de la mejor manera posible cuando sucedan.

Conclusiones del paper:

It is reasonable to characterize the mistakes surveyed as "**Classic Mistakes**." Most of the mistakes included in the survey were reported to occur **fairly frequently** and to have **significant adverse impact** when they do occur. Thus it is accurate to refer to these mistakes as "**mistakes that have been made so often, by so many people, that the consequences of making these mistakes should be predictable and the mistakes themselves should be avoidable.**"

Table 9 Classic Mistakes with the Highest Mistake Exposure

Rank	Classic Mistake	Average Frequency	Average Severity
1	Unrealistic expectations	60%-70%	Serious
2	Overly optimistic schedules	60%-70%	Serious
3	Shortchanged quality assurance	60%-70%	Serious
4	Wishful thinking	55%-65%	Serious
5	Confusing estimates with targets	55%-65%	Serious
6	Excessive multi-tasking	55%-65%	Serious
7	Feature creep	55%-65%	Moderate-Serious
8	Noisy, crowded offices	60%-70%	Moderate-Serious
9	Abandoning planning under pressure	50%-60%	Serious
10	Insufficient risk management	55%-65%	Moderate-Serious

Lista de los classic mistakes evaluados

Freq	MEI	Severity	Description
	TOP10	Serious	<p>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. Se abandona la planificación realizada y se deja de planificar, cuando hay dificultades o presión por terminar.</p>
		LOW	<p>Adding people to a late project Adding people can take more productivity away from existing team members than it adds through new ones. Agregar 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. Para que sea efectivo agregarlos necesito que sea bajo tiempo de capacitación por experiencia u otra razón, que requiera poca</p>

			comunicación, que las tareas sean independientes/poca coordinación con otras personas y que las tareas que se den sean divisibles.
			<p>Assuming global development has a negligible impact on total effort 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, due to communication and coordination efforts. Studies have shown that international development will typically increase effort by about 40% compared to single-site development.</p> <p>Asumir que tener personas en distintos países no afecta el impacto significativamente</p>
			<p>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.</p> <p>Codear tan rápido como puedas comiéndote muchos bugs o pasando cosas.</p>
TOP 10	TOP10	TOP10	<p>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.</p> <p>Confundir estimaciones de tiempo o costo, con objetivos de negocio.</p>
		LOW	<p>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.</p> <p>Hacer funcionalidades que no son tan requeridas por el solo hecho de probar tecnología o cosas de la tecnología que se está usando.</p>
TOP 10	TOP10	TOP10	<p>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—each task switch in software development can incur a 5-30 minute downtime.</p> <p>Cambiar o tener más de un proyecto trae problemas de productividad y cuesta seguir el hilo rápido.</p>

TOP 10	TOP10		<p>Feature Creep</p> <p>The average project experiences about a 25% change in requirements over its lifetime. Such a change produces at least a 25% 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.</p> <p>Un proyecto promedio tiene un 25% de porcentaje de adiciones de requerimientos/cambios respecto de la planificación inicial. Eso se debe tener en cuenta en el esfuerzo y la agenda del proyecto.</p>
			<p>Friction between developers and customers</p> <p>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.</p> <p>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.</p>
			<p>Heroics</p> <p>Sometimes there is a higher premium placed on a can-do attitude than on steady and consistent progress and meaningful 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>Se confía en la presencia de un programador muy experimentado que salva las papas siempre.</p>
		TOP10	<p>Inadequate design</p> <p>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.</p>
			<p>Insufficient planning</p> <p>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.</p>
TOP 10	TOP10	Mod-Serious	<p>Insufficient risk management</p> <p>The most common problem with risk management is not doing any risk management at all, and the second most common problem is not doing enough risk management.</p>

			No se hace manejo de riesgos para riesgos ocurrentes/classic mistakes que deben considerarse si o si. El problema es no hacer gestión de riesgos o que no sea suficiente.
LOW			<p>Lack of automated source-code control 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. no usar sistemas como GIT de control de versiones entonces se pueden sobreescribir o pisar cosas del mismo archivo sin control.</p>
		TOP10	<p>Lack of effective project sponsorship (PM) 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. Sin un sponsor bien se pueden aceptar estimaciones o tiempos límites poco realistas, o aceptar cambios que traen problemas en el proyecto.</p>
			<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. PM, team, marketing, usuarios, clientes. No hay coordinación ni conformidad de todos o la mayoría.</p>
		TOP10	<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.</p>
LOW			<p>Letting a team go dark 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. Cuando trabajan medio a oscuras, no saben el estado del proyecto, o tiempos límites, impedimentos</p>
TOP 10	TOP10	Mod-Serious	Noisy, crowded offices

			<p>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.</p>
TOP 10		MED	<p>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% to a development schedule. Si no contemplas las tareas necesarias y base pero que no son tan visibles en la descripción de la tarea trae problemas de tiempo.</p>
			<p>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, usually outsourcing motivated by cost savings results in higher costs and longer schedules. Tercerizar para reducir costo trae problemas</p>
LOW		LOW	<p>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. Subestimar el tiempo/curva de aprendizaje de una nueva herramienta o método. Los beneficios de la nueva herramienta se ven más adelante.</p>
TOP 10	TOP10	TOP10	<p>Overly optimistic schedules 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. 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</p>
			<p>Planning to catch up later Project planners commonly plan to catch up later, but they rarely do. Most projects that get behind schedule stay behind schedule. Dejar para después cosas que se deberían planificar en el momento.</p>
			<p>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</p>

			<p>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.</p> <p><i>Usan políticas distintas pero nunca se establece un estándar. Focalizar en resultados y no en políticas.</i></p>
LOW		LOW	<p>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.</p> <p><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></p>
LOW			<p>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> <p><i>Quien hace la planificación y tareas a seguir sabe que están mal pero las aprueba igual sin negociar, luego se tiene que replanificar.</i></p>
			<p>Requirements gold plating 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><i>Agregar funcionalidades sin “justificación” razonable por parte del cliente</i></p>
LOW			<p>Research-oriented development 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><i>Querer mejorar la performance al máximo, velocidad, buenos algoritmos, uso de memoria óptimo pero con poco tiempo (good enough)</i></p>
TOP 10	TOP10	TOP10	<p>Shortchanged quality assurance Projects that are in a hurry often cut corners by eliminating design and code reviews, eliminating test planning, and performing only perfunctory testing. It</p>

			<p>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>Recortar en calidad. Ante falta de tiempo se reduce diseño o reviews de código nuevo, se elimina el testeo o se reduce. Produce varios bugs en el release</p>
			<p>Shortchanged upstream activities</p> <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>Recortar en tareas que no son código, como diseño, arquitectura o requerimientos. Si no analizas te trae problemas.</p>
LO W			<p>Silver-bullet syndrome</p> <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>Implementar nuevas tecnologías o metodologías pensando que en corto tiempo va a funcionar y arreglar todo.</p>
LO W			<p>Subcontractor failure</p> <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>Subcontratar piezas o sistemas aparte con tu sistema in-house que trae problemas de calidad, especificaciones incorrectas, riesgos, etc.</p>
LO W		MED	<p>Switching development tools in the middle of a project</p> <p>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>
			<p>Trusting the map more than the terrain</p> <p>Project teams sometimes invest more confidence in the plans they create than in the experience their project is giving them.</p> <p>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>

			<p>Unclear project vision</p> <p>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.</p>
LOW			<p>Uncontrolled problem employees</p> <p>Failure to deal with problem personnel (e.g., a prima donna programmer) can threaten development effectiveness. Failure to take action to deal with a problem employee is the most common complaint that team members have about their leaders.</p>
			<p>Undermined motivation</p> <p>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.</p>
TOP 10	TOP10	TOP10	<p>Unrealistic expectations</p> <p>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.</p>
		LOW	<p>Wasted time during the fuzzy front end</p> <p>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.</p> <p>Perder tiempo en el comienzo donde se esperan aprobaciones o definiciones de procesos.</p>
		TOP10	<p>Weak personnel</p> <p>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.</p>
TOP 10	TOP10	TOP10	<p>Wishful thinking</p> <p>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.”</p> <p>No optimismo sino pensar/esperar que funcione aunque no hiciste correctamente las cosas.</p>

Según Otero:

Los classic mistakes se subclasifican en 4 categorías: Personas, Proceso, Producto y Tecnología:

1. Personas
 - a. Poca motivación
 - b. Personal débil (no funcionan como equipo)
 - c. Empleados sin supervisión
 - d. Heroicos (toman mucho riesgo)
 - e. Añadir gente a un proyecto atrasado
 - f. Oficinas ruidosas
 - g. Fricción entre desarrolladores y clientes (mala comunicación)
 - h. Expectativas no realistas
 - i. Falta de participación de los involucrados
 - j. Optimismo
2. Proceso
 - a. Falta de gestión de Riesgos
 - b. Falta de planeamiento
 - c. Diseño inadecuado
3. Producto
 - a. Requerimientos que cambian siempre
 - b. Developer Gold-Plating: Agregar features que no son necesarios alargan la agenda.
 - c. Feature creep
4. Tecnología
 - a. "La bala de plata": No hay soluciones absolutas!
 - b. Cambiar herramientas a mitad de proyecto

Estos errores se resuelven con Gestión de Proyecto, Seguimiento, Gestión de Configuración..

¿Qué podemos hacer para hacer frente a los problemas?

Primero, reconocer que tenemos un problema. Armar un equipo interdisciplinario que comparta la problemática y participe con su visión de identificación de causas.

Paper: ISO 25000 - The Applicability of ISO/IEC 25023 Measures in the Integration of Agents and Automation Systems

El paper evalúa características, subcaracterísticas y mediciones que se usan para medir y cuantificar la adecuación de una integración entre un sistema de automatización industrial y agentes de software.

Adecuación Funcional

La capacidad para proporcionar funciones que satisfacen las necesidades **declaradas e implícitas** cuando el producto se usa en condiciones especificadas.

- **Completitud Funcional:** grado en el cual las funcionalidades cubren todas las tareas y objetivos del usuario especificados.
 - Medida: functional coverage: % de funcionalidades pedidas que fueron cubiertas
- **Corrección Funcional:** capacidad del producto o sistema para proveer resultados correctos con el nivel de precisión requerido (las funcionalidades las cumple correctamente - en base a un input, da el output esperado).
- **Pertinencia Funcional:** capacidad del producto para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados. (no hace cosas innecesarias).
 - Medida: functional appropriateness of the usage objective - proporción de las funciones requeridas por el usuario que brinda un resultado apropiado para lograr el objetivo de uso específico.
 - Medida: functional appropriateness of the system - qué proporción de las funciones requeridas por el usuario para lograr sus objetivos da un resultado apropiado.

Eficiencia de Desempeño

Desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones.

- **Comportamiento temporal:** Los tiempos de respuesta y procesamiento y los ratios de throughput cuando lleva a cabo sus funciones bajo condiciones determinadas en relación con un benchmark (banco de pruebas) establecido.

- Medidas: mean response time (first response), mean turnaround time (task completion), rate of task completion (mean throughput)
- **Utilización de recursos:** Las cantidades y tipos de recursos utilizados cuando lleva a cabo su función (bajo condiciones determinadas).
 - Medidas: CPU, memory, IO device utilization, bandwidth utilization.
- **Capacidad:** Grado en que los límites máximos de un parámetro cumplen con los requisitos. (ej: no alojar 500MB para un string)
 - Medidas: transaction processing capacity, user access capacity, user access increase adequacy.

Compatibilidad

Capacidad de dos o más sistemas o componentes para intercambiar información y/o llevar a cabo sus funciones requeridas cuando comparten el mismo entorno hardware o software.

- **Coexistencia:** Capacidad del producto para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes sin detrimiento.
- **Interoperabilidad:** Capacidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.
 - Medidas: data format exchangeability, data exchange protocol sufficiency

Usabilidad

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

- **Capacidad para reconocer su adecuación:** permite al usuario entender si el software es adecuado para sus necesidades.
 - Medidas: completitud de la descripción, demonstration coverage, entry point self descriptiveness.
- **Capacidad de aprendizaje (learnability):** permite al usuario aprender su aplicación.
 - Medidas: user guidance completeness, entry field defaults, error message understandability, self explanatory UI.
- **Capacidad para ser usado (operability):** permite al usuario operarlo y controlarlo con facilidad.
 - Medidas: operational consistency, message clarity, understandable categorization of information, functional customizability, UI customizability, appearance consistency, undo capability, monitoring capability, input device support.
- **Protección contra errores de usuario:** proteger a los usuarios de cometer errores.

- Medidas: Avoidance of user operation error, user entry error correction, **user error recoverability**
- **Estética de la interfaz de usuario:** Capacidad de la interfaz de usuario de agradar y satisfacer la interacción con el usuario.
- **Accesibilidad:** permite que sea utilizado por usuarios con determinadas características y discapacidades.
 - Medidas: accessibility for users with disabilities, supported languages adequacy.

Fiabilidad (reliability)

Capacidad para desempeñar las funciones especificadas (**bajo unas condiciones y periodo de tiempo determinados**).

- **Madurez:** capacidad para satisfacer las necesidades de fiabilidad en condiciones normales (ej: un producto cuando arranca tiene 80 errores, con el tiempo bajan).
 - Medidas: mean time between failure, failure rate, fault correction, test coverage.
- **Disponibilidad:** capacidad de estar operativo y accesible para su uso cuando se requiere.
 - Medidas: Mean down time.
- **Tolerancia a fallos:** operar según lo previsto en presencia de fallos de hardware o software (fail safe).
 - Medidas: failure avoidance (mitigación), redundancy of components, mean fault notification time.
- **Capacidad de recuperación:** recuperar los datos afectados y restablecer el estado deseado en caso de interrupción o fallo.
 - Medidas: Mean recovery time, backup data completeness.

Seguridad

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

- **Confidencialidad:** Capacidad de protección contra el acceso de datos e información no autorizados, ya sea accidental o deliberadamente.
 - Medidas: controlabilidad del acceso, corrección de la encriptación de datos, fuerza de la criptografía.

- **Integridad:** Capacidad del sistema o componente para prevenir accesos o modificaciones no autorizados a datos o programas de ordenador.
 - Medidas: integridad de datos (proporción de ítems de datos que deben ser protegidos y permanecen sin corromper), data corruption prevention (% de uso de métodos disponibles y recomendados que fueron incorporados en el sistema), buffer overflow protection.
- **No repudio:** Capacidad de demostrar las acciones o eventos que han tenido lugar, de manera que dichas acciones o eventos no puedan ser repudiados posteriormente. (Existencia de un log de acciones coherente, consistente e inmutable).
 - Medida: digital signature usage. Proporción de eventos capturados con firma digital.
- **Responsabilidad:** Capacidad de rastrear de forma inequívoca las acciones a la entidad que las realizó.
 - Medidas: user audit trail completeness, system log retention.
- **Autenticidad:** Capacidad de demostrar la identidad de un sujeto o recurso.
 - Medidas: authentication mechanism sufficiency, authentication rules conformity.

Mantenibilidad

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

- **Modularidad:** Capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.
 - Medidas: coupling of components (ratio entre componentes independientes y los que deben ser independientes), ...
- **Reusabilidad:** Capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos.
 - Medidas: reusability of assets, coding rules conformity.
- **Analizabilidad:** Facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.

- Medidas: system log completeness, diagnosis function effectiveness, diagnosis function sufficiency
- **Capacidad para ser modificado:** Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
 - Medidas: modification efficiency, modification correctness, modification capability.
- **Capacidad para ser probado:** Facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.
 - Medidas: test function completeness, autonomous testability, test restartability, autonomous testability,

Otras métricas, de clase: mean time to repair (tiempo medio de reparación), lines of code (analizabilidad, modularidad), complejidad ciclomática

Portabilidad

Capacidad del producto de ser transferido de forma efectiva y eficiente de un entorno hardware, software, operacional o de uso, a otro.

- **Adaptabilidad:** capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos (hardware, software, operacional o de uso).
 - Medidas: hardware environmental adaptability, system software environmental adaptability, operational environment adaptability: all ratios of software functions that successfully operate under new conditions.
- **Capacidad para ser instalado:** Facilidad con la que se puede instalar y/o desinstalar exitosamente en determinado entorno.
 - Medidas: installation time efficiency, ease of installation.
- **Capacidad para ser reemplazado:** ser utilizado en lugar de otro producto software con el mismo propósito y en el mismo entorno (Ej: al instalar Chrome te importa todo solo).
 - Medidas: product quality equivalence, usage similarity, functional inclusiveness, data reusability/import capability

Characteristics	Sub-characteristics	Measure	IA suitability
Functional Suitability	Functional Completeness	Functional coverage	Good
	Functional Correctness	Functional correctness	Very Good
	Functional Appropriateness	Functional appropriateness of usage objective Functional appropriateness of the system	Very Good Good
Performance Efficiency	Time behavior	Mean response time	Very Good
		Response time adequacy	Good
		Mean turnaround time	Very Good
		Turnaround time adequacy	Good
		Mean throughput	Very Good
Compatibility	Resource Utilization	Mean processor utilization	Neutral
		Mean memory utilization	Neutral
		Mean I/O devices utilization	Neutral
		Bandwidth utilization	Good
Capacity	Co-existence	Transaction processing capacity	Poor
		User access capacity	Poor
		User access increase adequacy	Poor
Interoperability	Interoperability	Co-existence with other products	Poor
		Data formats exchangeability	Neutral
		Data exchange protocol sufficiency	Neutral
		External interface adequacy	Poor

		Description completeness	Good
	Appropriateness recognisability	Demonstration coverage	Neutral
		Entry point self-descriptiveness	Very Poor
		User guidance completeness	Very Good
	Learnability	Entry fields defaults	Very Good
		Error messages understandability	Very Good
		Self-explanatory user interface	Good
		Operational consistency	Very Good
		Message clarity	Very Good
		Functional customizability	Good
	Usability	User interface customizability	Poor
	Operability	Monitoring capability	Very Good
		Undo capability	Very Good
		Understandable categorization of information	Very Good
		Appearance consistency	Good
		Input device support	Good
		Avoidance of user operation error	Very Good
	User error protection	User entry error correction	Very Good
		User error recoverability	Very Good
		User interface aesthetics	Very Poor
	Accessibility	Accessibility for users with disabilities	Very Poor
		Supported languages adequacy	Very Poor
		Fault correction	Poor
	Maturity	Mean time between failure (MTBF)	Very Good
		Failure rate	Very Good
		Test coverage	Poor
	Availability	System availability	Very Good
		Mean down time	Very Good
		Failure avoidance	Good
	Fault tolerance	Redundancy of components	Neutral
		Mean fault notification time	Very Good
	Recoverability	Mean recovery time	Very Good
		Backup data completeness	Poor

Security	Confidentiality	Access controllability	Very Good
		Data encryption correctness	Neutral
		Strength of cryptographic algorithms	Neutral
	Integrity	Data integrity	Very Good
		Internal data corruption prevention	Good
		Buffer overflow prevention	Neutral
	Non-repudiation	Digital signature usage	Good
	Accountability	User audit trail completeness	Neutral
		System log retention	Very Good
	Authenticity	Authentication mechanism sufficiency	Very Good
		Authentication rules conformity	Very Good
Maintainability	Modularity	Coupling of components	Very Good
		Cyclomatic complexity adequacy	Neutral
	Reusability	Reusability assets	Very Good
		Coding rules conformity	Good
	Analyzability	System log completeness	Very Good
		Diagnosis function effectiveness	Poor
		Diagnosis function sufficiency	Poor
	Modifiability	Modification efficiency	Neutral
		Modification correctness	Very Good
		Modification capability	Neutral
Portability	Testability	Test function completeness	Good
		Autonomous testability	Very Good
		Test restartability	Neutral
	Adaptability	Hardware environmental adaptability	Very Good
		System software environmental adaptability	Good
		Operational environment adaptability	Very Good
	Installability	Installation time efficiency	Good
		Ease of installation	Very Good
	Replaceability	Usage similarity	Very Good
		Product quality equivalence	Good
		Functional inclusiveness	Good
		Data reusability/import capability	Neutral

Pendiente: revisar el [juego](#)

Misc: Menti de atributos y subatributos

Versión Menti:

- Cuál de los siguientes atributos corresponde a adecuación funcional?
 - NO: Utilización de recursos (subatributo de Eficiencia en el desempeño)
 - NO: Coexistencia (subatributo de Compatibilidad)
 - SI: Pertinencia funcional
- Cuál de los siguientes atributos corresponde a eficiencia de desempeño?
 - NO: Capacidad de recuperación (subatributo de Fiabilidad)
 - NO: Accesibilidad (subatributo de Usabilidad)
 - SI: Comportamiento temporal
- Cuál de los siguientes atributos corresponde a Compatibilidad?
 - NO: No repudio (subatributo de Seguridad)
 - NO: Estética (subatributo de Usabilidad)
 - SI: Coexistencia
- Cuál de los siguientes atributos corresponde a Usabilidad?
 - NO: Utilización de recursos (subatributo de Eficiencia en el desempeño)
 - NO: Facilidad de instalación (subatributo de Portabilidad)
 - SI: Operabilidad
- Cuál de los siguientes atributos corresponde a Fiabilidad?
 - NO: Confidencialidad (subatributo de Seguridad)
 - NO: Autenticidad (subatributo de Seguridad)
 - SI: Madurez
- Cuál de los siguientes atributos corresponde a Seguridad?
 - NO: capacidad de ser reemplazado (subatributo de Portabilidad)
 - NO: Tolerancia a fallos (subatributo de Fiabilidad)
 - SI: Integridad
- Cuál de los siguientes atributos corresponde a Mantenibilidad?
 - NO: adaptabilidad (subatributo de Portabilidad)
 - NO: Corrección funcional (subatributo de Adecuación funcional)
 - SI: Modularidad
- Cuál de los siguientes atributos corresponde a Portabilidad?
 - NO: coexistencia (subatributo de Compatibilidad)
 - NO: Madurez (subatributo de Fiabilidad)

- SI: Facilidad de instalación

Versión alternativa:

- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Pertinencia Funcional?
 - Adecuación Funcional
- - ¿A qué atributo de la ISO 25.000 pertenece el subatributo Comportamiento temporal?
 - Eficiencia de Desempeño
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Coexistencia?
 - Compatibilidad
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Operabilidad?
 - Usabilidad
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Madurez?
 - Fiabilidad
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Integridad?
 - Seguridad
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Modularidad?
 - Mantenibilidad
- ¿A qué atributo de la ISO 25.000 pertenece el subatributo Facilidad de Instalación?
 - Portabilidad

Versión métricas

¿Para medir qué atributo de la ISO 25.000 es apropiada cada una de las siguientes Métricas?

- **Líneas de código:** Mide el número de líneas de código en cada archivo. El valor excluye cadenas de documentos, comentarios y líneas en blanco
 - **Mantenibilidad (analizabilidad)**
 - Fiabilidad
 - Usabilidad

Explicación: Las líneas de código afectan la capacidad de ser analizado el código.

Dependiendo la cantidad de líneas, uno sabe cuán fácil va a ser de mantener o no.

También habla de acoplamiento

- *Tiempo medio de reparación*: Es una medida del tiempo requerido para reparar un sistema y restaurarlo a su funcionalidad completa.
 - Fiabilidad
 - **Mantenibilidad**
 - Compatibilidad

Explicación: NO es fiabilidad (se diferencia de la capacidad de recuperación, porque

- *Tiempo medio de recuperación*: Es una medida del tiempo entre el punto en el que se descubre la falla por primera vez hasta el punto en que el equipo vuelve a funcionar.
 - Portabilidad
 - Eficiencia del desempeño
 - **Fiabilidad (capacidad de recuperación)**
- **Tiempo medio entre fallas:** Mide el tiempo previsto que transcurre entre una falla anterior de un software y la siguiente falla durante la operación normal. O, el tiempo entre un colapso del sistema y el siguiente.
 - **Fiabilidad (madurez)**
 - Usabilidad
 - Mantenibilidad
- **Tiempo medio de respuesta:** Mide el tiempo de espera promedio que experimenta el usuario después de emitir una solicitud hasta que la solicitud se completa dentro de una carga específica del sistema.
 - **Eficiencia de desempeño (comportamiento temporal)**
 - Fiabilidad
 - Portabilidad
- **Cantidad media de rendimiento:** Mide el número promedio de tareas concurrentes que el sistema puede manejar durante una unidad de tiempo establecida.
 - **Eficiencia de desempeño.** (comportamiento temporal)
 - Adecuación funcional
 - Mantenibilidad
- **Cobertura funcional:** Mide las funciones faltantes detectadas en la evaluación versus el número de funciones descritas en la especificación de requisitos.
 - **Adecuación funcional. (completitud funcional)**
 - Fiabilidad
 - Mantenibilidad

U2: Modelos de calidad de software

Materiales

- U2. Presentación de clase.
- Paper: When good enough software is enough
- Paper: Standing on principle
- Juego de Menti con atributos y subatributos

[PPT U2: Modelos de Calidad de SW]

Qué es la calidad

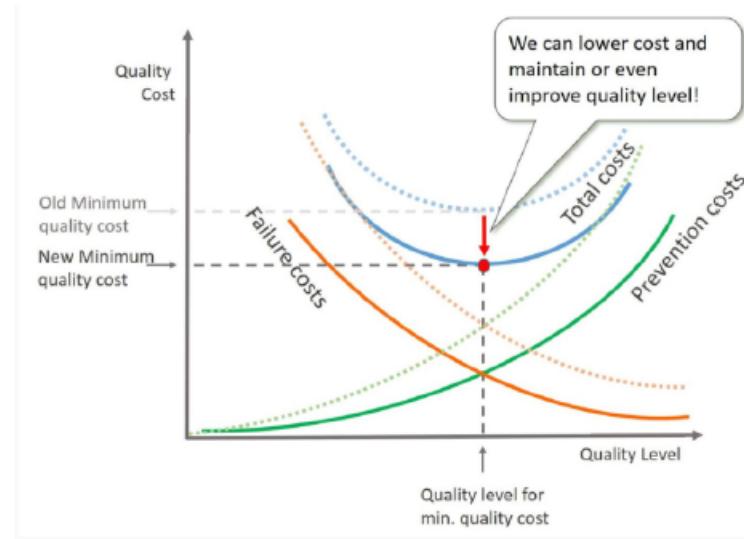
- Cumplir con los requerimientos de alguna persona (Weinberg)
- Adecuación al uso - Satisfacción de las necesidades del cliente (Juran)
- Ausencia de deficiencias (Juran)
- La totalidad de aspectos y características de un producto o servicio que se sustentan en su capacidad de cumplir las necesidades especificadas o implícitas (ISO)

Costos asociados a la calidad y a la no-calidad

- **Costos de calidad:** correspondientes a acciones de prevención y medición (surgen durante análisis, diseño y construcción)
- **Costos de no-calidad**
 - Costos visibles: fallas internas (durante pruebas y despliegue) y externas (durante producción/mantenimiento)
 - Costos invisibles
 - Baja motivación y desgaste de los equipos de trabajo
 - Re-trabajo constante (\$\$)
 - Imagen negativa ante el cliente

Los requerimientos de calidad deben establecerse al inicio junto con los requerimientos del proyecto. Además, la calidad se discute en cada paso del proyecto (arquitectura, construcción, pruebas, aceptación).

Existe un punto en el que elevar la calidad reduce el costo:



Visiones de la Calidad

La calidad puede ser percibida desde 5 perspectivas:

"Subjetivas":

1. **Visión Trascendental:** La calidad se puede reconocer pero no se puede definir (ej: Audi vs. Peugeot)
2. **Visión del Usuario:** La calidad es adecuación al propósito
3. **Visión basada en el valor:** La calidad depende de la cantidad de dinero que el usuario está dispuesto a pagar por el producto

"Objetivas", medibles:

4. **Visión de la Manufactura/Proceso:** La calidad es conformidad con la especificación, cumplir con los requerimientos
 - Minimizar el retrabajo
 - Minimizar el desperdicio
 - Visiones: [CMMI](#), [ITIL](#), [ISO 15504](#)
5. **Visión del Producto:** La calidad está vinculada a las características inherentes del producto.
 - En particular: [ISO 25010](#)

Visión de producto: ISO 25000 - *Calidad del producto de SW*

Ver U1 - [Paper: ISO 25000 - The Applicability of ISO/IEC 25023 Measures in the Integration of Agents and Automation Systems](#)

Visión manufatura/proceso: *Calidad del proceso*

Premisa: manufactura de calidad => producto de calidad

Modelos de calidad de proceso:

1. CMMI - Capability Maturity Model Integrated
2. ITIL
3. ISO 15504 (SPICE)

CMMI (Capability Maturity Model Integrated)

Modelo para la mejora y evaluación de los procesos de desarrollo, mantenimiento y operación de software. Permite ir trabajando el proceso de modo que se va volviendo cada vez más maduro.

- Determina la madurez de un proceso
- Busca mejorar el proceso mediante un camino incremental de mejora
- No es un proceso para desarrollar SW (**NO es una metodología**). Dice qué hacer, pero no dice ni cómo ni quién. Por ejemplo, dice que “todo proyecto debe tener un plan”. Esto permite comparar para ver en qué nivel de madurez se encuentra el proyecto (si tiene o no un plan x ej).
- Está organizado en 5 niveles de madurez.



Proceso: conjunto de actividades que la gente usa para desarrollar y mantener software y sus productos asociados.

CMMI habla de dos cosas del proceso:

- **Madurez: capacidad organizacional, no individual, de cumplir sistemáticamente con los objetivos.** Grado en el que un proceso está:
 - definido y documentado
 - administrado y controlado
 - medido y efectivo
- **Capacidades de un proceso:** habilidad inherente de un proceso de software para producir los resultados planificados. El CMMI busca que la organización produzca productos de alta calidad consistente y predeciblemente.

Proceso Maduro	Proceso Inmaduro
Soportado por la gerencia	La gerencia dice soportarlo...
Definido, documentado, conocido, y practicado	Improvisado sobre la marcha
Existe infraestructura adecuada para soportarlo	Aunque esté definido no se sigue rigurosamente
Adecuadamente medido	No hay entrenamiento formal ni herramientas para sustentarlo
Adecuadamente controlado	Presupuestos y plazos son generalmente excedidos por estimaciones no realistas
Presupuestos y plazos realistas	
Riesgo conocido y controlado	
Proactivo	Es una organización "reactiva"
Es como respirar... institucionalizado	

ITIL

Information Technology Infrastructure Library. 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.

Paper: When good enough software is best

Fuente: Edward Yourdon: When good enough software is best - [IEEE Software. Volume 12. Issue 3](#) May 1995
 pp 79-81

Resumido de lectura propia + clases 1c2024

El gerenciamiento de proyectos de desarrollo de software tiene **5 parámetros**: costo, tiempo, personal, funcionalidad y calidad. Se debe alcanzar un **equilibrio** en cada proyecto, en base a lo

que el cliente/usuario necesita. Para eso, se debe negociar y priorizar constantemente durante el proyecto. Se le debe mostrar los trade-offs al cliente.

Otra clasificación de parámetros: tiempo, costo, alcance. Se pueden optimizar 2, no 3.

El software debe ser suficientemente bueno, no necesariamente perfecto.

El balance entre los factores es elegido por el cliente / marketing. A nosotros nos toca mostrarle los trade offs de elegir cada uno.

Casos de ejemplo

- Desarrollar un sistema completo, el cliente se las arregla con Excel (good enough).
- Tarde nunca es mejor. Cuando ya se está usando un software A, cuesta cambiarlo por otro B por más que sea mucho mejor. Conviene lanzar B antes, con menos funcionalidades/ más bugs, pero good enough.
- El intercambio entre tiempo y personas no es lineal. En la mayoría de los casos, agregar gente a un proyecto atrasado, lo atrasa aún más.

Pregunta de parcial: ¿En qué tareas se pueden incorporar personas para acelerar un proyecto?

- Tareas que requieran bajo nivel de comunicación y supervisión. Para no generar más retraso.
- Tareas paralelizables e independientes, que es lo que provoca el acortamiento del calendario. Tiene que ser una tarea del camino crítico para que genere el efecto deseado.
- Persona que se suma tenga alto skill, conozca el dominio y conozca al equipo

Paper: Standing on principle

Fuente: lectura Vicky + clase 1C2024

Con clientes

- No dejarse convencer por el cliente de hacer un mal trabajo
- Detectar la necesidad real detrás de un pedido de funcionalidad. Buscar soluciones a las necesidades reales del usuario.
 - Sus requerimientos pueden ser solucionados de otra forma
- Ir más allá del interés del cliente primario para ver si se puede resolver una necesidad más amplia con facilidad.
- No dejar que el cliente intervenga en los detalles de diseño/técnicos/ de implementación si no tiene esa expertise.
- Dar la información honesta a los stakeholders, ya sean buenas o malas noticias.
 - Si uno se compromete a cumplir con algo, los demás esperarán que lo cumplas.
 - Si algo cambia, hay que informarlo lo antes posible para poder encontrar una solución alternativa.

Con managers

- Explicar el proceso de estimación, comparar con el del manager. No permitir que un objetivo de negocio se convierta en una estimación.
- Avisar si hay sobrecarga de tareas. Permitir (como PM) que te lo digan.
- **Las estimaciones iniciales tienen un 80% de error. Avisarlo.**
- Explicar los trade-offs en la estimación: más equipo, tiempo, menos funcionalidad, calidad, etc.

Las cinco dimensiones de un proyecto de software

En U3: [Dimensiones](#)

U3: Software Engineering Approaches

Materiales

- PPT U3: Software Engineering Approaches
- Paper: Timebox development

¿Qué es un proyecto?

Es un esfuerzo **temporal** emprendido para crear un producto o servicio **único** con el fin de lograr un **objetivo**.

- Objetivo: Tiene un objetivo o beneficio que lo guía
- Temporal: Tiene fecha de inicio y fin
- Único: no es recurrente, cada uno es diferente al otro.
- Posee recursos limitados
- Consta de una sucesión de actividades o fases en las que se coordinan dichos recursos.

¿Qué es el "Project Management"?

La administración de proyectos es una disciplina que consiste en 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

(Tomado de PPT U3 + Paper Standing on Principle)

Hay 5 dimensiones de un proyecto de software. No son independientes entre sí, y siempre hay trade-offs - no se puede cumplir todo a la vez.

Enfoque de 3 dimensiones *Triple constraint*

Alcance
Costo
Tiempo

Enfoque de 5 dimensiones

Alcance
Costo
Tiempo

Recursos
Calidad

En un proyecto dado, cada dimensión puede tomar uno de 3 roles:

- **Driver:** objetivo vital a lograr, tiene poco/nada de flexibilidad → lo que persigo
- **Restricción:** es un factor limitante, que no está bajo el control directo del líder del proyecto, y también tiene poco/nada de flexibilidad → lo que me limita
- **Grado de libertad:** cualquier dimensión que no es ni driver ni restricción, es un grado de libertad. Un factor que se puede ajustar y equilibrar para lograr los objetivos generales del proyecto. Existe flexibilidad para manejar esta variable → lo que puedo cambiar

No puedo maximizar las 3/5 dimensiones. Hay que identificar la que es el driver, cuál es una restricción y sobre cuáles tengo libertad. Cuando hay un cambio en los requerimientos, hay que acordar sobre qué dimensión se puede ajustar

Ejemplo: En un proyecto, el tiempo puede ser la restricción, y las características junto con la calidad el grado de libertad; mientras que, en otro proyecto como el de implementar el software de un avión, la calidad es una restricción, y el tiempo puede no serlo.

Roles principales de un proyecto

- Stakeholders
 - Son todos los involucrados por el proyecto (incluye a los demás, y también a desarrolladores y demás)
 - 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 (en algunos lugares se llama product owner)
 - Experto en el dominio del problema del proyecto
 - Asegurar su capacidad para la función y su disponibilidad (son muy demandados). Puede tomar decisiones
- Usuarios Directos
 - Interactúan directamente con el sistema
- Usuario Indirectos
 - Hacen uso del sistema, aunque no necesariamente lo operan (ej: sistema que usa el cajero del banco -directo- pero beneficia indirectamente al cliente del banco)

- Equipo de proyecto
 - Líder del proyecto: hace gestión de proyecto, coordinación de recursos, chequea riesgos, desvíos.
 - Individual contributors - personas que ejecutan las tareas en sí (desarrolladores, analistas de db, infraestructura, etc)

Roles de un proyecto	Roles en SCRUM	Explicación
Stakeholders	Stakeholders	Todos los involucrados por el proyecto
Sponsor	~SCRUM Master	Dueño del proyecto. Tiene autoridad para llevar adelante el proyecto.
Usuario Campeón	Product Owner	Experto en el dominio. Define qué es un producto bien hecho.
Usuario directo	-	Operan el sistema
Usuario indirecto	-	Usan el sistema pero no operan
Equipo	Dev team	

CYNEFIN

Marco conceptual para la toma de decisiones. Es un modelo que compara características de 5 dominios de complejidad diferentes:

1. simple
2. complicado
3. complejo
4. caótico
5. desordenado

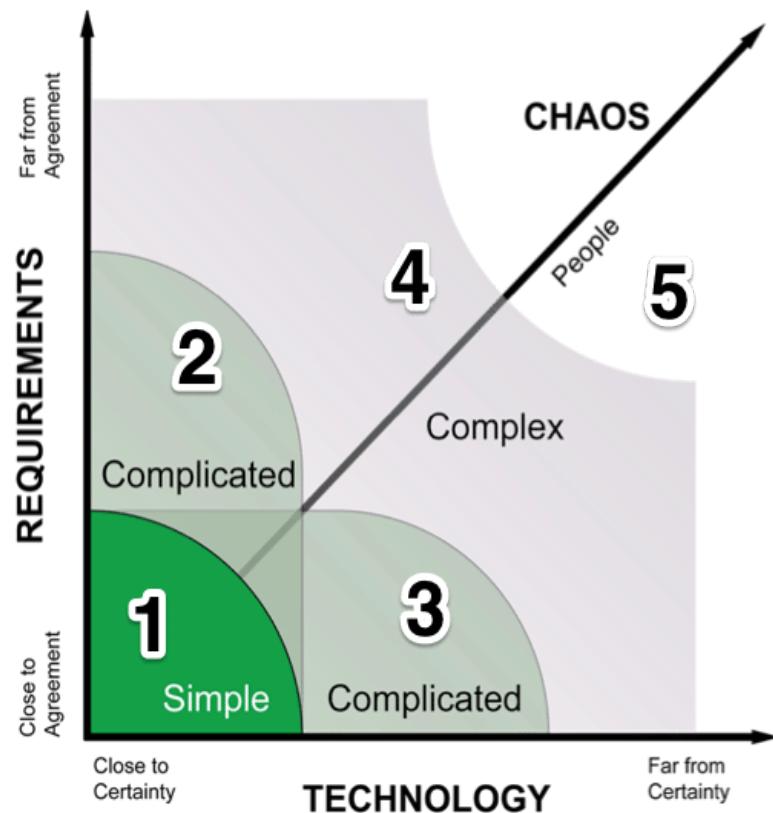
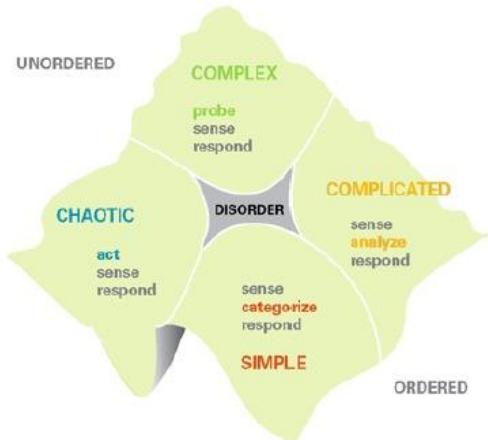


Diagrama de Stacey. Tomado de ADR, otra forma de pensar CYNEFIN.

Este modelo:

- Ayuda a hacer un análisis donde, entendiendo el dominio en el cuál estamos, podemos determinar que tipo de framework nos conviene utilizar.

- Ayuda a los líderes a determinar el contexto operativo prevaleciente para que puedan tomar las decisiones adecuadas.
- Los contextos simples y complicados suponen un **universo ordenado**, donde las relaciones de causa y efecto son perceptibles, y las respuestas correctas se pueden determinar en función de los hechos.
- Los contextos complejos y caóticos están **desordenados**; no hay una relación inmediatamente aparente entre causa y efecto, y el camino a seguir se determina según los patrones emergentes.
- **El mundo ordenado es el mundo de la gestión basada en hechos; el mundo desordenado representa la gestión basada en patrones.**



Contexto Simple

Significa que la situación es estable y existen reglas probadas para aplicar. Las **Mejores Prácticas** son métodos o técnicas superiores a cualquier otra alternativa que producen los mejores resultados que utilizando otros medios.

La relación entre causa y efecto es clara: si se realiza X, se espera Y.

En este dominio se establecen los hechos, se categorizan y se responde con una regla o se aplica una mejor práctica (sense-categorize-respond).

Contexto Complicado

La relación causa y efecto requiere análisis o experiencia; existen múltiples respuestas correctas. Las **Buenas Prácticas** son métodos o técnicas que pueden aplicarse según la decisión de un experto.

Se evalúan los hechos, se analizan y se aplica una buena práctica (*sense-analyze-respond*). Es posible trabajar racionalmente hacia una decisión pero requiere un juicio refinado y con experiencia.

Contexto Complejo

La relación causa y efecto sólo puede ser definida en retrospectiva. No hay respuestas correctas. Los **Diseños Útiles e Informativos** pueden desarrollarse. Se pueden considerar experimentos para encontrar una solución a los problemas. Las soluciones son **adaptativas**.

Se exploran soluciones, se analizan y se responde al problema (*probe-sense-respond*). No es predecible el resultado de las acciones propuestas. El ámbito donde nos encontramos es desconocido y el cambio es constante, lo que no permite anticipar todos los resultados.

Contexto Caótico

La relación causa y efecto es incierta. Los eventos en este dominio son muy confusos para esperar una respuesta basada en el conocimiento. Cualquier acción es la respuesta apropiada. Nos encontramos frente a una crisis donde lo importante es 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*).

★ Contextos comparados

	Significado	Técnica de decisión	Relación causa-efecto	Acción
Simple	Situación estable, existen reglas probadas para aplicar	Mejores prácticas	Relación clara. Si se hace X, se espera Y	Categorizar sense-categorize-respond
Complicado	La relación causa-efecto requiere análisis o experiencia. Hay muchas respuestas correctas.	Buenas prácticas	Decisión del experto. Se evalúan los hechos.	Analizar sense-analyze-respond
Complejo	La relación causa-efecto sólo puede ser decidida en retrospectiva	Diseños útiles e informativos	Se exploran soluciones, se analizan y se responde al problema	Experimentar probe-sense-respond
Caótico	Relación incierta entre causa y efecto	Cualquier acción es una respuesta adecuada	Se actúa para establecer el orden, se analiza lo que tiene algo de estabilidad, y	Actuar act-sense-respond

			se responde para transformar el caos en complejidad	
--	--	--	---	--

★ SCRUM

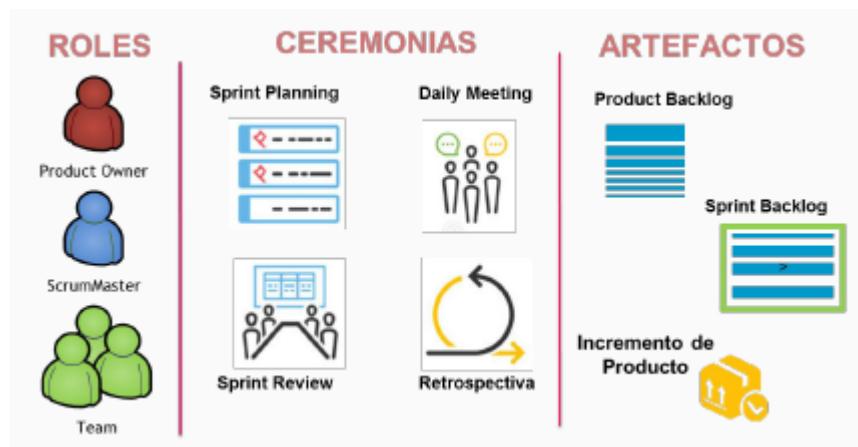
Marco de trabajo (framework) para la administración de proyectos que enfatiza en el trabajo en equipo, en el proceso **iterativo** hacia un objetivo bien definido.

Comienza con una simple **premisa**: “**Comenzar con lo que puede ser visto o conocido. Luego, seguir el avance y modificar lo necesario**”.

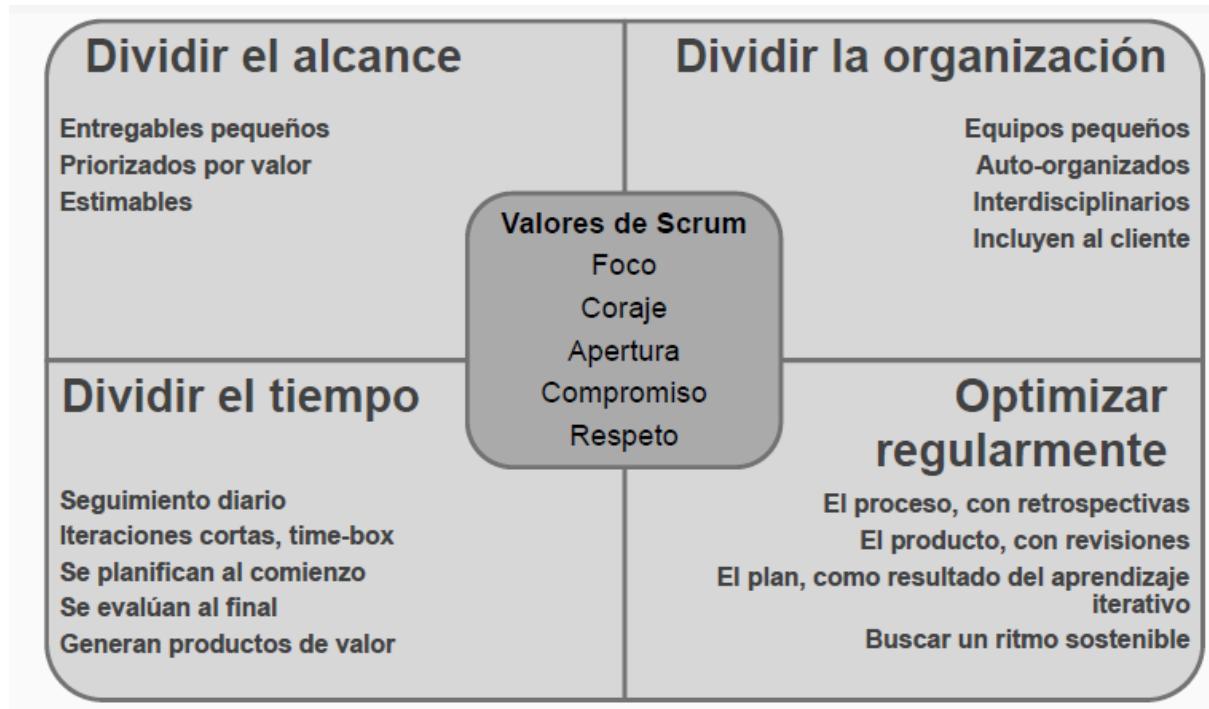
Scrum es un **marco metodológico** pensado para **construir productos de forma incremental**, en una serie de periodos de tiempo llamados **Sprints**.

Un **Sprint** es un período fijo de tiempo (1 a 4 semanas). En cada Sprint el equipo Scrum construirá y entregará un Incremento de Producto.

Cada **incremento** es una versión mejorada del producto que alcanza criterios de aceptación y el nivel de calidad requerido.



Valores de Scrum



Valores: Foco, coraje, apertura, compromiso y respeto

- Dividir el alcance
 - Entregables pequeños
 - Priorizados por valor
 - Estimables
- Dividir la organización
 - Equipos pequeños
 - Autoorganizados
 - Interdisciplinarios
 - Incluyen al cliente
- Dividir el tiempo
 - Seguimiento diario
 - Iteraciones cortas, time-box
 - Se planifican al comienzo
 - Se evalúan al final
 - Generan productos de valor
- Optimizar regularmente
 - El proceso, con retrospectivas

- El producto, con revisiones
- El plan, como resultado del aprendizaje iterativo
- Buscar un ritmo sostenible

Roles

SCRUM MASTER	PRODUCT OWNER	EQUIPO
Dueño del proceso	Dueño de la definición de éxito o terminado	Dueño de los procesos de producción/ingeniería
<ul style="list-style-type: none"> - Colabora con el equipo - Elimina impedimentos - Cuida el proceso. Se asegura de que la metodología scrum se aplique correctamente. - Protege y cuida al equipo 	<ul style="list-style-type: none"> - Representa al cliente/usuarios - Dueño del backlog - Define prioridades - Establece el plan preliminar de entregas - Administra el ROI priorizando los requerimientos - Cuida el valor del producto - Se asegura que el equipo trabaje de forma adecuada desde la perspectiva del negocio 	<ul style="list-style-type: none"> - Pequeño (5-9 personas) - Interdisciplinario - Auto-organizado - Responsables de construir el producto - Define colaborativamente cómo transformar el product backlog en un incremento de funcionalidad al final de la siguiente iteración.

Ceremonias

1. Sprint planning
2. Daily Meeting
3. Sprint review
4. Retrospective session

Sprint planning

- **Objetivo:** definir sprint backlog
- **Formato:** se realiza al inicio del sprint y debe dar respuesta a dos cuestiones:
 - Qué se entrega al final del sprint
 - Cuál es el trabajo necesario para realizar el incremento del producto y cómo se realizará
- **Participantes:** scrum master, product owner, dev team
- **Al finalizar la reunión:**
 - Se compromete el sprint backlog
 - Se asignan las user stories comprometidas
 - Se generan las tareas
- **Lo que se hace:**
 - Definir objetivo del sprint
 - Revisar estimaciones o estimar historias priorizadas (planning poker)
 - Comprometer el sprint backlog
 - (Auto)asignación de historias
- **Lo que no se hace:**
 - Priorizar backlog - ya está priorizado de antes
 - Refinar historias o tareas
 - Revisar o resolver impedimentos
- **Frecuencia**
 - 1x sprint
 - Al inicio de cada sprint
- **Duración:** 1 a 2hs

Daily Meeting

- Objetivo: Sincronizar el trabajo y detectar impedimentos
- Formato:
 - 1 vez por día, al comenzar el día

- Duración: 10-15 minutos
- Participantes: scrum master, dev team, product owner, stakeholders
- Cada miembro del equipo explica:
 - Qué hice desde la última reunión?
 - Qué pienso hacer hasta la próxima reunión?
 - Qué impedimentos tengo para cumplir mis compromisos?
- Al finalizar: el equipo intenta remover los impedimentos.
- Qué se hace:
 - Sincronizar
 - Fomentar participación, comunicación, colaboración de todos
 - Revisar burndown chart
 - Reportar impedimentos
- ¿Qué no se hace?
 - Catarsis
 - Resolver problemas
 - Refinar historias o tareas
 - Tratar temas ajenos al proyecto
 - Extendernos más de 15 min.

Sprint review

- **Objetivos:** revisar el incremento del producto y obtener feedback del negocio
- **Formato:**
 - 1 vez por sprint, al finalizar el sprint
 - 60 minutos
- **Participantes:** scrum master, dev team, product owner, stakeholders
- **Formato:** el equipo expone el objetivo del sprint, se muestran las funcionalidades desarrolladas y se obtiene feedback del negocio.
- **Al finalizar:** se aprueban las características presentadas.
- **Qué se hace:**
 - Mostrar software funcionando
 - Estar abierto a cambios
 - Aceptar y absorber cambios menores sobre la funcionalidad presentada que no impacten en el cierre del sprint actual
 - Solicitar feedback de funcionalidades presentadas
- **Que no se hace:**
 - Catarsis
 - Resolver problemas
 - Refinar historias o tareas

- *Evaluar si las estimaciones fueron correctas*
- Tratar temas ajenos al proyecto

Retrospective session

- Objetivo: revisar la forma de trabajo del último sprint para la mejora continua
- Participantes: scrum master, dev team, product owner
- Formato:
 - 1 vez por sprint, al final.
 - 60 a 120 min.
 - Se inspecciona lo que funcionó bien, lo que no resultó como se esperaba y se quiere mejorar, propuesta de mejoras, lecciones aprendidas, acciones.
- Al finalizar, se obtienen las lecciones aprendidas para mejorar en el sprint.
- Lo que se hace:
 - Revisar el sprint
 - *Evaluar cumplimiento de acciones*
 - Generar nuevas acciones
 - Asignar responsables
- No se hace:
 - catarsis
 - resolver problemas
 - buscar culpables
 - revisar sprints previos
 - planificar el próximo sprint

Artefactos

(Tomado de otros años)

- **Product backlog :** Lista ordenada según prioridades de los requerimientos para un determinado aplicativo. Se va arreglando cada tanto.
 - Ordenado por prioridad de negocio y estimación, no por valor de negocio
 - Se debe mantener actualizado con los nuevos requerimientos
 - No contiene todas las historias desde un principio. No todas las historias tienen que estar desarrolladas ni estimadas.
- **Sprint backlog :** Lo que voy a implementar en ese sprint en particular.

- No se modifica durante el Sprint. El PO no define qué historias se comprometen en el sprint (lo hace el equipo). El equipo no puede tomar historias según su especialidad (tienen que tomarlas indistintamente)
- **Product increment:** La salida del sprint. La suma de todos los elementos de sprint. Existe uno al final de cada sprint. No necesariamente significa que va a ingresar código a producción. Podría ser que por ejemplo corrimos una prueba, probamos hacer algo (aprendimos, lo aprendido va a al backlog). Si tiene errores, también va a al backlog. No es el conjunto de funcionalidades puesta en producción. No incluye al producto testeado.

KANBAN

Kanban es una **administración visual de flujo de trabajo** que ayuda a realizar más con menos estrés. La palabra japonesa "Kanban" significa Tablero Visual.

Viene de Lean.



Conceptos básicos

- Visualizar el flujo de trabajo
- **Limitar el trabajo en progreso**, es decir, cuánto trabajo podes hacer en forma simultánea
- Realizar mediciones y optimizar el flujo
- Busca la mejora continua (calidad, proceso y tiempo de entrega)

El principal objetivo es detectar cuellos de botella que representan estancamiento/bloqueos para avanzar. Evitar la abrumación. Mejorar la comunicación, evitando duplicación y retrabajo.

¿Cuándo uso Kanban?

Cuando se responde afirmativa a alguna de estas preguntas

- Parece que hay miles de tareas para realizar constantemente?
- Siempre hay que cambiar de una tarea a otra perdiendo el foco y sin suficiente seguimiento?
- Parece que se trabaja sin parar sin ser todo lo productivo que se quiere ser?
- Hay problemas de comunicación en el equipo que provocan esfuerzos duplicados, retrabajo, etc?

Cuando:

- Proyectos inestables, con muchos cambios
- Requerimientos que cambian constantemente de prioridad
- En entornos de resolución de incidencias.

Principios

- Comenzar con lo que saben. Entender el proceso actual de desarrollo.
- Acordar perseguir una mejora incremental y evolutiva. Cambios pequeños y que se mejoran a lo largo del tiempo (vs. cambio big bang)
- Promover actos de liderazgo en todos los equipos: no solo el manager es el responsable, el equipo debe promover los cambios en función de sus observaciones
- Enfocarse en las expectativas y necesidades del cliente: comprender lo que necesita el cliente.
- Administrar el trabajo y no a los trabajadores: se respetan roles y responsabilidades existentes, pero se empoderan para auto-organizarse.
- Revisar regularmente los servicios ofrecidos. Se promueve que todos los colaboradores compartan ideas sobre todas las fases del trabajo.

Trabajo dividido en

- To do - Pull: cola de trabajo ordenada, de la cual el equipo va tomando las tarjetas
- WIP / Doing (Work in progress): trabajo en el que estamos trabajando actualmente

- **WIP limit:** se acuerda un límite sobre los ítems sobre los que el equipo trabaja simultáneamente en cada etapa, para evitar el burn out por context switch.

Es lo que diferencia una lista de tareas de un Kanban: se previene el multitasking en favor de la eficiencia o reducción del *lead time*

- Done

Métricas

- Touch time: tiempo neto de trabajo sobre una tarea determinada
- Lead Time: tiempo entre el momento de pedido y el momento de su entrega
- Cycle Time: tiempo entre el inicio y el final del proceso (para un ítem de trabajo)

LEAN

Lean es una filosofía/cultura organizacional 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.

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

Principios Lean

1. Define value
2. Map value stream
3. Create flow
4. Establish pull
5. Pursue perfection

Modelo de gestión Lean

~~El Modelo de gestión LEAN 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.~~

Filosofía Lean / 14 Principios

(Los principios se dividen por peldaño de la pirámide)

Filosofía

1. Gestionar basado en una filosofía (propósito) de largo plazo

Procesos

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
8. Recurrir únicamente a tecnología confiable y testeada para servir a las personas y los procesos

Personas

9. Desarrollar líderes que entiendan el trabajo, vivan 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 **desafiándolos y ayudándolos a mejorar**

Resolución de problemas

12. Ve y observa por ti mismo para comprender profundamente la situación
13. Tomar decisiones por consenso, teniendo en cuenta todas las opciones, e implementar ágilmente
14. Propiciar el proceso de convertirse en una organización que aprende a través de la reflexión (HANSEI) y la mejora continua (KAIZEN)

Según prof. Cecilia Collazzo: Buscan la mejora continua (importante hacer retrospectiva y reflexión para mejorar) - eliminar desperdicio (tiempo muerto entre tarea y tarea) - respeto por las personas - aumentar calidad para tener deliveries sin problemas - **valor para el cliente**

Sistema de Gestión LEAN: 4Ps Ejes y 14 Principios de Gestión



Paper: Timebox development

Fuente: paper leído de primera mano + notas de clase 1C2024

Estrategia de calendarización del trabajo diferente, donde el tiempo no es negociable. El objetivo se debe acotar acordemente.

Es una práctica para el momento de construcción de la solución, que intenta generar una sensación de urgencia en el equipo y mantenerlo enfocado en las funcionalidades más importantes.

Cómo:

- Proyectos de 60 a 120 días, no más. No se alarga el período.
- Se redefine el producto para adaptarse al calendario, enfocando primero en las funcionalidades esenciales, y después en las secundarias.
- Al final se acepta el sistema o se rechaza en su totalidad (porque no anda o porque no sirve)

Se aplica cuando:

- El equipo se puede comprometer
- Hay una lista priorizada de funcionalidades, que diferencia funcionalidades esenciales de opcionales.

-
- hay una agenda realista, con la que el equipo se puede comprometer, que dura entre 60 y 120 días
 - proyectos del tipo indicado: in-house business software, tecnologías de desarrollo rápido, generación de código veloz, con un equipo que ya tiene experiencia en esa tecnología. No algo muy customizado y hecho a medida.
 - hay suficiente involucramiento del usuario final (idealmente, full time)

Riesgos:

- Intentar timeboxear cosas que no son apropiadas
 - Procesos de planeamiento, análisis o diseño (fallar ahí tiene demasiado impacto)
- Sacrificar calidad en vez de funcionalidades
- Jamás se debe extender el timebox.

Beneficios:

- Facilita que el proyecto termine, y no se quede en una meseta en el 90% completado.
- Evita que se usen las soluciones que toman más tiempo para resolver un problema (versión de 2 días, vs 2 semanas vs 2 meses).
- Evita el feature creep.
- Motiva al equipo

Este paper trata de un framework o metodología para llevar adelante proyectos, sólo es aplicable por períodos cortos de tiempo de 1 a 3 meses. Se enfoca en mantener al equipo motivado y con el sentido de urgencia adecuado para poder cerrar un MVP o realizar la última parte de un proyecto más largo. En general una vez concluído con el TBD se premia al equipo por la dedicación, se otorga algún tipo de recompensa.

U4: Estimaciones de software

Materiales

- PPT: U4 Estimaciones de software
- Paper: Fundamentals of Function Point Analysis
- Paper: Comparing Effort Estimates Based on Use Case Points with Expert Estimates

PPT: Estimaciones de software

★ ¿Qué nos preguntamos para estimar?

Requerimientos → Tamaño → Esfuerzo → Costo → Duración

1. (REQUERIMIENTOS): no se estiman, están dados.
2. **TAMAÑO:** es lo primero que hay que preguntarse y estimar: ¿Cuál es el tamaño de lo que tenemos que construir?
Esto incluye: qué se necesita y cuál es su complejidad. La unidad de tamaño (líneas de código, módulos) tiene una traducción a esfuerzo. La conversión depende de la tecnología y metodología que esté usando (la complejidad).
3. **ESFUERZO** se estima ANTES que duración. No se deriva nunca de la duración. Se estima en horas/hombre. El esfuerzo se deriva del tamaño, técnicamente no se estima. Pero si uno usa un método rudimentario de estimación, se estima directamente el esfuerzo.
 - a. **El esfuerzo es la cantidad de horas ininterrumpidas que son necesarias para ejecutar la tarea por parte de una persona que tiene el conocimiento necesario para hacerlo** (fuente: clase 6 1C24). Salvedades:
 - i. Horas ininterrumpidas: puede haber cortes en la realidad
 - ii. Conocimiento necesario para hacerlo: esto se ve afectado si la persona no sabe hacerlo y tiene que investigar.
 - iii. Las horas de esfuerzo y duración son diferentes - las horas calendarizadas tienen un montón de cosas que afectan: cuándo se

trabaja, feriados, cuántas personas participan (overhead por gestión y comunicación), curva de aprendizaje, etc.

- iv. ¿Por qué hablamos de una sola persona? Con más personas hay overhead de gestión, coordinación. El esfuerzo con un equipo mayor suele ser mayor. La duración puede ser la misma o menor.

- 4. **COSTO.** Se deriva del esfuerzo.
- 5. **DURACIÓN:** esfuerzo calendarizado. Esto es una actividad de **planificación**, no de estimación. Convierte el esfuerzo en una línea de tiempo, con tareas paralelizables, dependencias. Incluye consideraciones de recursos y

Mail del profesor Diego Mansilla

Violeta: Vicky / Rosa: Diego Mansilla

- Los requerimientos no se estiman. Están dados.

En realidad, lo que vas a estimar es la implementación de esos requerimientos en el SW, si no tenes requerimientos (iniciales, preliminares, avanzados o alguna forma de requerimientos) no podes estimar. En la PPT están al inicio porque son el punto de partida de las estimaciones.

- El tamaño se estima

Sí, el ideal es intentar estimar tamaño. Se puede estimar con las técnicas paramétricas que mencionamos en la clase

- El esfuerzo se deriva (no estima) del tamaño

Sí y no. Si por algún método estimaste tamaño, podes derivar el esfuerzo con las reglas de conversión tamaño → esfuerzo asociadas a la técnica que utilizaste para estimar tamaño.

Pero si no tenes información suficiente, historia o los inputs necesarios para las técnicas de estimación, entonces podes usar las técnicas rudimentarias de estimación y estimar Esfuerzo.

- La duración NO es estimar, porque es el esfuerzo calendarizado y es una actividad de planificación.

Exacto. Al esfuerzo estimado le aplicas reglas de calendarización y de asignación de recursos y de ahí obtenes la duración.

Por ende yo podría decir que sólo se estima en tamaño y que la duración es parte de otra fase de trabajo. El esfuerzo y el costo serían una especie de cuenta de conversión de unidades de tamaño a otra unidad.

No solo tamaño, **tamaño o esfuerzo**, como comenté antes. Duración y costo se pueden derivar del Esfuerzo.

★ Distinciones útiles

- No estimamos requerimientos. Se estima la implementación de los requerimientos. Se lo usa para estimar el tamaño.
- **Lo único que se estima es: tamaño (en métodos paramétricos) o esfuerzo (en métodos rudimentarios).**
- Esfuerzo!=Progreso (el ejemplo del viaje a la costa en 4 horas vs 10 horas porque fue marcha atrás)
- Esfuerzo != duración. La definición de esfuerzo es horas ininterrumpidas de una persona que sabe lo que hace. Duración: tiempo calendarizado. Pueden coincidir, pero no necesariamente significan lo mismo. Pueden haber tareas paralelizables, por ende la duración puede ser diferente al esfuerzo (mayor o menor). También puede haber dependencias entre las tareas, o entre las personas (una misma persona tiene que hacer dos tareas). Esto hace que no sea una simple multiplicación.
 - Esfuerzo > duración: clase de ing de software. 100 personas x 2hs each: 200hs de esfuerzo, pero dura 2 hs. (Es poco común que la duración sea menor que el esfuerzo)
- **Del esfuerzo se deriva la duración. Nunca al revés. Hacerlo al revés sería partir de la calendarización. La estimación busca el esfuerzo. Pero el trabajo real no es el esfuerzo, se le suman otras cosas.**
- El costo se deriva del esfuerzo, no de la duración.
- Estimar no es calendarizar. Durante la planificación se calendariza. **El calendario NO se estima.** La calendarización es un paso posterior a la estimación.
- “Tardar” habla de duración, calendarizada, no de esfuerzo.

Por qué fallan las estimaciones

- Optimismo
- Estimaciones informales (“estomacales”)
- No hay historia
- Mala definición del alcance / tamaño → se arrastra en esfuerzo
- Novedad / Falta de Experiencia
- Complejidad del problema a resolver
- No se estima el tamaño
- Porque la estimación fue buena pero cuando empieza el proyecto:
 - Mala administración de los requerimientos
 - No hay seguimiento y control
 - Se confunde progreso con esfuerzo

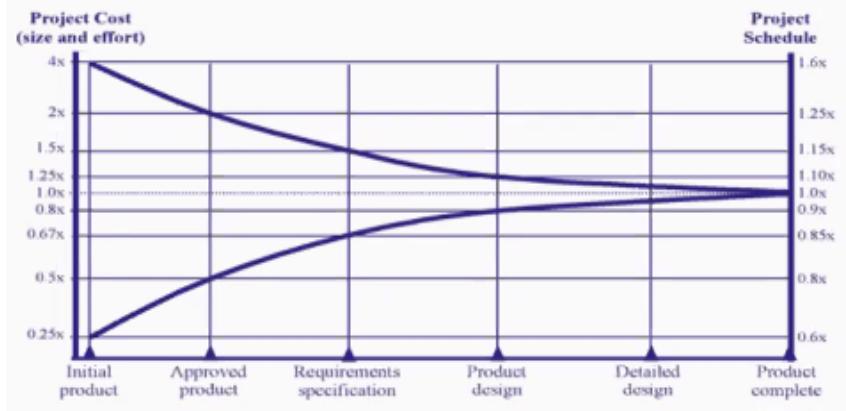
Se debe separar: mala estimación de mala gestión.

Si la realidad es diferente a la que yo esperaba que fuera, la estimación parece haber sido mala. Pero hay que diferenciar la estimación de la gestión. (Ejemplo: estimo gastar n litros para ir a Mar del Plata, pero uso $2n$ porque fui marcha atrás - el problema no es de la estimación).

Sí hay que tener en cuenta algunas cuestiones en la estimación: curva de aprendizaje, ciertas incertidumbres. No la mala praxis en sí.

Nivel de incertidumbre

- El “cono de la incertidumbre” muestra niveles de incertidumbre (probable) de las estimaciones en cada etapa del proyecto, respecto al costo (tamaño y esfuerzo) y a la duración.
- Cuanto más refinada la definición, más exacta será la estimación.
- No asumir que solo por el paso del tiempo y de las fases de un proyecto se avanza con menos incertidumbre en las estimaciones. Acá impacta la gestión. La estimación puede estar bien, pero **puede haber imponentables, cosas no tenidas en cuenta, o MALA gestión.** El proyecto va a avanzar si está bien gestionado, no se puede dejar de prestar atención a eso.



Tips para estimar

- Diferenciar entre estimaciones, objetivos de negocio y compromisos
 - Se estima tamaño, esfuerzo y costo.
- Aclarar la incertidumbre
 - Asociar las estimaciones a un % de confiabilidad
 - Se recomienda presentar rangos en vez de un único valor
 - Se recomienda presentar junto con la estimación los supuestos que se tuvieron en cuenta para llegar a la misma.
 - Si mi estimación falla puedo ver por qué
 - estaban mal los supuestos
 - método incorrecto de estimar.
- Ley de Parkinson: toda tarea se expande hasta ocupar el tiempo que tiene asignado.
 - Relacionado con el timeboxing: si puedo hacer lo mismo en menos tiempo, cuál es el verdadero esfuerzo que toma una tarea? Tener en cuenta si es sostenible, buscar el punto justo.
- Considerar todas las actividades relacionadas al desarrollo de software , no sólo codificación y testing (análisis, diseño, actividades de SCM, testing, etc.)
- No asumir que sólo por el paso del tiempo y las fases de un proyecto se avanza con menor incertidumbre en las estimaciones (cono de la incertidumbre). El paso del tiempo no es lo que define la reducción de la incertidumbre, sino la verdadera refinación de la estimación.
- Recolectar la historia/datos para experiencia futura

ESTIMACIONES CREÍBLES

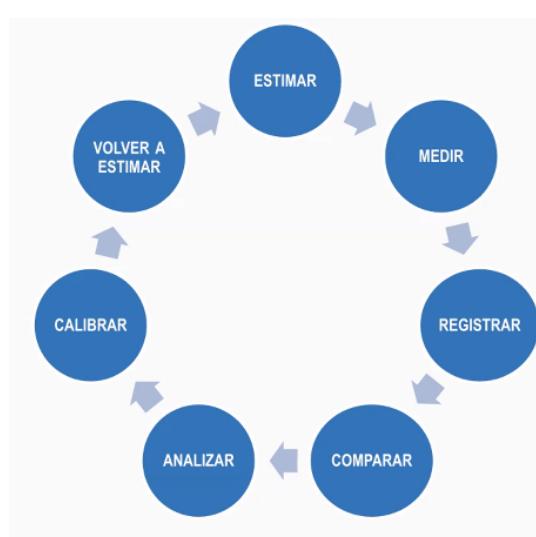
- Las estimaciones las hacen las personas, no herramientas ni modelos

- Se necesitan juicios razonables y compromisos con los objetivos organizacionales que no pueden delegar a modelos automáticos.
- Las estimaciones se basan en comparaciones
 - Para estimar se usan similitudes y diferencias con proyectos previos.
 - Para poder estimar, necesitamos historia de proyectos pasados
 - Las estimaciones se pueden mejorar colectivamente buscando historia
- Un método creíble debe ser de caja blanca (transparente - visible para la persona a la que le estamos transmitiendo la estimación, se debe poder explicar)

Nuestro cliente tiene que saber que tenemos un método de estimación, mostrárselo (que sea una caja blanca). En la medida que cumplamos con eso, ganamos confiabilidad por parte del cliente. Necesitamos también nosotros confiar en nuestro método.

★ Ciclo de estimación (“dorado”)

Se estima (con cualquier método). Si hay desvíos, se ajusta/calibra. Es más difícil calibrar los métodos rudimentarios, el aprendizaje le queda al experto. Los paramétricos generan un método y un aprendizaje general, y se va generando historia.



1- Estimo
2- Ejecuto una serie de pasos metodológica que me permite retroalimentar a mi estimación y ajustar.

Con esto se puede armar un método de estimación confiable. Refino mi proceso una y otra vez con el tiempo.

Cuándo sé si salió bien? Cuando terminó el proceso. Ahí me fijo por qué se desvió, comparando elemento por elemento para calibrar a futuro.

El ciclo se aplica para ir automatizando y mejorando las estimaciones en el futuro.

... → Estimar → medir → registrar → comparar → analizar → calibrar → estimar...

Si hay una estimación, hay que hacer algo con ese número. No estimar si uno no le va a dar bola.

- Estimar: Comenzar por estimar el tamaño para derivar el esfuerzo y costo. Tenemos que saber bien cuáles son las entradas (m_2 , líneas de código, módulos).
- Medir: Mientras evoluciona el proyecto, medir el tamaño, el esfuerzo y el costo incurrido.
- Registrar: anotar las mediciones de forma clara.
- Analizar: Razones de desvíos, supuestos que variaron, temas no contemplados
- **Calibrar:** Ajustar cada una de las variables y parámetros que intervienen en el proceso de estimación. Ej: "en realidad no tuvimos en cuenta que requería un entorno cloud", entonces la próxima vez, si sabemos que si se necesita un entorno cloud lo vamos a tener en cuenta para la estimación.
- **Volver a estimar:**
 - El mismo proyecto (ahora con más info).
 - *Nuevos proyectos*, con la calibración. Independientemente del método que usemos, vamos a tener un método bueno; nosotros vamos a creer en los métodos.

Métodos de estimación

Todos los métodos tienen una secuencia de cosas que se hacen. Lo importante a entender.

Métodos rudimentarios

Se caracterizan porque su estimación principal es del esfuerzo (ni costo ni tamaño), medido en horas humanas. Basados en la experiencia e intuición de las personas que estiman.

Problemas:

- Son dependientes de tener a una persona con experiencia disponible
- Casi omiten el tamaño y van directo al esfuerzo como la variable más representativa de lo que tengo que construir. Normalmente se mide en horas humanas.
- "Estomacales", informales.

Métodos:

- Juicio experto ("cinco dedos oscilantes"). Basada en la experiencia de una o más personas expertas, con un juicio oscilante, que da un rango.

- PERT (Program Evaluation and Review Technique): Un juicio experto más elaborado.
Trabaja con el más probable, el pesimista y el optimista. Pondera: $(4x \text{ medio} + 1x \text{ optimistas} + 1x \text{ pesimistas})/6$.
- "Democratización de juicio experto"
 - Wideband Delphi: Estimación estomacal pero consensuada. Siguiendo un proceso definido se va refinando la estimación consensuada por los expertos. "Democratiza" la estimación, dándole cabida a muchas visiones sobre el esfuerzo que demanda una actividad y consolidarlas.
 - Planning Poker: Se basa en Wideband Delphi y va haciendo sucesivas refinaciones. (ver [detalle](#))

Métodos paramétricos

Estiman TAMAÑO.

Ventajas:

- No dependen de una persona sino que dependen de un proceso y de una historia.
- Para ser útiles, los métodos paramétricos **REQUIEREN una historia de ejecución** que me permita comparar.
 - Uso un conjunto de pasos para transformar una medición que yo tengo (parámetro) en otra cosa (un esfuerzo)
- Tienen la capacidad de estimar tamaño. Llegan a una unidad de medida de tamaño que depende del método usado.
- Son transparentes y comunicables.

Los parámetros a utilizar inicialmente son las unidades de medida de cada método.

Originalmente se hacía en Lines of Code, pero eso depende 100% de la tecnología. Otras medidas son independientes de la tecnología.

Métodos

- Function points (ver [paper](#))
- Use case points (ver [paper](#))
- Story points: otra medida, muy dependiente de la redacción de la user story en lenguaje natural
- Object points; Se parte del hecho de que conozco el software, lo tengo descripto y sé sus componentes. Cuando me llega un pedido de cambio, cuento en cuántos objetos impacta ese cambio. Es usado principalmente para el mantenimiento de un software que ya se conoce y se entiende dónde impacta todo.

Notas

- Resumen en 1 minuto: Lo importante es que se cuentan cosas, se meten variables de contexto, y así se convierte en tamaño. Ese tamaño en base a la historia/experiencia se convierte en esfuerzo. Cada uno usa su método en el proyecto en el que está. La misma técnica te da el tamaño de conversión a horas. (20.20h)
- El objetivo final de conocer esto es armarse un método de estimación propio.
- Tienen inputs llamados “parámetros” que nosotros procesamos, le agregamos complejidad y con eso definimos el tamaño de lo que tenemos que estimar. Depende del proceso.

Notas generales sobre métodos

- La estimación es el proceso completo. **Los métodos rudimentarios no estiman tamaño sino esfuerzo, se enmarcan en algo más grande.**
- **Una estimación inicial debe ser un rango**, ya que uno se encuentra al principio del cono de incertidumbre
 - Hay mucha incertidumbre al inicio. Se debe reducir lo más posible, pero igual no se puede eliminar, por ende, se usa un rango.

Método: Planning Poker → TÉCNICA PARA LLEGAR AL ESFUERZO, no para estimar completo

- Es una variación del método Wideband Delphi, con métricas similares a los métodos sofisticados.
- Se estiman tamaños de tareas relativos entre sí (tomando alguna como modelo). La estimación en Story Points (SP) puede variar entre proyectos, pero el tiempo "no".
- Muy buena para proyectos ongoing o desarrollo incremental a largo plazo. No tanto para primeras estimaciones o desarrollos de 0-100% en corto tiempo
- Es un método iterativo.

Como WD, democratiza la estimación. Originalmente, PP estimaba tamaño (en unidad de medida: story points), pero **hoy en día se usa para estimar esfuerzo**. No buscaba estimar esfuerzo, pero se usa para eso.

Las estimaciones siguen siendo estomacales de un individuo, y se ponen en conjunto al resto del equipo para llegar a un consenso.

Reglas

- Participantes: Todo el equipo tiene que participar de la estimación. Compromiso.
- Timebox: se establece un límite de tiempo para la estimación y se cumple (1h generalmente).
- Priorizar: antes del planning poker, se tienen que haber priorizado y leído todas las user stories (requerimientos), para conocer lo que se está estimando.
- Escala: se miden las user stories en **Story Points**, usando una sucesión de Fibonacci modificada: 0, 1/2, 1, 2, 3, 5, 8, 13, 20, 40, 100, infinito, ?, (café). Existen otras escalas.
- **Tarea pivoté:** se define una user story pivoté y se le da un tamaño en story points.
 - 5 u 8 SP son buenos candidatos
 - Puede ser una User story actual o una anterior que sea significativa y todos la conocen o un caso general. Pueden ser varias de diferentes tamaños.
- Velocity: se define la velocidad del equipo para saber el mínimo de SP que necesito estimar para completar un sprint
- Definición de “terminado”. Dar definición de “terminado” de una user story. Tiene que estar claro y ser conocido por todos (testeadó? en producción? uat? etc)

Dinámica

Es un proceso iterativo, hasta que se llegue a la cantidad de [SP] que marca la velocity para un sprint:

1. El Moderador plantea la User Story a todo el equipo. La describe, indica el criterio de completado.
2. Cada integrante propone una estimación sin saber la de los demás (para evitar el efecto ancla).
3. Una vez que todos tienen una estimación, dan vuelta las cartas todos al mismo tiempo
4. El mayor y el menor, dan explicaciones y discuten entre todos (2' máx)
5. Se vuelven a mostrar las cartas por si alguno quiere cambiar su estimación.
6. No se recomienda hacer más de 2 rondas de estimación para la misma User Story
7. Se anota el ID de la User Story en el pizarrón en la columna de los SP estimados.
8. Se continúa con la siguiente User Story.

Finalización

- Al finalizar la estimación, se pueden visualizar todas las tareas y las estimaciones juntas y comparar entre ellas para ajustarlas.
- Teniendo en cuenta la velocidad del equipo y las prioridades, se planifica el siguiente Sprint.
- Ventajas de la técnica
 - Como se estima en tamaño, si cambia el equipo, las tareas no se modifican sino que se modifica la velocidad de cada sprint. Feriados, exámenes, ausencias programadas bajan la velocidad de ejecución.
 - Al ser corto el sprint, se corrigen los desvíos cada 2 semanas (promedio).
 - Se pierde poco tiempo en la planning

Tool: <https://planningpokeronline.com/>

Notas

- En Scrum: se estima antes de la planning en teoría, pero se suele hacer en la planning
- En Kanban se estima cuando la tarea entra al tablero

Paper: Fundamentals of function point analysis

Function Point Analysis is a structured technique of problem solving. It is a method to break systems into smaller components, so they can be better understood and analyzed. **Function points are a unit measure for software**, much like an hour is to measuring time.

Measurement is made from a functional perspective, therefore it is **independent of technology**.

Function Point Analysis can provide a mechanism to track and monitor scope creep. Function Point Counts at the end of requirements, analysis, design, code, testing and implementation can be compared. If the count changes, scope creep has taken place.

FPA should be performed by trained personnel, using the most current version of the Function Point Counting Practices Manual.

Componentes

In Function Point Analysis, systems are divided into five large classes and general system characteristics.

- Transaccionales
 - External Inputs (EI). Componentes que cruzan el límite entre el interior del sistema y el exterior, ingresando datos.
 - External Outputs (EO). Componentes que salen del sistema (reportes, archivos)
 - External Enquiries (EQ). Contiene input y output que cruza la barrera del sistema.
- De datos
 - Internal Logical Files (ILF). Grupo identificable de información lógicamente relacionada, dentro de la aplicación.
 - External Interface Files (EIF). Grupo identificable de datos relacionados externos, que se consultan únicamente (no se modifican desde el sistema).

Pasos

1. Determinar los componentes del producto.
2. Categorizar dichos componentes según su tipo: EI, EO, EQ, ILF o EIF.
3. Asignar niveles de complejidad (high, avg, low) a cada componente.
 - a. Transaccionales: según la cantidad de archivos que se actualizan o referencian, y la cantidad de tipos de datos.
 - b. De datos: se basa en la cantidad de tipos de registros y datos.
4. Calcular Function Points NO ajustados (UFP). Cada tipo de componente con su nivel tiene un puntaje de FP y se suman todos los puntajes de todos los componentes.
5. Calcular el factor de ajuste de valor (VAF). El ajuste se basa en 14 general system characteristics que afectan la funcionalidad general. Se ve si están presentes y se le da un puntaje de 0 (*not present*) a 5 (*strong influence*) a cada una. Con eso se aplica una fórmula de ajuste. Algunas de las 14 características tomadas en cuenta:
 - a. Performance
 - b. Heavily used configuration
 - c. Transaction rate
 - d. Complex processing
 - e. Reusability
 - f. ...

6. Obtener los Function Points netos (NFP)

Notas de clase

- Function points: Es una medida que se utiliza para reflejar un tamaño, siguiendo una regla.
- El método es una función que recibe, procesa y devuelve información (se parece al DFD: Diagrama de Flujo de Datos). Abstacta las funciones que tiene el sistema de software, que permite comparar independientemente de dónde usó la técnica de medición.
 - Entrada: componentes (EI, EO, EQ, ILF, EIF)
 - Ponderación, sumatoria y ajustes por factores del entorno.
 - Salida: function points
- Es independiente a las tecnologías a utilizar.
- Lleva gran cantidad de tiempo. Es difícil de aprender y perfeccionar.
- **Útil en proyectos que ya tienen avanzado el análisis y diseño**, para poder contabilizar todo.
- Hay un International FP group que tiene tablas de conversión de tamaño a esfuerzo.

Paper: Use case points

- **Entrada: Casos de uso ya descriptos. Contempla cantidad de actores (interacciones) y de casos de uso, cada uno con un factor de peso.**
- Ajuste: A eso se lo ajusta por complejidad técnica y factores ambientales.
- Salida: Use case points (o esfuerzo: UCP * CF)

El número de Use Case Points de un producto depende de:

- La cantidad de casos de uso
- La complejidad de dichos casos, según su cantidad de transacciones
- Los actores que intervienen
- Factores técnicos y ambientales

Proceso

1. Detectar y clasificar actores
2. Clasificar casos de uso (según complejidad)
3. Hallar
 - a. Peso no ajustado de los actores (UAW)
 - b. Peso no ajustado de casos de uso (UUCW)

- c. Use Case Points no ajustados (UUCP)
- d. Factores de ajuste (Technical Complexity y Environmental)
- e. Use Case Points netos (UCP)

Ojo: Pueden cambiar mucho las estimaciones xq ante mayor detalle del use case, más transacciones (más use case points). Hay que estandarizar el nivel de detalle. La especificación debe incluir casos de uso bien definidos.

Es un método muy dependiente de cómo se describe el caso de uso en lenguaje natural, y por ende, difícil de estandarizar.

Story points (resumen ajeno)

No es un método sino una unidad de medición que se usa en los métodos.

Velocidad del equipo: Cuántos story points meto en el sprint, depende de la velocidad de mi equipo.

En un sprint meto por ejemplo una tarea de 5, una de 8 y una de 2 story points, ¿ Requerimientos no funcionales: considerarlos. Por ejemplo para cuántos alumnos armar el SIGA. Definición de terminado: tener en cuenta testing? aceptación del usuario? en producción?

Object points

- Entrada: Objetos
- Salida: Object Points (ajustados)

Objetos: pantallas, reportes y módulos (y su complejidad)

- Son objetos concretos con una tecnología concreta que ya conozco
- No está relacionado necesariamente con objetos de OOP
- El método original contempla solo estos tres tipos de objetos. Se pueden otros (stored procedures, clases, scripts SQL, etc ...)

Usado para proyectos de mantenimiento, donde ya conozco los componentes que voy a estar utilizando

Proceso

1. Se asigna a cada componente un peso de acuerdo a su clasificación por complejidad en simple, medio o difícil.
2. Se le brinda un peso a cada nivel de complejidad (teniendo directa proporción con el esfuerzo que requiere la implementación de c/u)
3. Sumando la cantidad de objetos de cada complejidad (usando sus pesos según la tabla) da como resultado los Object Points
4. Considera como factor de ajuste el porcentaje de reuso de código

Temas nombrados para el primer parcial

(clase virtual Dalceggio, Mansilla, Collazzo - 1C2024)

- Intro de classic mistakes
- Calidad de proceso, de producto
- Norma iso 25000
- Software eng approaches (proyectos, scrum, kanban, cynefin, timebox, lean)
- Estimaciones, incertidumbre, ciclo, etc. planning poker y estimacion más genérica.
Técnicas paramétricas de estimación
- No hace falta saber de memoria cómo function points. Si contabilizar
- Dif entre esfuerzo y duración. Se pasa de uno al otro aplicando reglas.

10-20 pregs, vf/choice y escenario de desarrollar un concepto.

Preg de parcial:

- Puede el esfuerzo coincidir con la duración? Sí
- La duración puede ser el triple del esfuerzo? Sí.

Los papers entran a nivel conceptual.

Classic mistakes: cuáles son los principales, cómo se aplican. Distribución de procesos, personas y

Técnica del diagrama causa-efecto

Concepto de agregar gente a un proyecto atrasado: cuándo sí, cuándo no.

Cómo aplicar good enough a estimaciones: no perder el tiempo. No usar más tiempo en estimar que en trabajar.

10 pregs, 1 punto each. No reutilizan parciales, los hacen de cero.

- VF - ojo con palabritas sueltas que cambian significado, y parciales viejos donde las rtas ya no se siguen considerando correctas hoy
- Multiple choice
- desarrollar, ejemplificar - corto!
- completar una tabla
- parciales circulando con muchas rtas incorrectas
- Preg: una métrica x puede estar asociada a x atributo

40-60 min

Atributos ISO

1. Adecuación funcional(3)
2. Eficiencia de desempeño(3)
3. Portabilidad(3)
4. Compatibilidad(2)

-
- 5. Fiabilidad (4) madurez, tolerancia a fallo, capacidad de recuperación, disponibilidad
 - 6. Usabilidad (6): adecuación, aprendizaje, uso, protección frente a errores de usr, estética, accesibilidad
 - 7. Seguridad: confidencialidad, no repudio, autenticidad, (coherencia),
 - 8. Mantenibilidad: modularidad, analizabilidad,

2o parcial

U5: Software Configuration Management

Basado en Nacho + Paper segun Isarasibar + Notas de clase 1C2024

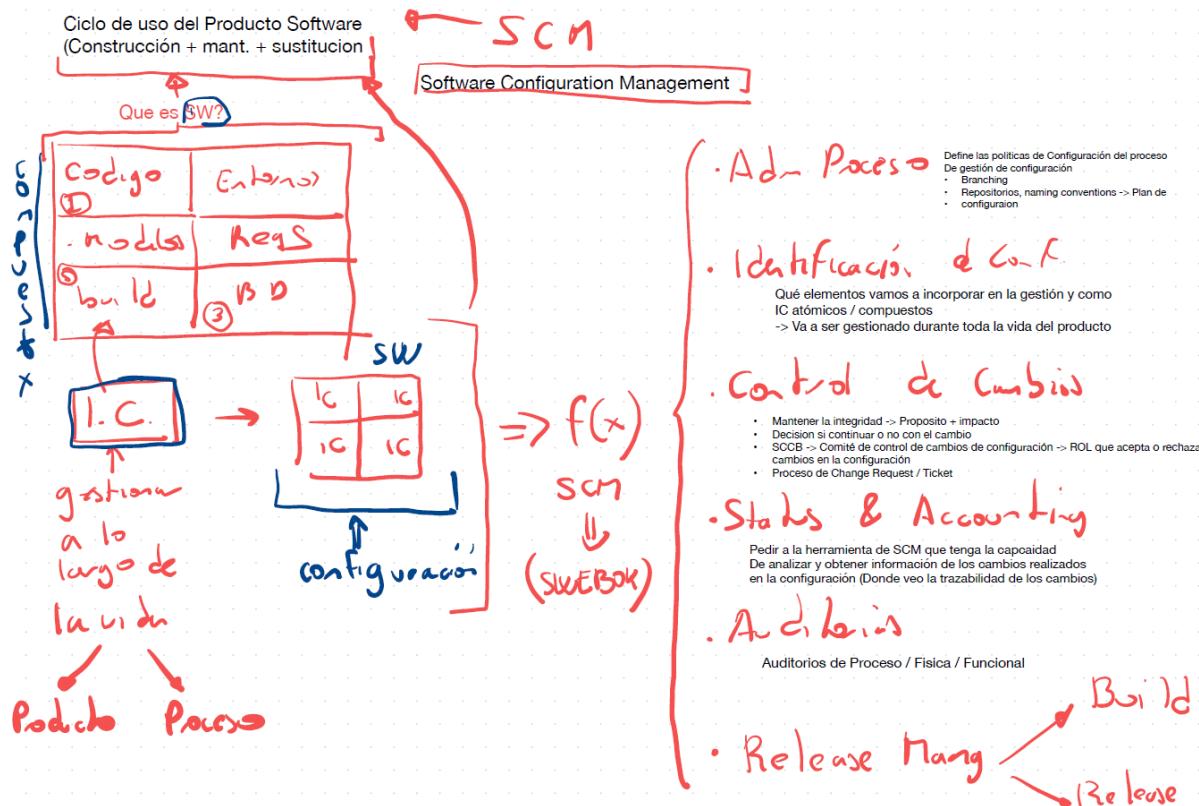
Materiales

- PPT: Tema 5
- Paper: But I only changed a line of code
- Paper (Opcional) Gitflow, Github Flow, Trunk...

Notas adicionales

- CM es Configuration Management. No confundir con Change Management!

Overview en clase



Pantallazo rápido según Diego Mansilla - 1C24 - 24/6 de 19 a 19.38hs

Gestionando los cambios al software

¿Qué es la Gestión de Configuración de software?

Informalmente: necesito ordenar el desarrollo de software de forma que pueda comprender el impacto de cómo vamos a tocar el código para generar el producto correcto con las funcionalidades correctas, en un ambiente distribuido (equipo, ambientes, etc):

Formalmente:

La Gestión de Configuración de Software (SCM) es una **disciplina** orientada a administrar la evolución de **Productos, Procesos y Ambientes**, definiendo mecanismos para administrar diferentes versiones de los mismos, **controlando los cambios efectuados**, y auditando y reportando la actividad de cambios realizada.

Su **propósito** es establecer y **mantener la integridad** de los artefactos del proyecto de software a lo largo del ciclo de vida del mismo.

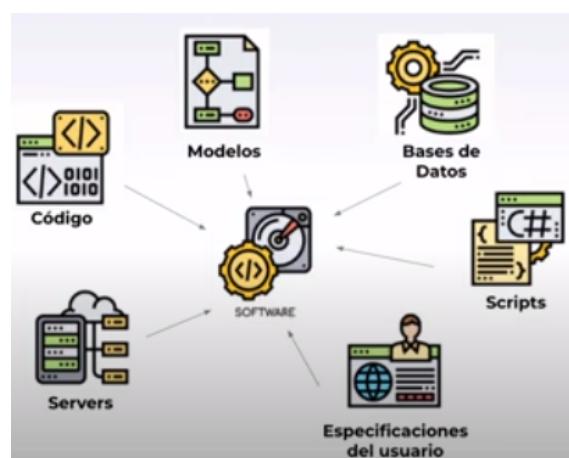
Hay 3 problemáticas:

1. Producto (código)
2. Procesos (lo que hago con el código)
3. Ambientes (dónde lo instalo)

¿Qué es el software?

El equipo de desarrollo define qué elementos incluir en la definición de su software, porque cada elemento va a ser controlado a lo largo de toda su vida (cambios que sufre, motivo, cuál es la versión que corresponde). El equipo define qué se gestiona o no. **No es únicamente el código lo que se gestiona.**

Ejemplo: Código + Contexto de ejecución (hardware) + modelos (reglas de negocio) + DB + especificaciones de usuario + scripts (de building)



Conceptos clave

Item de configuración (IC)

Un IC es cualquier elemento involucrado en el desarrollo del producto (código, servers, especificaciones de usuario, scripts, modelos, bases de datos) que está bajo el control de la gestión de configuración (SCM). Esto lo define el equipo.

Para cada ítem se conoce información/metadata sobre su configuración (nombre, versión, fecha de creación, autor, entre otros).

Existen 2 tipos de ICs:

- De proceso (impactan más en el proyecto, son sobre el proceso de desarrollo)
- De producto (exceden al proyecto)

Además existen ICs simples y compuestos.

- Simple: es atómico, no se puede subdividir.
- Compuesto: cuando tengo un grupo de ICs que no me importan individualmente (ej: framework - no me interesan las partes, sino el todo)

ICs de Proceso

Son los IC **generados por el proceso de desarrollo** y ayudan a mejorarlo. Ejemplos:

- Plan de gestión de cambios (CM, change management).
- Propuestas de cambio
- Plan de desarrollo
- Plan de Calidad
- Lista de Control de entrega
- Planes de proyecto

Algunos de estos ítems sólo son gestionados durante **una parte** de la vida del producto SW. Ej: el plan de proyecto sólo es gestionado durante el desarrollo del proyecto y luego se deja de gestionar o se elimina. Otros ítems de proceso pueden seguir siendo gestionados durante toda la vida del producto sw: ejemplo el plan de CM.

ICs de Producto

Son los ítems propios del producto software y que son gestionados durante todo el ciclo de vida del mismo (seuelen trascender temporalmente el desarrollo del proyecto puntual). Son transversales a todos los ciclos de mantenimiento del producto. Ejemplos:

- Requerimientos
- Plan de Integración
- Manual de Usuario
- Arquitectura del Software
- Estándares de codificación
- Casos de prueba
- Código fuente
- Documento de despliegue

Aseguran que se pueda seguir usando el producto de software más allá de lo que fue la gestión del desarrollo.

Configuración de software

Es el conjunto de ítems de configuración (ICs) que gestionaremos para la construcción de nuestro producto de software.

La configuración puede o no contener código, dependerá del momento de la construcción del producto de software. Ej: al inicio del proyecto, puede estar compuesta únicamente por una especificación de requerimientos de software.

Línea Base

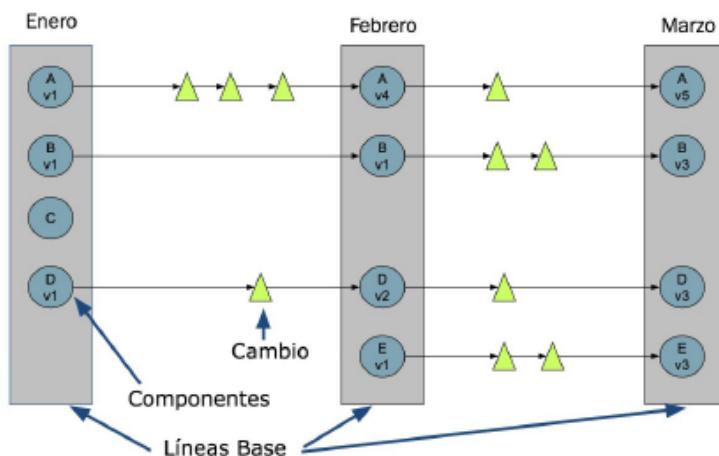
Representa un estado de la configuración de un conjunto de ítems en el ciclo de desarrollo, que puede tomarse como **punto de referencia para una siguiente etapa del ciclo**. Sirve de base de comparación.

La línea base es un momento en que la **configuración es significativa porque representa algo** (requerimientos iniciales, primera entrega, etc).

Se la usa porque se verifica que esta configuración del ítem o conjunto de ítems satisface algunos requerimientos funcionales o técnicos.

Esta línea base **puede no contener código y puede no coincidir con un release de Software.**

IEEE (IEEE Std. No. 610.12-1990) define a la línea base como “una descripción y revisión acordada de los atributos del producto, que luego sirven como base para un mayor desarrollo y definición del cambio, y este cambio solo se puede realizar a través de procedimientos formales de control de cambios”.

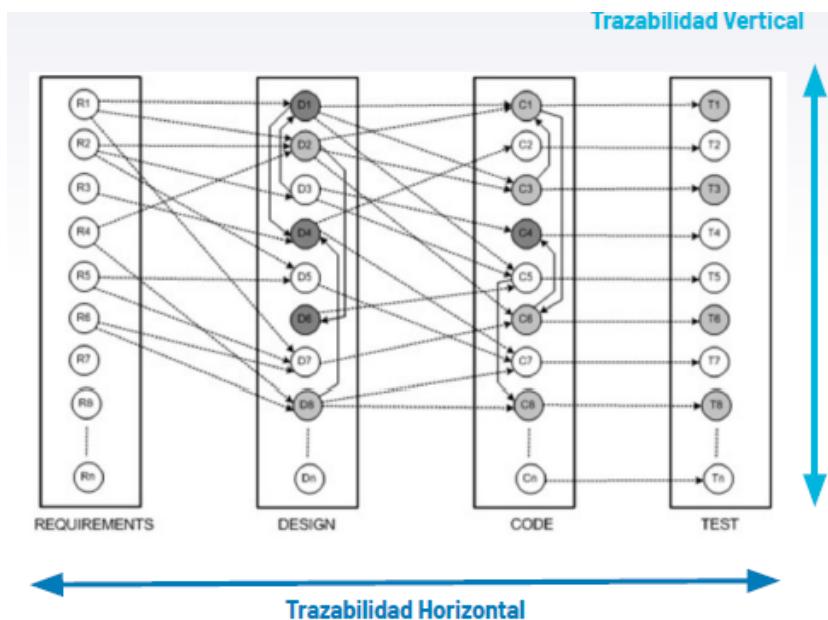


Trazabilidad

A la configuración se le sacan “fotos” para poder rastrear lo que le pasa a un IC a lo largo de los diferentes estados por los que pasa ese IC.

La trazabilidad es la capacidad de rastrear un ítem de configuración a lo largo de todo el proceso.

- Trazabilidad Horizontal: Representa la capacidad de trazar todos los IC generados en cada etapa del proceso por un mismo requerimiento. (**Puedo ver un requerimiento, dónde está implementado, en qué ambiente se puede usar**).
- Trazabilidad Vertical: Representa la capacidad de trazar todos los IC del mismo tipo, generados en la misma etapa. (En un determinado momento de mi ciclo de vida, genero una determinada de IC asociadas a un concepto puntual)



Pregunta de examen: ¿Para qué me interesa la trazabilidad?

Quiero determinar IC, configuración, baselines y trazabilidad para:

- **Mejorar la eficiencia del proceso de depuración.** Ante una falla en un baseline, con **trazabilidad horizontal** puedo encontrar con más facilidad y rapidez el defecto y la equivocación que lo causó.
 - Cuando testeo, testeo una **baseline** particular con un nombre único. Comparar 2 baselines nos sirven para saber qué cambios se introdujeron.
 - La trazabilidad **horizontal** entre ICs me sirve para encontrar más rápido el defecto y su causa. Si ante una funcionalidad que estoy probando puedo ir hacia atrás en la cadena al requerimiento original, voy a poder ejecutar de forma más eficiente en proceso de depuración.
- Consulta de trazabilidad vertical. Permite tracear multiples requerimientos sobre un mismo IC. Hace referencia a todos los IC y requerimientos que hace que una entidad usuario tenga x datos (uno o más requerimientos). Las dos trazabilidades son combinadas. (??)

Branching

Branching = acción de crear líneas de desarrollo separadas de una línea de trabajo. Permite trabajar en múltiples versiones de un producto usando el mismo set de ICs.

Tengo tantas líneas de desarrollo como features tenga. Existen diferentes estrategias de branching (dividir) para coordinar el trabajo en paralelo.

- Branch = línea de trabajo individual. Parte de un baseline de un repositorio existente.
- Branching: acción de crear líneas de desarrollo separadas.
- Merge: acción de unir las múltiples líneas de desarrollo.

Los branches son mecanismos usados para que diferentes personas puedan desarrollar nuevas funcionalidades para el software utilizando un entorno de trabajo separado del resto.

Estas líneas usan las líneas base de un repositorio existente como punto de partida. Así las features y bug fixes son desarrollados de forma separada, facilitando el trabajo en paralelo.

Permite a los miembros del equipo trabajar en múltiples versiones de un producto, utilizando el mismo set de ICs.

Tema relacionado: [Branching strategies](#)

Ambiente (Environment)

Un entorno de implementación es una **colección de servidores, clústeres y servicios configurados que colaboran para proporcionar un entorno para alojar módulos de software**.

Cada entorno se construye con un **propósito**, pudiendo tener características funcionalmente similares pero **no necesariamente ser iguales a nivel infraestructura**.

Ejemplo: los entornos de pre-producción y producción. No necesariamente son iguales a nivel infraestructura pero deberían ser equivalentes a nivel software.

Cada organización/equipo decide para qué será utilizado el ambiente y en qué etapa del proceso de desarrollo se utilizará.

Ambientes habituales: desarrollo, testing (o QA), producción (configuración mínima). Entre testing y producción puede haber staging (o acceptance). También puede haber ambiente de integración.



El significado del ambiente lo da el usuario. Puedo tener un ambiente de producción que se usa para testing. El significado se le pone en cada momento.

Cada ambiente cuesta dinero, por eso generalmente los que no son producción no van a tener la misma configuración exactamente.

Funciones de SCM (según SWEBOK)

Hay 7 funciones, que son todo lo que hay que hacer para gestionar la SCM según SWEBOK. Cada una tiene un propósito. Las últimas 3 son más prácticas.

1. Administración del Proceso SCM

- a. Contexto Organizacional de SCM
- b. Plan de SCM
- c. Vigilancia del Proceso SCM

2. Identificación de la CS

- a. Identificar ítems a ser Controlados
- b. Identificar ítems de 3ros o bibliotecas

3. Control de la CS

- a. Solicitar, evaluar y aprobar cambios al SW
- b. Implementar cambios al SW
- c. Desviaciones y excepciones del proceso

4. SC Status Accounting

- a. Información de Estado de la Gestión de CM
- b. Reportes de Estado de la Configuración

5. Auditoría de la Configuración

- a. Auditoría Funcional
- b. Auditoría Física
- c. Auditoría de Proceso

6. Release Management & Delivery

- a. Software Building
- b. Software Releasing

7. SCM Management Tools

- a. Herramientas para soportar los diferentes aspectos y niveles de la gestión de configuración

F1. Administración del proceso de Gestión de Configuración del Software (SCM)

Define el marco de trabajo de los developers.

Se ocupa de:

- Contexto Organizacional de SCM
- **Plan de SCM**
- Vigilancia del Proceso SCM

Acá se define **herramientas, reportes y procesos** que se usan para la gestión de configuración.

Define cómo se incorporan los ICs, cuándo se realizarán las líneas de base, cómo es la estrategia de desarrollo en paralelo y cómo serán nombradas las versiones.

Generalmente se hace mediante la definición de un plan de CM o Plan de Gestión de Configuración.

2. Plan de gestión de configuración (CM Plan)

Un plan de SCM define:

1. Lista los Ítems de Configuración, y en qué momento ingresan al sistema de CM
 - a. Cómo se registran cambios al software (Ej con Jira)
 - b. Roles que aprueban o rechazan al cambio (SCCB)
 - c. Están identificados los repositorios de los artefactos y hay establecida una política de trabajo sobre esos repositorios
2. Estándares de nombres, jerarquías de directorios, estándares de versionamiento
3. Estrategias de branching
4. Procedimientos para crear “builds” y “releases”
5. Reglas de uso de la herramienta de CM y el rol del administrador de la configuración
6. Define el contenido de los reportes de auditoría y los momentos en que estas se ejecutan

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

Ejemplo de proceso de SCM definido

- Está definido cómo se registran cambios de software (ej: Jira)
- Se establecieron los roles que aprueban o rechazan el cambio (quién decide si se hace o no)
- Están identificados los repositorios de los artefactos (Github, NPM).
- Hay establecida una política de trabajo sobre esos repositorios. Branching strategies, reglas de commits, nomenclaturas, sistema de carpetas, naming conventions, versionado.
 - El equipo conoce las reglas de desarrollo
- Están identificados los momentos en los que se realizan las líneas de base. Cuándo se hace deploy, qué hay en cada ambiente.
- Los cambios son gestionados completamente, desde el ingreso hasta su puesta en producción (Jira)

DISCLAIMER : Estos puntos no son todo lo que hay que tener, son una base mínima necesaria para garantizar la gestión.

Estrategias de branching

Es la estrategia que adoptan los equipos de desarrollo de software al escribir, mergear e implementar código cuando usan un sistema de control de versiones (git / SVN / TFS).

Es un conjunto de reglas que los desarrolladores pueden seguir para estipular cómo interactúan con una base de código compartida.

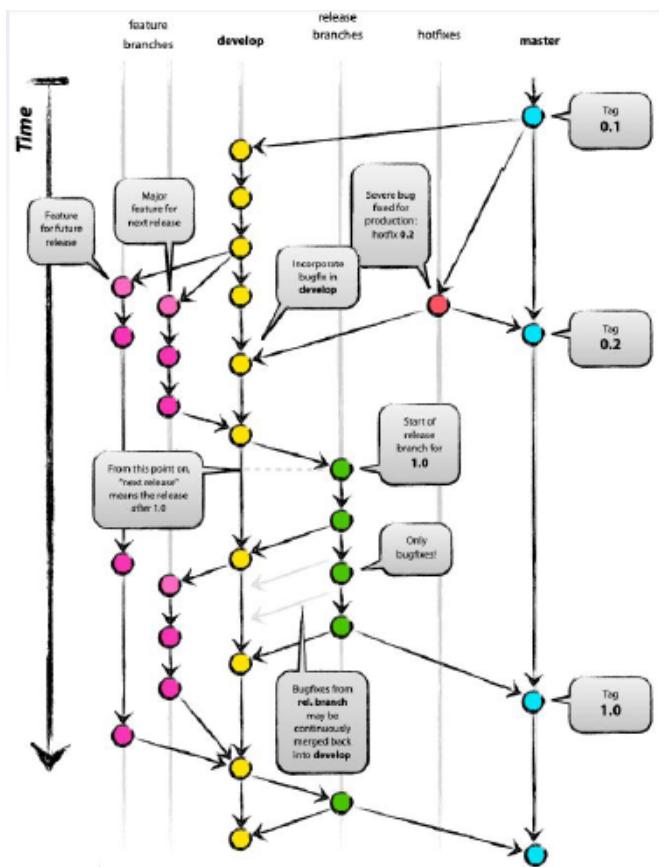
Una estrategia de branching tiene como **objetivo**:

- Mejorar la productividad al garantizar una coordinación adecuada entre los desarrolladores
- Facilitar el desarrollo paralelo
- Ayudar a organizar una serie de lanzamientos planificados y estructurados
- Trazar un camino claro al realizar cambios en el software hasta la producción
- Mantener un código libre de errores donde los desarrolladores puedan corregir problemas rápidamente y hacer que estos cambios vuelvan a la producción sin interrumpir el flujo de trabajo de desarrollo.

Gitflow

Consiste en **dos ramas principales** que existen a lo largo de todo el ciclo de vida del desarrollo, **master y develop** y tiene una serie de branches de soporte.

- En **master** vive el código productivo. Acá están las entregas, con esta rama genero mi artefacto para producción.
- En **develop** está la última versión de desarrollo. En develop están las features nuevas ya desarrolladas/terminadas. Los desarrolladores generan nuevas feature branches a partir de esta rama.
- A partir de master y develop se generan **feature branches**. Estas se desarrollan en paralelo. Acá están las features nuevas *en desarrollo*. Una feature branch no mergeada significa que la persona está trabajando en ese desarrollo, sin mirar las tareas de Jira.
- Cuando se está listo para unir múltiples cambios, se genera una **release branch** donde se testean cambios integrados.
- Cuando la release branch está ok, se genera un **merge a master**.
- Si tengo que hacer una corrección sobre master, hago una rama que se llama **hot fix** que parte de una copia de master. Un hotfix tiene que llegar a la rama de develop y luego a las feature branches.
- **Mínimo tiene 4 ramas base: master, release, develop, hotfixes. Adicionales: n feature branches.**



Pros:

- Funciona muy bien en equipos distribuidos con diferentes husos horarios. Termina convirtiéndose en la forma de comunicación del equipo.

Contras:

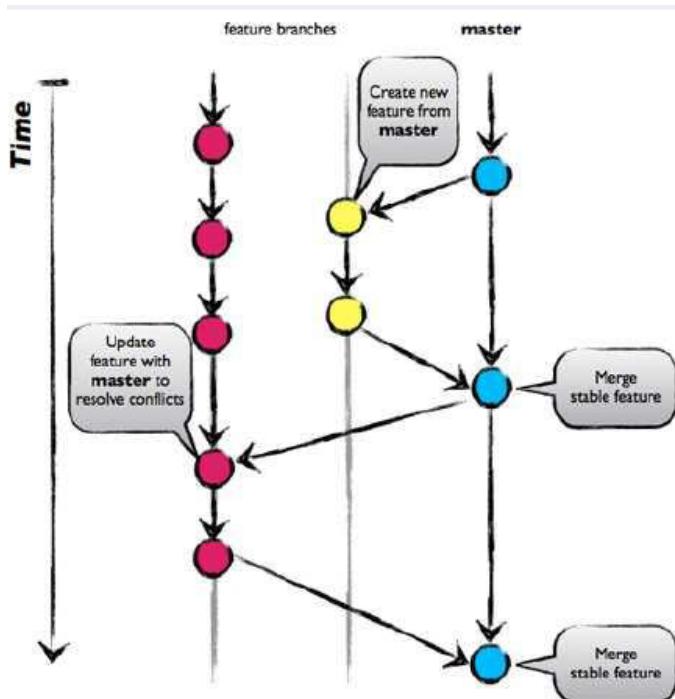
- Es muy complejo.
- Error prone.
- Difícil rastrear un problema cuando los tests de un release fallan y hay un mar de commits para revisar.
- No es ideal para equipos que quieren implementar CI y CD

Github flow

Creado por Github, es una alternativa simplificada.

- Existe una rama **master** de donde parten las ramas de desarrollo.

- Cada nueva feature tiene su rama separada, que después se integra a master cuando se termina. Una tarea en proceso está en una **feature branch**. Hay tantas feature branches como desarrollos en paralelo haya.
- Como mínimo tiene 1 branch, la master branch.
- Diferencias respecto de Gitflow:
 - No hay release branch. Todo deriva de la master branch y vuelve a ser mergeado a master.



Ideal para equipos pequeños que no tienen que manejar tantas versiones de producción.

Pros:

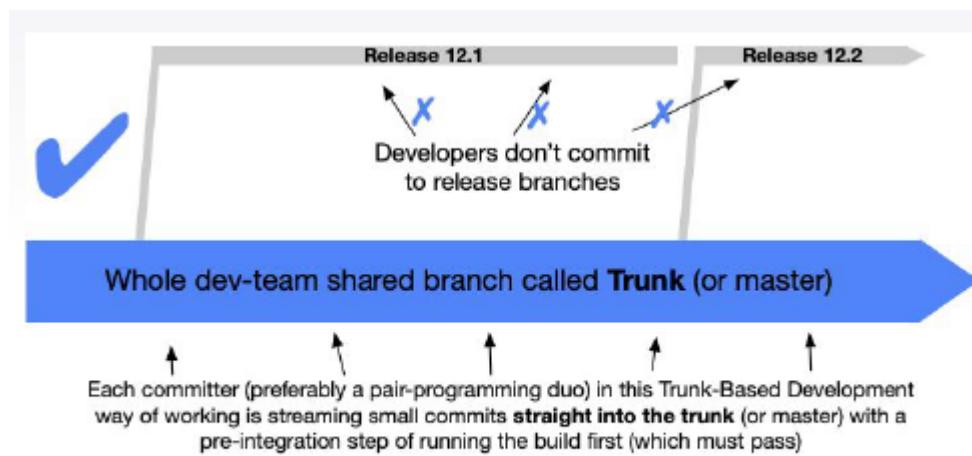
- Equipos alineados.
- Simplifica.
- Genera ciclos de producción breves y releases frecuentes.
- El código en master está constantemente en estado deployable y por ende sirve para CI y CD.

Cons:

- (1:23) Si tengo que resolver un problema rápido (hotfix), tengo que crear una feature branch en base a master/producción, pero el fix lo tengo que mergear a todas las otras feature branches.
- No sirve para manejar múltiples versiones productivas del mismo código
- Con muchos developers mergeando a la misma rama, hay poca trazabilidad y orden
- La falta de ramas de development la hace un poco

Trunk based flow (1:25:00)

- Concepto: Un modelo de branching de control de fuente, donde los desarrolladores colaboran en el código en una sola rama llamada "troncal" *, resiste cualquier presión para crear otras ramas de desarrollo de larga duración mediante el empleo de técnicas documentadas. Por lo tanto, evitan fusionar el infierno, no rompen la construcción y viven felices para siempre. Modelo original de TFS, de Microsoft (Team Foundation Server)
- Única rama: trunk/master
- Todos desarrollan y commitean directo al trunk. No hay feature branch.
 - Según el paper, cada uno commitea al meno una vez al día, entonces
- El desarrollo en paralelo no se basa en diferentes ramas
- Ventaja:
 - Sólo se necesita la línea del repositorio.
 - Es la más simple de todas
 - Facilita CI/CD sedgún paper
- Desventajas:
 - Único momento de conflicto: Si hay mucha gente desarrollando sobre el mismo componente, el primero gana, el segundo tiene que resolver conflictos de merge.
 - Cuesta resolver cuando hay un problema en producción, y detectar dónde surgió el problema y compatibilizarlo con lo que se desarrolló después de que se incorporó el problema. Requiere un proceso de testing adecuado para esto ->
- Dónde usarlo: en equipos muy chicos. Es fácil de gestionar.
 - No es ideal en equipos distribuidos



F2: Identificación de la configuración

Establece cómo está compuesto el Software.

Se ocupa de:

- Identificar ítems a ser Controlados
- Identificar ítems de 3ros o bibliotecas

Es la actividad de **definir qué elementos (ICs) estarán controlados por la gestión de configuración**. No todos los artefactos necesitan ser gestionados para garantizar la integridad del producto.

La actividad **establece guías o criterios** para entender qué elementos son parte o no de una configuración.

La identificación de la configuración también **define momentos o condiciones para establecer una línea de base o bien para liberarla** (es decir llevarla a otro ambiente, también conocido como "release").

1. Identificación de la configuración

Identifica los ICs que serán gestionados. Para ello, se pueden ejecutar distintas acciones, como por ejemplo:

- Identificar Código

- Buscar fuentes
- Chequear repositorios
- Entender la arquitectura
- Identificar ambientes de ejecución
 - Tratar de instalar una versión
 - Identificar Bases de datos, servers, configuraciones
- Identificar Documentación
 - Buscar especificaciones
 - Entender cambios anteriores

Qué elementos se registran de un IC

En general, es importante identificar para cada IC los siguientes atributos:

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

Estos atributos serán registrados y se utilizarán como información útil en la actividad de Status & Accounting de la configuración.

F3: Control de Cambios de la configuración

Segundo punto. El más crítico. Centro neurálgico del SCM.

Establece reglas para incorporar (o no) **cambios solicitados** al software.

Se ocupa de:

1. Solicitar, evaluar y aprobar cambios al SW
2. Implementar cambios al SW
3. Desviaciones y excepciones del proceso

El Control de Cambios de la Configuración asegura que los IC 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

El Control de Cambios de la Configuración consiste en establecer un **procedimiento/proceso de control de cambios** y controlar el cambio y la liberación de ICs a lo largo del ciclo de vida.

Software Configuration Control Board (SCCB)

Comité de control de cambios en un software. Este conjunto de personas o roles reciben el cambio, lo evalúan y, si se aprueba, planifican cómo y cuándo se hace el cambio.

- Es quién tiene la autoridad de aceptar o rechazar un cambio en la configuración de Software
- En proyectos pequeños puede residir en el Líder del proyecto, y a medida que se incrementa la complejidad, puede ir incorporando nuevos integrantes
- Puede haber diversos niveles de autorización de acuerdo a diferentes criterios
 - Por ejemplo: Un nivel puede ser compuesto solo por Stakeholders para autorizar o no un cambio de alcance, mientras que en otro nivel el Product Owner acepta o no un cambio en el backlog del sprint

Busca prevenir que surjan situaciones y cambios informales: que un dev defina algo sin que el líder de proyecto esté enterado.

Detalles:

- El Board puede ser una o más personas.
- Que se acepte un cambio no significa que se inicie la implementación del cambio inmediatamente
- **El propósito es que los cambios que ingresen, lo hagan de cambio ordenada y controlada.**

Change Management Process

Es el flujo y las herramientas definidas para identificar, analizar y aprobar cambios a una configuración. Difiere en cada Organización, pero en general se puede dividir en las siguientes etapas:

1. Identificar el cambio

Incluye formalizar el pedido de cambio. En general sólo se aceptan cambios de personas autorizadas a pedirlo.

2. Evaluar el impacto

Analizar el cambio y determinar el impacto que tendrá en el software, en el calendario, en el equipo que trabaja en él y quienes lo usarán. Evalúa los aspectos técnicos del cambio, sus posibles efectos secundarios y el impacto general en otras partes del sistema.

3. Decidir hacer o no el cambio

Entendiendo el impacto en el negocio, o en el calendario o en el equipo.

Si se rechaza, se puede dejar para otra fase del desarrollo.

4. Planificar la implementación del cambio

Incorporarlo dentro de las tareas a realizar, incorporándolo al calendario y asignando recursos para resolverlo.

Según SWEBOK, cap. 7 - Notar que existe un "camino de emergencia"

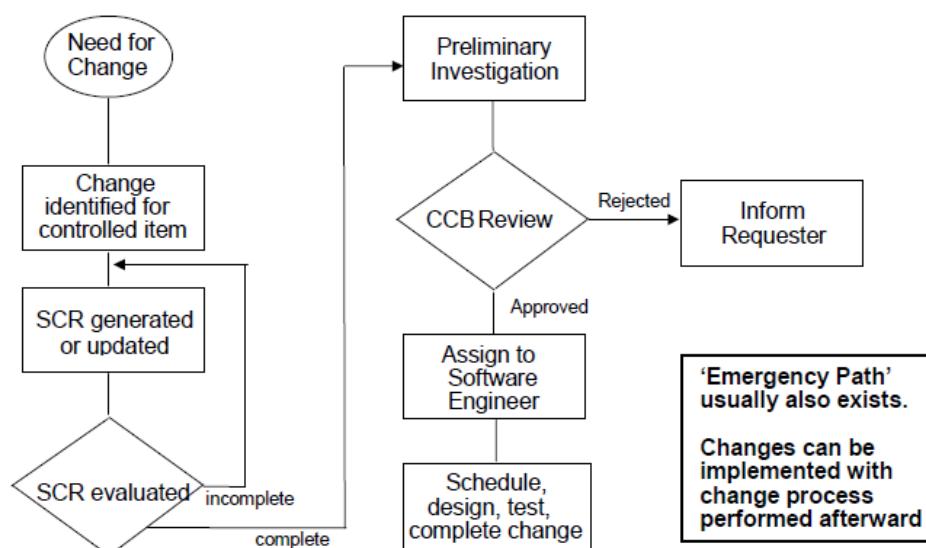


Figure 5 Flow of a Change Control Process

Mediciones relacionadas a Change Request

1. **Successful Changes:** Cantidad de cambios que han sido aprobados y desplegados en producción sin generar incidentes.

Contraparte: *unsuccessful changes*. Nos da información sobre la efectividad del proceso de testing.

2. **Changes in Backlog:** Cantidad de cambios pendientes de ser analizados. Es un indicio de la eficiencia del proceso de aprobación de cambios y se puede utilizar para evaluar la velocidad del equipo.
3. **Emergency Changes:** Cantidad de cambios que han sido aprobados por canales de urgencia o emergencia.
Información: nos indica si hay hotfixes necesarios. Puede tener que ver con cambios en ambientes con integraciones a servicios externos que cambian, por ejemplo.

F4: Status Accounting de la Configuración

Analiza la historia de los cambios realizados al software. TRAZABILIDAD.

Se ocupa de:

- Información de Estado de la Gestión de CM
- Reportes de Estado de la Configuración

Status Accounting tiene la función de registrar y reportar la información necesaria para administrar la configuración de manera efectiva.

(Es una capacidad que le pido a mi herramienta de gestión de configuración. La capacidad debe poder responder a todas estas preguntas.)

Para ello debe:

- Listar los ICs aprobados
- Mostrar el estado de los cambios que fueron aprobados
- Reportar la trazabilidad de todos los cambios efectuados al baseline

Una implementación exitosa de esta actividad debe poder contestar:

- ¿Qué cambios se realizaron al sistema?
- ¿Cuándo se realizaron dichos cambios?
- ¿Quién lo cambió?
- ¿Qué cambió? → alcance
- ¿Quién aprobó el cambio?
- ¿Quién solicitó el cambio?

Me permite entender todo lo que le sucedió a un IC, para que en el caso de un incidente yo pueda entender vía la trazabilidad de dónde proviene un cambio.

F5: Auditorías de la Configuración de Software

Valida los distintos aspectos de la Gestión de Configuración de Software.

Realiza 3 tipos de auditorías:

1. Auditoría Funcional
2. Auditoría Física
3. Auditoría de Proceso

Una auditoría de Configuración de Software es una examinación de un trabajo, producto o set de artefactos para evaluar aspectos técnicos, de seguridad, legales y de cumplimiento de normas con respecto a especificaciones, estándares, acuerdos contractuales u otros criterios.

Las auditorías pueden ser ejecutadas con diferentes niveles de formalidad (desde Revisiones informales basadas en checklists hasta Pruebas exhaustivas de la configuración que son planificadas con anticipación).

Tipos de Auditorías

1. Auditoria Funcional:
 - Es una evaluación del producto software que se realiza para verificar, vía testing, inspección, demostración o análisis de resultados, **que el producto ha cumplido los requerimientos especificados en la línea base de documentación funcional.**
 - En las auditorías funcionales, básicamente se verifica que una configuración dada **cumpla con alguna especificación de requerimientos** previamente definida.
2. Auditoría Física:
 - **Evaluá que los elementos modificados de la configuración sean consistentes con las especificaciones técnicas de los cambios en la versión.**
 - Esta auditoría también evalúa los cambios realizados en el código y en los diferentes ambientes
3. Auditoría de Proceso:
 - Evalúa que los cambios realizados al producto software **hayan seguido el proceso definido para los mismos.**

- El objetivo es confirmar que todos los cambios fueron solicitados, aprobados e implementados de acuerdo al proceso definido en la compañía.
- Ejemplo: Al identificar una línea base se identifica su composición (qué ítems la conforman). Las inconsistencias se reportan y se determina si el defecto corresponde a la configuración misma o a la documentación.

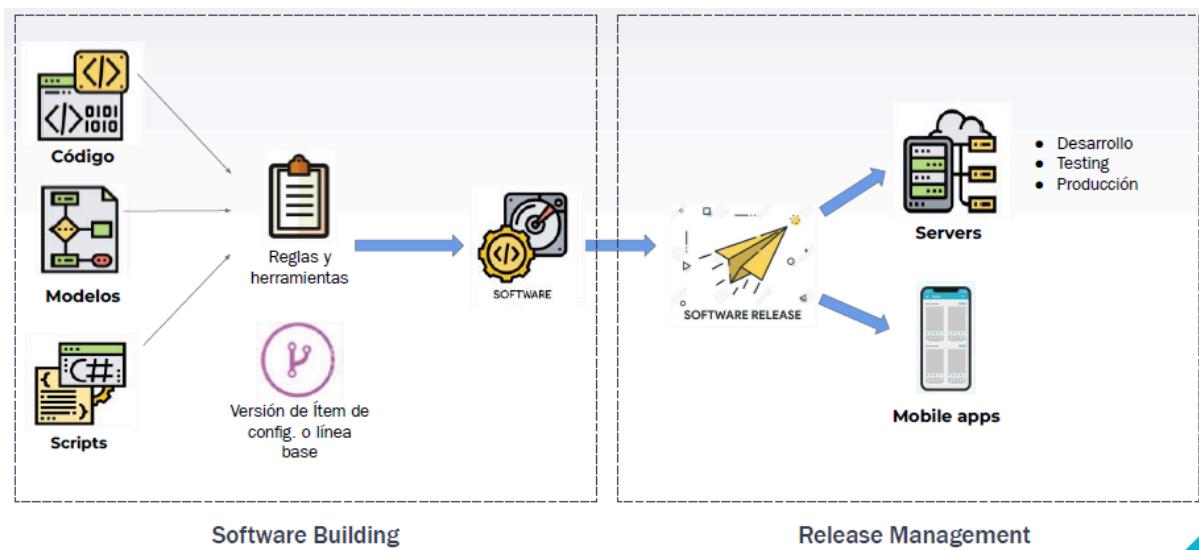
Nota: Las herramientas y testeos usados pueden ser las mismas que para testing, pero se usan sobre el ambiente de producción.

F6: Release Management & Delivery

Se ocupa del build y el release del software.

Se divide en 2 partes:

1. Software Building
2. Release Management



Release Management consists of ensuring the successful construction of the software package, based on the required ICs for the functionality to be delivered, and then releasing it in a controlled manner to other environments (tests, production, final user, etc.). It also includes the administration, identification, and distribution of a software product.

1. Software Building

Es la construcción del paquete de software que será distribuido, usando las versiones correctas de los ítems de Configuración y las herramientas apropiadas para construirlo.

Conceptos clave de Software Building:

- Build + tipos de build
- Integración continua (CI)

Build

Un Build es el proceso de convertir archivos de código fuente en artefactos de software independientes que se pueden ejecutar en un entorno. (El build es el software ejecutable en un entorno)

Tipos de Build:

1. Local builds: El developer lo realiza localmente en su entorno de desarrollo, corre Pruebas Unitarias (UT)
2. Integration builds: su objetivo es generar el entorno completo para pruebas de integración
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 administrador decide crear una nueva versión a ser liberada, o por el mismo sistema de integración si se utiliza el modo de deployment contínuo

Integración Continua (CI)

Es una práctica de desarrollo de software mediante la cual los desarrolladores combinan los cambios en el código en un repositorio central de forma periódica, tras lo cual se ejecutan versiones y pruebas automáticas.

Conlleva un **componente de automatización** (ej., CI o servicio de versiones) y un **componente cultural** (ej., aprender a integrar con frecuencia).

Es una práctica requerida para poder realizar entregas continuas (Continuous Delivery).

Prácticas

- Disponer de un repositorio de código único.

- Build
 - Automatizado mediante scripts o herramientas
 - Testable automáticamente.
 - Debe ser rápido
 - Se realiza mediante una herramienta de integración tras un commit - no lo hace el dev
- Se prueba en un clon de producción
- Cualquiera debe obtener fácilmente la última versión del ejecutable
- Todos deben poder ver qué pasa con el proceso de integración (transparencia)
- Automatizar el proceso de despliegue en el entorno de testing (deployment)

Responsabilidades del equipo

- 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 de la CI

- Detección temprana y mejorada de errores, y métricas que le permiten abordar los errores a tiempo, a veces tras solo unos minutos de la incorporación
- Progreso continuo para mejorar el feedback
- Mejor colaboración en equipo; todos los miembros del equipo pueden cambiar el código, integrar el sistema y determinar rápidamente los conflictos con otras partes del software
- Integración mejorada del sistema, lo que reduce las sorpresas al final del ciclo de vida de desarrollo del software
- Menor número de errores durante las pruebas del sistema
- Sistemas actualizados constantemente en los que realizar las pruebas
- **Fin del “en mi máquina funciona”**

Pregunta de parcial: Que no haya fallas en la IC, ¿garantiza que no haya fallas en el software?

No. Garantiza que al menos un set de tests automáticos no rompen, pero es sólo una base. No puede garantizar que no haya situaciones no contempladas en las pruebas escritas que no fallen.

2. Software Release Management

El Software Release Management gestiona la identificación (Identification), ensamblado (packaging) y la entrega (Delivery / Deployment) de los elementos de un producto Software (ej: un ejecutable, documentación, notas de lanzamiento o datos de configuración)

Define también el proceso de planificación, calendarización y gestión del software construido a lo largo de las etapas de desarrollo, testing, despliegue y soporte productivo.

Definiciones

- **Versión:** Es una instancia de un sistema que es funcionalmente distinta en algún aspecto de otras instancias.
 - **Variante:** Una instancia de un sistema que es funcionalmente igual a otras instancias, pero **difiere a niveles no funcionales** (Ejemplo: Mismo producto para Version Linux / Version Windows)
 - **Release: distribución del Software fuera del entorno de desarrollo.**
-
- **Semantic Versioning:** Es una propuesta de un set de reglas y requerimientos que dictaminan cómo los números de versión son asignados e incrementados.
Una regla muy frecuente de versionado es:
 - Dado un número de versión MAJOR.MINOR.PATCH, se incrementa:
 - El número de versión MAJOR cuando se realizan cambios incompatibles en la API o SW
 - El número de versión MINOR cuando se agrega funcionalidad de una manera compatible con versiones previas
 - El número de versión PATCH cuando se arreglan bugs de forma compatible con versiones anteriores
 - Se pueden agregar etiquetas adicionales para pre-releases o metadatos de build como extensiones al formato MAJOR.MINOR.PATCH.

Una baseline puede no coincidir con el release. Pero un release sí va a coincidir con un baseline. ¿Por qué?

La línea base refleja una configuración determinada que para mí es significativa, pero no es necesariamente distribuida en un release (por ej: momento histórico el primer archivo creado). Los baselines también se nomenclan, pero el baseline no necesariamente termina siendo un release, porque es algo que puedo no distribuir fuera del entorno de desarrollo.

Seguramente un release (una foto de una configuración particular que se está distribuyendo por fuera del ambiente de desarrollo) coincide con un baseline.

La versión del IC, del baseline y del release no tienen por qué coincidir entre sí.

Consideraciones para release management

Una vez generada la versión o release debemos buscar el mecanismo más efectivo para hacer despliegue/deploy controlado a los distintos usuarios. A estos mecanismos se los denomina [Deployment Strategies](#).

Aspectos a evaluar para realizar un release, con el **objetivo de evitar el downtime**:

- Qué usuarios deberán recibir los cambios (geográfico, premium, por segmento)
- En qué forma se deberá hacer el despliegue
- Entornos por los que deberá pasar (testing, staging, producción, otros)
- Procesos de Roll Forward
- Procesos de Roll Back
- Validación de despliegue correcto / incorrecto
- Riesgos del despliegue y cómo se minimizan
- Aprobación por el negocio y/o área de QA
- Quiénes realizarán el deployment de la versión definida

Continuous Delivery (CD)= Entrega continua

Es un conjunto de prácticas que permiten asegurar que el software puede ser desplegado en producción rápidamente y en cualquier momento, aplicando el concepto de [Pipelines](#).

Los artefactos resultantes del proceso de construcción incluyen código ejecutable y bibliotecas, que luego se puede combinar en una instalación paquete e implementado en un entorno para verificación o uso de producción.

Esto se logra haciendo que cada cambio llegue 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 sólo apretar un botón (se instala automáticamente, pero con autorización/aprobación previa de un humano).

Etapas habituales de las pipelines:

- Testing and QA del Software Construido. Ejecuta las pruebas automáticas
- Change Management, Governance, and Approvals: Contiene las actividades de aprobación de los cambios a desplegar
- Deployment Strategies. Define la estrategia con la que el Software será desplegado en el ambiente
- Verification: Define las actividades para verificar que el software fue correctamente desplegado
- Rollback : Define las acciones para volver al estado anterior, en el caso de que falle la verificación

Continuous Deployment/Despliegue continuo

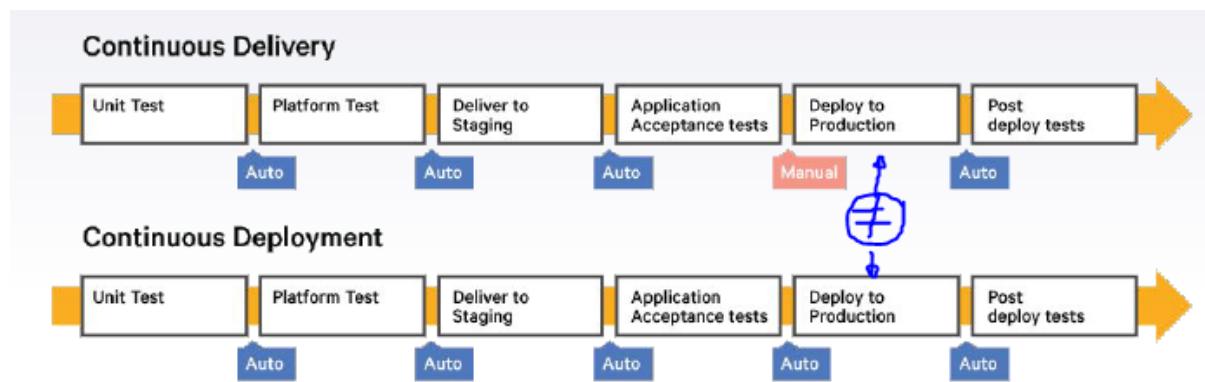
Es el paso siguiente al Continuous Delivery, en donde **también el despliegue a producción se realiza en forma automática** por un proceso y no por personas.

Comparación CI/CD/CD

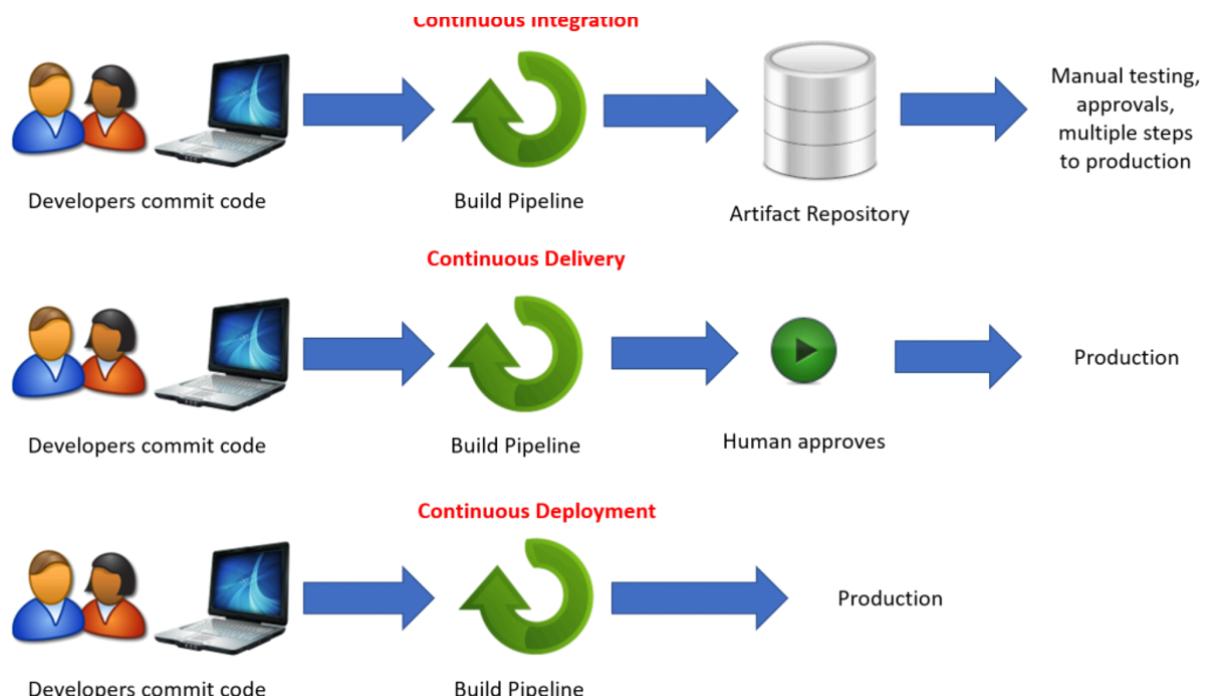
Continuous Integration (CI)	<	Continuous Delivery (CD)	<	Continuous Deployment (CD)
Integración continua		Entrega continua		Despliegue continuo
<p>Proceso de desarrollo en el que los devs integran el código nuevo con mayor frecuencia a lo largo del ciclo de desarrollo, y lo añaden a la base de código al menos una vez al día.</p> <p>En cada iteración de compilación se realizan pruebas automáticas para identificar rápidamente los problemas de integración. Hace más fácil la solución de problemas, y así evita problemas en la construcción final.</p>		<p>Retoma el proceso de integración continua y automatiza la entrega de aplicaciones a los entornos seleccionados.</p> <p>La CD se centra en entregar cualquier cambio validado de la base de código a los usuarios de la forma más rápida y segura posible.</p> <p>Automatiza el envío de los cambios a diferentes entornos (desarrollo, pruebas y producción).</p> <p>Para su envío a producción, requiere la aprobación manual.</p>		<p>Retoma el proceso de entrega continua y automatiza la publicación al entorno de producción.</p> <p>Esta automatización se basa en una serie de pruebas predefinidas. Una vez que las nuevas actualizaciones pasan esas pruebas, el sistema envía las actualizaciones directamente a los usuarios del software.</p>

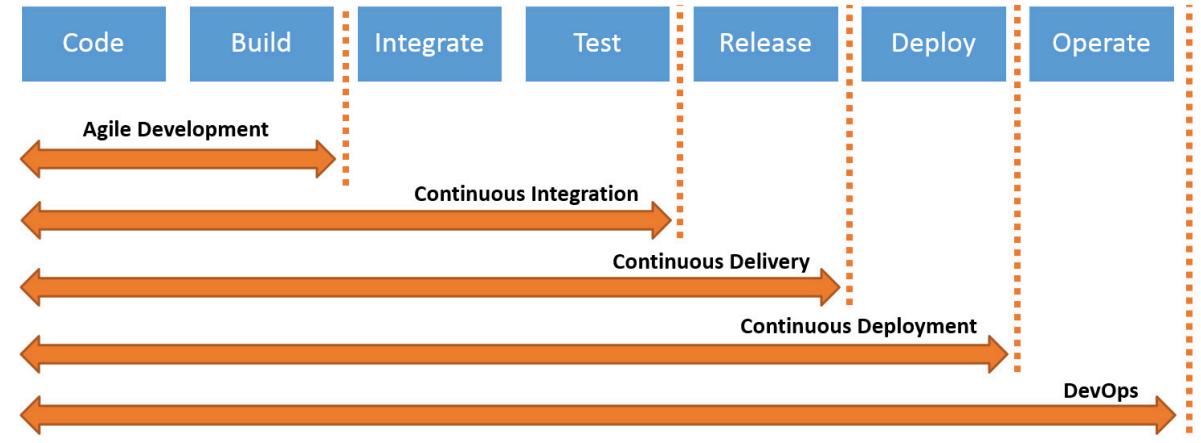
CD vs CD

En Continuous Deployment, la aprobación del release es automática, mientras que en Continuous Delivery, la aprobación es humana. En ambos la instalación en producción después de esa aprobación es automática.



Comparaciones misc de internet





Deployment Strategies

Un deployment strategy es un mecanismo de cambiar o actualizar una aplicación de Software.

- Objetivo: realizar el cambio sin generar un downtime, sin que los usuarios finales noten el cambio de la aplicación.

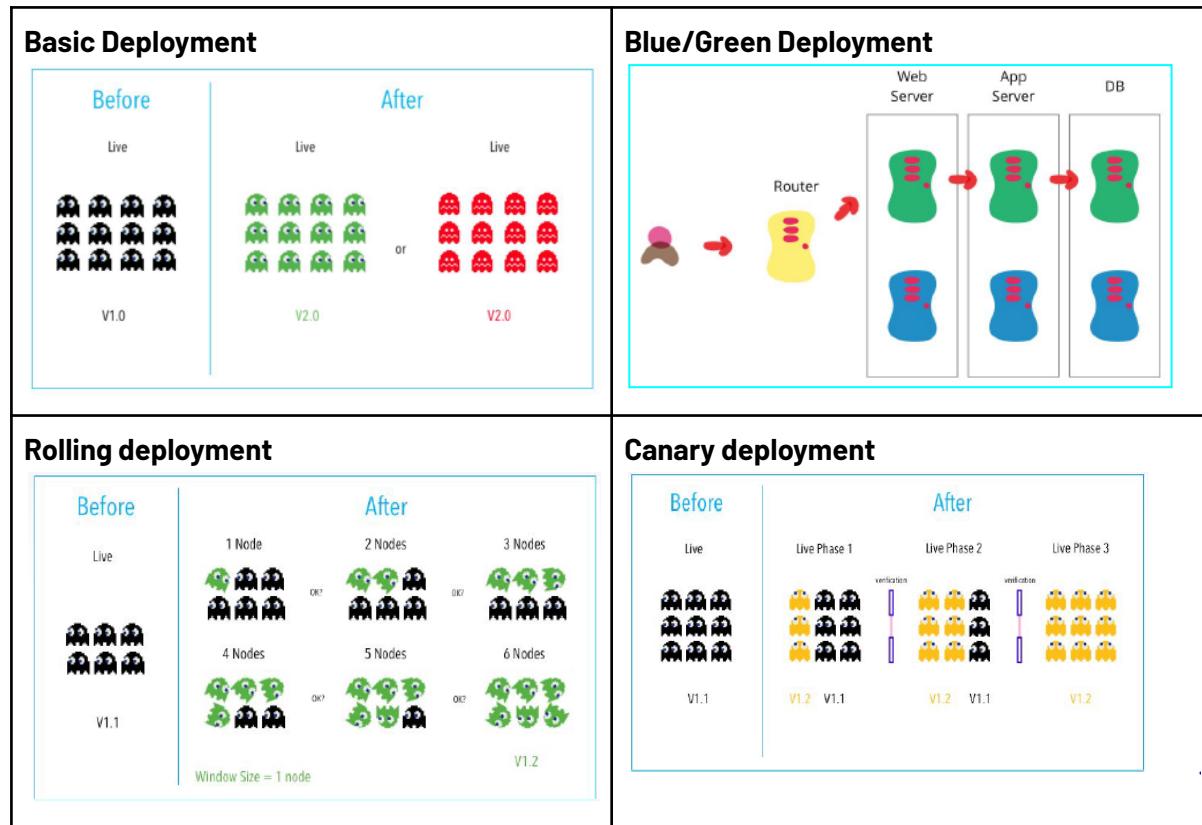
Estrategias existentes:

- **Basic Deployment:**
 - Todos los nodos de la aplicación son actualizados en el mismo momento con la nueva versión.
 - El tiempo de deployment es el que tarda en traspasarse el build.
 - Ventajas:
 - Muy simple de usar, no requiere usar mecanismos específicos
 - Desventajas
 - Riesgo alto de downtime. O sale 100% bien, o 100% mal.
- **Blue / Green Deployment**
 - Concepto: Disponemos de dos ambientes lo más similares posible, uno "green" y el otro "blue". El despliegue se realiza sobre uno de los ambientes, mientras el otro contiene la versión anterior. Un balanceador selecciona a qué grupo de servers utilizar como productivo
 - Ventajas
 - Simple de implementar
 - Desventajas
 - En organizaciones grandes, se torna una solución costosa
- **Rolling Deployment**

- Concepto: Los nodos de un ambiente son actualizados 1 a 1(o en lotes) con la nueva versión del software. Si algo falla, el roll back se hace en cada nodo, **sin downtime** porque se usan los otros nodos mientras tanto.
- Implementation: In this process, updates happen one node at a time, often managed by a load balancer, which gradually shifts traffic from the old version to the new one. This ensures a zero-downtime deployment, maintaining functionality and a positive user experience. Rolling deployment is ideal for applications that require high availability and minimal disruption during updates.
- Ventajas
 - Rollout gradual con un incremento de tráfico controlado. Simple de hacer roll back
- Desventajas
 - Dificultad técnica en el desarrollo: las versiones nueva y vieja deben poder utilizar la misma BD, para mantener la consistencia aunque se esté trabajando con dos versiones distintas del software en producción.

● **Canary Deployment**

- Concepto: se despliega el cambio en un set controlado de nodos y se va chequeando el comportamiento.
- Process: Canary deployment involves gradually introducing the new release, often controlled by feature flags, to a small group of real users. This initial phase aims to test functionality and gather user feedback. Based on metrics and user experience data, the rollout can be expanded to more users or rolled back if issues arise. This strategy is particularly effective for applications with significant user bases, where user impact and feedback are critical metrics.
- Es actualmente una de las formas más comunes de hacer deploy
- Ventajas
 - Despliegue en pequeñas fases (ej: 2% de los nodos, 10%, 50%, 100%)
 - En el caso de fallas, se quitan los nodos que fueron incorporándose
 - Solo existe 1 ambiente de producción
- Desventajas
 - Es complejo de testear producción.
 - Herramientas de monitoreo complejas para entender que está desplegado



Release Management Metrics

Métricas asociadas a los flujos de building y de releasing del software.

Métricas relacionadas a deployment:

1. **Deployment Frequency:** Cada cuanto tiempo una organización despliega software a Producción satisfactoriamente
 - a. Medición: Cant. Deployments por día
 - b. Objetivo: Indica cada cuanto tiempo agregamos valor a los clientes
2. Change Failure Rate: El porcentaje de despliegues que llevaron a una degradación del servicio.
 - a. Medida: Número de Incidentes / Número de despliegues
 - b. Objetivo: Incrementar la satisfacción de los clientes al reducir el número de downtimes
3. Lead Time for Changes: El tiempo que le lleva a un Commit llegar a Producción
 - a. Medida: Lead Time en días
 - b. Objetivo: Potencialmente puede indicar la productividad del proceso de desarrollo

-
4. **Mean Time To Recovery (MTTR):** El tiempo que le demora a la organización recuperarse luego de un incidente
- Medida: Tiempo medido en horas
 - Objetivo: Incrementar la satisfacción de los clientes al reducir el número de downtimes

F7: Herramientas de SCM

Las herramientas dan soporte a la gestión de configuración en diferentes niveles. Son elegidas por cada organización.

Se utiliza un conjunto de herramientas (no una sola). Cada una sirve para algunas fases del SCM.

Dentro de las herramientas, podemos mencionar las que se ocupan de:

- Gestionar las versiones de los código fuente, archivos de configuración y artefactos relacionados.
- Automatizar el Build y establecer mecanismos (pipelines) para permitir el delivery continuo
- Repositorios que almacenan las historias de los builds
- Gestionar los cambios funcionales y el control de cambios
- Dar soporte al despliegue de aplicaciones
- Comunicar y trabajar colaborativamente

Toolings de automatización de entornos

Todo esto cambió con la aparición de los contenedores. Una forma de ejecutar software en forma aislada del hardware (última revolución en automatización de despliegues, builds, etc).

Con eso surge la nueva cultura de trabajo: [DevOps](#).

Metal / Físico

- Hardware: Dependiente (drivers de placas, video, controladores, etc.)
- Aislamiento: Todos los servicios comparten el SO y kernel, pudiendo afectar a otros.
- Cambios: No hay snapshots, al romperse hay que reinstalar.
- Performance: No hay overhead de performance.
- Aprovisionamiento: Requiere scripts con herramientas para gestión de paquetes, dependencias, acceso al servidor físico.

Máquina Virtual

- Hardware: Cuasi independiente, hardware virtualizado. Sólo respeta arquitectura (x86, ARM)
- Aislamiento: Todos los servicios comparten el SO y el kernel pudiendo afectar a otros.
- Cambios: Más fáciles que metal, existen snapshots, ergo se puede versionar una VM.
- Performance: gran pérdida de performance porque cada máquina tiene varios SO
- Aprovisionamiento: Al tener las imágenes de base y snapshots de una máquina virtual, el aprovisionamiento es más rápido y simple. Un developer podría tener su entorno similar a producción.

Contenedores

- Hardware: Independiente del hardware, solo respeta arquitectura. RAM/CPU flexibles.
- Aislamiento: Filosofía de 1 container por servicio, aislando los servicios entre sí.
- Cambios: Tienen archivos de configuración, son versionables, hay repositorios de containers.
- Performance: mucho más livianos que VM, pierde muy poca performance respecto del metal.
- Aprovisionamiento: Utiliza capas de aprovisionamiento, soporta snapshots, el bootup es en segundos o minutos. Fácil de mover cambios desde y hacia producción.

DevOps - fusión entre desarrollo y operaciones

El problema original

En los modelos de desarrollo tradicionales, el que desarrolla:

- No escribe los requerimientos
- No hace testing funcional
- No pasa a producción
- No está de guardia

Durante décadas los operadores de IT y sistemas vivieron en plena queja con desarrolladores que pensaban que hacer software no es fire & forget, y esperaban que las operaciones manejen automáticamente cualquier problema, sin conocimiento del dominio.

GENERANDO FALTA DE COMUNICACIÓN Y COLABORACIÓN

Definición de 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 **principalmente un cambio cultural**, ya que en un modelo de DevOps, los equipos de desarrollo y operaciones ya no están “aislados” sino que trabajan cooperativamente o directamente se fusionan en uno solo.

DevOps NO es una metodología, ni un rol, es más bien un conjunto de prácticas, herramientas y una **filosofía cultural** que automatizan e integran los procesos entre los equipos de desarrollo de software y IT.

Filosofía de trabajo: CALMS

CALMS es la filosofía de trabajo detrás de DevOps.

1. **Cultura:** Ser dueños del cambio para mejorar la colaboración y comunicación.
2. **Automatización:** Eliminar el trabajo manual y repetitivo lleva a procesos repetibles y sistemas confiables, reduciendo el error humano.
3. Lean: Remover la burocracia para tener ciclos más cortos y menos desperdicio
4. Métricas: Medir todo, usar datos para refinar los ciclos.
5. Sharing: Compartir experiencias de éxito y fallas para que otros puedan aprender.

Están además ordenados por importancia. Cultura y automatización son los más importantes.

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

Paper: But I only changed 1 line of code

Nota: Este paper trata sobre SCM y 4 de las funciones descriptas por SWEBOK y la PPT de clase. Se resaltan en color las novedades. Lo demás se puede obviar.

Software Configuration Management: "El arte de identificar, organizar y controlar modificaciones a software".

Es un proceso que se aplica durante todo el ciclo de desarrollo y mantenimiento.

Determina un set bien especificado de partes de un software y los procedimientos y herramientas exactas para construir el producto (y sus distintas versiones) con esas partes.

SCM se define (en el paper) en 4 áreas funcionales:

1. Identificación de la configuración
2. Control de la configuración
3. Status Accounting
4. Auditoría de la configuración

Configuration Identification

Identificar los Items de configuración (ICs): Partes necesarias para diseñar, desarrollar, construir, mantener, testear y deployar. En general, es cualquier cosa que se necesite para realizar estos puntos.

Objetivo: lograr la integridad del proyecto.

- Ítems fundamentales para el proyecto
- Ítems que cambien en el tiempo
- Ítems que tengan relaciones entre sí
- Ítems que van a compartir entre distintas personas en el proyecto

Línea base: el grupo de ICs que se toman como punto de regreso (safecheck). Implica que el equipo y el cliente den el ok.

Configuration Control

El proceso de controlar y limitar los cambios al software.

Asegurarse que los cambios a un IC sólo sucedan luego de su análisis, evaluación, revisión, y aprobación por un grupo que controla los cambios (Software Configuration Control Board).

El SCCB se asegura de que los impactos de los cambios son correctamente evaluados. Deciden si vale la pena.

Lo forman uno o varios de: program management, systems engineering, software engineering, software quality assurance, software configuration management, independent test, y un customer representative.

Configuration Status Accounting

Es la actividad responsable de rastrear y mantener la información correspondiente a cada uno de los ICs (trazabilidad de los cambios).

Se logra información para trazar:

- Cambios solicitados
- Cambios realizados
- Cuándo fueron realizados los cambios
- Causa de cada cambio
- Quién autorizó cada cambio
- Quién realizó cada cambio
- Qué ICs fueron afectados por cada cambio

Hay reportes como: log de transacciones, log de cambios, uso de recursos, estado de ICs, change request status report, changes in progress, change completion report

Auditoría de la configuración

It is generally accepted that following good software development practices will contribute to consistent delivery of quality software products (assets). Audits should be conducted to help

assure that software development policies, processes, and procedures are being consistently followed and adhered to.

Software configuration audits verify that the software product is built according to the requirements, standards, or contractual agreement. Auditing also verifies that all software products have been produced, correctly identified, described, and that all requested changes are resolved.

- A functional configuration audit is intended to verify that the software functions as defined by the software requirements documentation.
- A physical configuration audit is intended to verify that all the items identified to be included in software release are actually included and that no additional items are included.

Tema 6: Testing de SW

Materiales

- General: PPT de testing, 2024
- Paper: Testing without a Map
- Paper: Using Test Oracles
- Paper: Exploratory Testing explained (opcional)

¿Qué es un SW de “calidad”?

- Aquel que cumplía con los requisitos,
- Aquel que al ser usado NO tenga FALLAS.

Aseguramiento de la calidad (QA → Quality Assurance)

- Es un patrón planificado sistemático de todas las acciones necesarias para brindar una adecuada confianza de que un producto o componente cumple con los requerimientos técnicos establecidos.
- El Testing/Prueba de software es una de las tantas actividades involucradas en QA / aseguramiento de calidad.
- QA implica la revisión y auditoría de los productos y actividades para verificar que cumplen los procedimientos y estándares aplicables, y suministrar a los gerentes de proyecto y otros del resultado de estas revisiones y auditorías.

Asegurar la Calidad vs. Controlar la Calidad (QA vs QC)

Una vez definidos los requerimientos de calidad tengo que tener en cuenta que:

- La calidad no puede “inyectarse” al final.
- La calidad del producto depende de las tareas realizadas durante el proceso.
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos.
- (QC se hace al final, QA se hace todo el tiempo)

Testing

Objetivo

- Encontrar **fallas** en el producto
- De la manera más **eficiente** posible (lo más rápido y barato posible)
- De la manera más **eficaz** posible:
 - Encontrar la mayor cantidad de fallas
 - No detectar fallas que no son (falsos positivos)
 - Encontrar las más importantes

Definición

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. (Se observa, registra y evalúa comportamientos en determinadas condiciones).

Informalmente:

Probar es ejecutar un componente con el objetivo de producir fallas. Una prueba es exitosa si encuentra fallas. Tener como objetivo que el sw no tenga errores, nos hace orientar inconscientemente hacia ese fin.

El proceso del testing

En el testing nosotros debemos encontrar fallas en el producto. Para esto:

- diseñamos y construimos la prueba correspondiente
- determinamos el resultado esperado
- ejecutamos la prueba con una entrada válida
- comparamos el resultado obtenido contra el resultado esperado.

Las pruebas siempre se realizan contra un resultado esperado.



Las cajas azul oscuro comprenden en "Plan de prueba", las celestes la "Ejecución de prueba"

Conceptos clave

Incidente de testing

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

Equivocación vs. defecto vs. falla

Equivocación: Acción humana que produce un resultado incorrecto.

Defecto: Paso, proceso o definición de dato incorrecto, o ausencia de cierta característica

Falla: Resultado de ejecución incorrecto. Es algo producido por el SW distinto al resultado esperado.

- Una equivocación lleva a uno o más defectos que están presentes en el código
- Un defecto lleva a cero, una o más fallas.

-
- La falla es la manifestación del defecto.
 - Una falla tiene que ver con uno o más defectos.

Otros conceptos relacionados

Condiciones de Prueba

Son descripciones de situaciones que quieren probarse ante las que el sistema debe responder. Crear condiciones es un proceso “creativo”.

Casos de Prueba

Son lotes de datos necesarios para que se dé una determinada condición de prueba. Crear casos de prueba es un proceso “laborioso”.

Criterio de selección (de casos de prueba)

Es una condición para seleccionar casos de prueba. De todas las condiciones posibles, sólo se seleccionarán algunas: la menor cantidad de aquellas que tengan mayor probabilidad de encontrar un defecto no encontrado por otra prueba.

Ver: [Partición!](#)

Depuración

- Proceso para **eliminar un defecto que posee el software**.
- No es una tarea de prueba , sino consecuencia de ella.
- La prueba detecta la falla (efecto) de un defecto (causa).
- La depuración puede introducir nuevos defectos.

Proceso

1. DETECTAR el defecto que causa la falla detectada por testing.
2. DEPURAR. Encontrado el defecto debemos:
 - a. Eliminar el defecto
 - b. Encontrar la razón del defecto
 - c. Hallar una solución y aplicarla
3. VOLVER A PROBAR
 - a. Asegurar que sacamos el defecto
 - b. Asegurar que no introdujimos nuevos defectos (regresión)
4. APRENDER PARA EL FUTURO

Partición!

Es imposible realizar una prueba de caja negra exhaustiva, porque se debería probar todos los valores posibles de datos de entrada. Entonces se realiza una selección de casos de prueba.

- Dividimos todos los posibles casos de prueba en subconjuntos de datos de entrada (“**clases de equivalencia**”). Los casos de una misma clase son equivalentes entre sí: detectan los mismos defectos.
- Se selecciona un ejemplo de cada clase y así se logra cubrir todas las pruebas.

Estas pruebas son llamadas “Pruebas por Partición de Equivalencia” o “Pruebas basadas en subdominios”. El éxito está en la selección de la partición !!!

Algunos criterios para seleccionar casos de prueba:

- Variaciones de eventos
- Clase de equivalencia (de entrada / salida)
- Condiciones de borde
- Ingreso de valores de otro tipo
- Integridad del modelo de datos (de dominio / entidad / relación)

¿Cómo se parte en clases de equivalencia?

2 pasos:

1. Identificar clases de equivalencia.
 - a. Dividir cada condición de entrada en 2 grupos: clases válidas / inválidas
 - b. Para cada condición de entrada:
 - i. Rango de valores ($100 < \text{monto} < 200$) \rightarrow 1 válida, 2 inválida
 - ii. Conjunto de valores (DNI / CI / PAS) \rightarrow 1 válida, 1 inválida
 - iii. Debe ser (ej: primera letra es A) \rightarrow 1 válida, 1 inválida
 - c. Todos los casos de prueba de una misma clase son equivalentes entre sí (detectan los mismos errores). Si creemos que los elementos de una clase de equivalencia no son tratados en forma idéntica, debemos dividir la clase en clases menores.
2. Definir casos de prueba
 - a. **Condiciones de borde.** La experiencia indica que los casos de prueba que exploran condiciones de borde producen mejores resultados que las que no lo hacen.

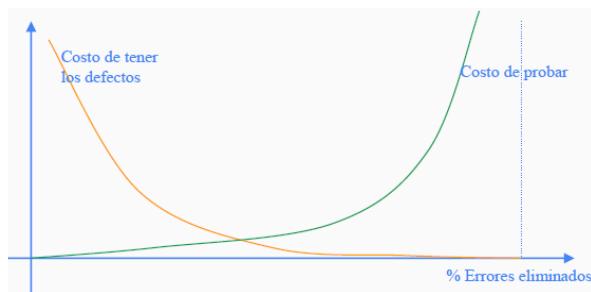
-
- i. Acoto los casos en un rango de valores y clasifico. Ejemplo: listado de códigos postales entre 1000 y 8000.
 - 1. Casos válidos: extremos del rango
 - 2. Casos inválidos: valores siguientes a los extremos
 - ii. Aplico la misma técnica para los datos de salida.
 - iii. Si la entrada o salida es un conjunto ordenado, concentrar la atención en el primero y último de los elementos del conjunto.
 - 1. Prestar especial atención a archivos y tablas vacíos, primer registro/fila, último registro/fila, fin del archivo/tabla.
- b. **Clases inválidas.** Ingresar clases inválidas de diferente tipo al de la clase válida. (Ej: ingreso valores numéricos en vez de alfabéticos o viceversa, fechas erróneas, etc). Si el entorno de desarrollo ya resuelve estas validaciones, no hace falta hacer estos casos de prueba.
- c. **Combinación de datos de entrada.** Esto también puede producir clases válidas o inválidas: ej: si el estado civil es divorciado, los datos del cónyuge se deben ignorar. Estas condiciones cruzadas se agregan a la lista.
- d. **Conjetura de errores o prueba de sospecha.** Creamos casos de pruebas basados en errores posibles o situaciones propensas a tener errores.
- i. Orígenes:
 - 1. Partes complejas de un componente
 - 2. Circunstancias del desarrollo. El desarrollador puede dar input.
 - ii. Es un proceso intuitivo y ad hoc, basado en creatividad y experiencia.
 - iii. Situaciones ideales:
 - 1. Componentes hechos "a las apuradas"
 - 2. Componentes modificados por varias personas en diferentes momentos
 - 3. Componentes con estructuras anidadas, condiciones compuestas, etc.
 - 4. Uso o sospecha de uso de la técnica de copy & paste de otros componentes.

Ver también: [Prueba funcional - de caja negra](#)

Economía del Testing

Probar: establecer confianza en que un programa hace lo supuesto

No se puede demostrar que un programa es correcto. Por ende, continuar probando es una decisión económica.



Final de las pruebas

No hay una receta única, la respuesta depende de cada situación particular.

Criterios posibles de fin de testing:

- El software supera el conjunto de pruebas diseñado
- Se acepta cierta cantidad de fallas no críticas ("good enough")
- La cantidad de fallas detectadas es similar a las fallas estimadas

Abaratamiento

Para abaratar y acelerar el testing -sin degradar su utilidad- se debe **diseñar un software apto para ser testeado** (software testeable).

Técnicas:

- Diseño modular
- Ocultamiento de información
- Puntos de control
- Programación no egoísta

Nota en base a parciales: módulos más cohesivos hacen menos cosas, por ende son más fáciles de testear.

Primeras conclusiones

- Las pruebas no mejoran el SW, sólo muestran cuántas fallas se han producido debido a distintos tipos de defectos.
- El buen diseño y construcción no sólo benefician a las pruebas, sino también a la corrección de los componentes y su mantenimiento.
- No probar no elimina los 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.

Enfoques de prueba (caja negra/blanca/gris)

Prueba funcional - de caja negra

Es una prueba funcional producida por los datos o por la entrada/salida.

Prueba lo que el software **debería hacer**.

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

Cómo se seleccionan los casos de prueba: Ver [Partición!](#)

Definir condiciones y casos

Si partimos de componentes generados por la etapa de requerimientos, un SLDC (ciclo de vida de desarrollo de software) puede producir por ejemplo:

- Wish list
 - Cada statement (declaración) es un requerimiento funcional
 - Un requerimiento que no es testeable no es implementable. Significa que el requerimiento está incompleto.
 - Pensar en “variaciones” de las declaraciones funciona muy bien.
- Casos de uso
 - La secuencia de eventos dentro de un caso de uso tiene 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
- 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 / con un alumno / con muchos (n) alumnos
 - Si n estuviera acotado, aplica además el concepto de condiciones de borde
- Diagrama de transición de estados
 - Si se parte de esto, el ciclo de vida de un objeto define reglas de negocio que deben ser probadas. Transiciones válidas e inválidas.

Conclusiones

- Ninguna técnica es completa
- Las técnicas atacan distintos problemas
- Lo mejor es combinar varias de estas técnicas para complementar las ventajas de c/u
- Sin especificaciones de reqs todo es muchísimo más difícil
- Debemos tener muy en cuenta la conjectura de errores

Prueba estructural: de caja blanca

- Es una prueba estructural. AKA clear box o glass box.
- Prueba **lo que hace el SW**.
 - El tester conoce la estructura interna y la implementación del software.
 - El tester determina las entradas y las salidas para ejecutar ciertos caminos del código.
 - Se usa para incrementar el grado de cobertura de la lógica interna del componente (esto se puede porque se conoce el funcionamiento interno)

Pregunta de final: ¿Cómo una prueba de caja blanca complementa la de caja negra?

Dar ejemplo.

Al conocer cómo está implementado, uno puede agregar condiciones de prueba según eso. Se agrega una condición de implementación, no funcional. No se debe construir condiciones a lo que está escrito en el código, porque las líneas de código van a funcionar... lo que se debe probar es si el código está bien elegido.

Rta de ChatGPT:

- **Mejor cobertura de pruebas:** Mientras las pruebas de caja negra se centran en validar la funcionalidad desde la perspectiva del usuario final, las pruebas de caja blanca se

enfocan en asegurar que todas las rutas posibles dentro del código hayan sido probadas.

- **Detección de errores diferentes:** Las pruebas de caja negra pueden pasar por alto errores internos en la lógica del programa que solo se descubrirían mediante pruebas de caja blanca. Por otro lado, las pruebas de caja blanca pueden no detectar problemas de integración entre componentes, que son capturados mejor por las pruebas de caja negra.

Ejemplo 1: Errores en la lógica del programa. Imagina un sistema de gestión de inventario donde se necesita asegurar que el cálculo de la cantidad disponible de productos se realiza correctamente.

- Caja negra: Se diseñan casos de prueba basados en los requisitos funcionales. Por ejemplo, se ingresan diferentes cantidades de pedidos y se verifican las cantidades restantes de productos después de cada pedido. Si la lógica de cálculo está incorrecta (por ejemplo, no se está restando la cantidad del inventario correctamente), estos errores podrían no ser detectados si los casos de prueba solo evalúan las entradas y salidas sin conocer cómo se implementa internamente el cálculo.
- Prueba de caja blanca: Accediendo al código fuente, se diseñan pruebas para ejecutar diferentes caminos de ejecución dentro del método de cálculo de inventario. Se podrían crear casos de prueba específicos para verificar cómo el programa maneja los límites (por ejemplo, cuando la cantidad de pedido es mayor que el inventario disponible) o cómo se actualizan las variables internas que controlan el inventario. Este enfoque puede revelar errores en la lógica como condiciones incorrectas en las estructuras de control (por ejemplo, un bucle que no termina correctamente) que podrían no ser evidentes sólo con pruebas de caja negra.

Ejemplo 2: Problemas de integración entre componentes.

Supongamos un sistema de comercio electrónico que incluye un módulo de pagos y un módulo de gestión de inventario.

- Prueba de caja negra: Se diseñan pruebas para verificar que el proceso de pago funcione correctamente desde la perspectiva del usuario final. Por ejemplo, se realizan pedidos con diferentes métodos de pago y se verifica que el estado de los pedidos cambie correctamente en la interfaz de usuario. Sin embargo, si hay

problemas de integración entre el módulo de pagos y el módulo de gestión de inventario (por ejemplo, un pago exitoso no actualiza correctamente el inventario), estos problemas podrían no ser detectados mediante pruebas de caja negra.

- Prueba de caja blanca: Se podrían diseñar pruebas que analicen cómo los diferentes componentes interactúan entre sí. Por ejemplo, se podría revisar el flujo de datos entre el módulo de pagos y el módulo de inventario para verificar que las actualizaciones de inventario se realizan correctamente después de cada transacción exitosa. Esto podría incluir pruebas de integración que simulen transacciones reales y verifiquen la consistencia de los datos entre los diferentes componentes del sistema.

Grados de cobertura

- 1er grado: **Cobertura de sentencias.**
 - Prueba cada instrucción (**sólo una rama del if, no ambas**). Indica si todas las líneas fueron ejecutadas con las pruebas. Si alguna no se ejecuta, hay que agregar casos.
- 2o grado: **Cobertura de decisiones**
 - Prueba cada rama (salida) de un "IF" o "WHILE"
- 3er grado: **Cobertura de condiciones** (más cobertura que la de decisiones y sentencias)
 - Prueba cada expresión lógica (A AND B) de los IF, WHILE.
- 4o grado: **Prueba del Camino Básico**/complejidad ciclomática
 - Prueba todos los caminos independientes que pueden ejecutarse en esa pieza de código.
 - Las piezas de código se representan a través de un grafo de flujo
 - **Complejidad ciclomática.** Métrica que da medición cuantitativa de la complejidad lógica de un programa
 - Cantidad de caminos independientes que trae el código.
 - Cada camino independiente agrega un nuevo conjunto de sentencias de procesamiento o una nueva condición
 - Cómo calcularla:
 - Número de regiones del grafo

- $V(g) = A - N + 2$ (A = aristas, N = nodos)
La complejidad varía entre [1, +infinito)
- A mayor complejidad ciclomática, mayor cantidad de casos de prueba tiene que haber. La complejidad ciclomática se relaciona mucho con la mantenibilidad de un SW. A mayor complejidad ciclomática mayores esfuerzos en mantenimiento.

Un mismo conjunto de casos de prueba puede ofrecer distintos grados de cobertura en distintas implementaciones de una función.

Pregunta: ¿Puedo tener cobertura de sentencias al 100% y a su vez tener cobertura de decisiones incompleta?

Se puede, porque un if podría tener una rama en la que no se ejecuta nada. Si no se probó esa rama, se puede tener todas las instrucciones ejecutadas, pero no se probó esa rama.

Prueba de caja gris

Es una prueba que combina elementos de la caja negra y caja blanca. Se conoce **una parte** (no la totalidad) 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.

Por ejemplo, cuando ves el código pero llama a una API externa, conoces una parte de la implementación o estructura interna.

- No es caja 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.
- El conocimiento es “parcial”, no “total” (total sería caja blanca).

Conclusiones de enfoques de prueba

- Las pruebas de caja blanca son un importante complemento a las pruebas de caja negra
 - Hay defectos que serían casi imposibles de detectar a través de caja negra

- Los defectos que originan las fallas son encontrados más rápidamente bajo caja blanca, lo que deriva en una prueba más económica
- 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

Preguntas de parcial: Ejemplo de un defecto casi imposible de reconocer en caja negra, pero más fácil en caja blanca

Respuesta pendiente

Tipos de prueba

Prueba unitaria

- Se realiza sobre una unidad de código claramente definida
- Generalmente, lo realiza el área que construyó el módulo
- Se basa en el diseño detallado
- Comienza una vez codificado, compilado y revisado el módulo
- **Los módulos altamente cohesivos son más sencillos de probar.**

Dice ChatGPT: es una técnica de testing en programación donde se verifican y validan unidades individuales de código fuente. Una "unidad" en este contexto suele referirse a la unidad más pequeña de un programa que puede ser probada de manera aislada. Por lo general, una unidad es una función, método o procedimiento.

El objetivo principal de las pruebas unitarias es asegurarse de que cada unidad del software funciona correctamente según lo esperado. Esto se logra mediante la escritura de casos de prueba que comprueban diversos escenarios y condiciones posibles para esa unidad específica.

Prueba de integración

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

Tipos:

- No Incrementales
 - BIG BANG. Se prueba la integración completa de todos los módulos. (no recomendada, suele romperse todo)
- Incrementales ([requieren conocer la arquitectura de lo que se está construyendo](#))
 - BOTTOM UP
 - TOP DOWN
 - "Sandwich"

Los puntos claves son:

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

Prueba de aceptación de usuario

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**. Lo debería ejecutar un usuario campeón, muy comprometido, con mucho conocimiento de dominio.
- Lo ideal es que el usuario sea experto.

Esto se relaciona con la [conjetura de errores y pruebas de sospechas](#).

Pruebas No Funcionales

Las pruebas no funcionales buscan comprobar la satisfacción de los requerimientos no funcionales del SW.

Se clasifican de acuerdo al requerimiento no funcional bajo testeо. Todos los requerimientos de la [ISO 25000](#) son testeables. Aquí se mencionan algunos ejemplos:

- Volumen.
Verificar que el sistema soporta los volúmenes máximos definidos en la cuantificación de requerimientos (capacidad de almacenamiento, procesamiento y transmisión).
- 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 (ej: demanda esperada, peaks de demanda, etc)
- Ayudan a identificar cuellos de botella y causas de degradación
- Se realizan de la mano de las de volumen
- Stress
 - Se somete al sistema **excediendo los límites de su capacidad definidos** en la cuantificación de requerimientos (capacidad de almacenamiento, procesamiento y transmisión)
 - Similar a la prueba de volumen pero excediendo los máximos definidos
 - Busca el punto de ruptura
 - Busca ver el comportamiento en términos de estabilidad, disponibilidad y manejo de errores. (Se fija si hay una "salida digna").
 - **Simula ser una situación no prevista: desenchufo el server a ver qué pasa**
- Seguridad
 - Prueba los atributos/requerimientos de seguridad del sistema (si puede ser vulnerable, si el control de acceso es adecuado, etc)
 - Ejemplo: Penetration Test (prueba técnica).
 - Se simula el comportamiento de un intruso (interno o externo).
 - 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 y externos.
 - Ejemplo: prueba funcional (20:13hs). Hay que probar que se haga lo que tiene que hacer, y que no haga lo que no tiene que hacer. Ej: roles de seguridad: que pueda hacer lo que el rol demanda, y no hacer lo que el rol no debería poder hacer.
 - Puede ser automática o manual.
- Usabilidad
 - Prueba los atributos de usabilidad definidos en los requerimientos
 - Prueba de campo/observación.
 - Quiere descubrir:
 - ¿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 frustran 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
- Otras pruebas:
 - Recuperación
 - Portabilidad
 - Escalabilidad
 - Etc.

Pruebas de Regresión

(típica del mantenimiento de software)

- Verifica 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.
- Se prueba lo anterior, NO lo nuevo.
- Reutilización de condiciones y casos.
- Se debería evaluar realizarlas en cada mantenimiento.
- La automatización es una muy buena opción.

Prueba de humo / Smoke Test

- Verificar de una manera muy rápida que en la funcionalidad del sistema no hay ninguna falla que interrumpa el funcionamiento básico del mismo
- Test de alto nivel, no se entra en detalles y no se planifica
- Tiene sentido hacerlo rápido antes de pasárselo al equipo de testing.

Pruebas alfa y beta

- Se entrega una primera versión al usuario que se considera que está lista para ser probada por ellos.
 - Normalmente, plagada de defectos
 - Una forma económica de identificar defectos (ya que el trabajo lo hace otro)
 - No siempre se puede hacer
- Alfa : Lo prueba el usuario en mis instalaciones, en un entorno controlado, suele estar presente el developer.

- Beta: la hace el usuario en su propia instalación (usuario final, no controlado por el dev)
 - El entorno no es controlado por el dev
 - El usuario es quien registra las fallas y reporta regularmente

V-Model

Dalceggio: este modelo es un concepto de una época cascadiana y secuencial.

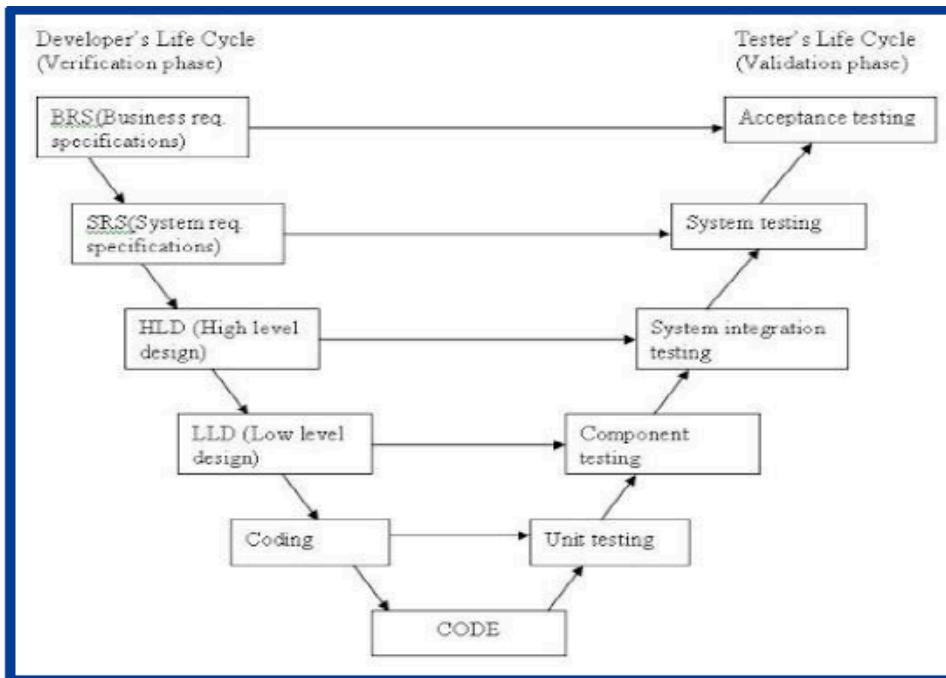
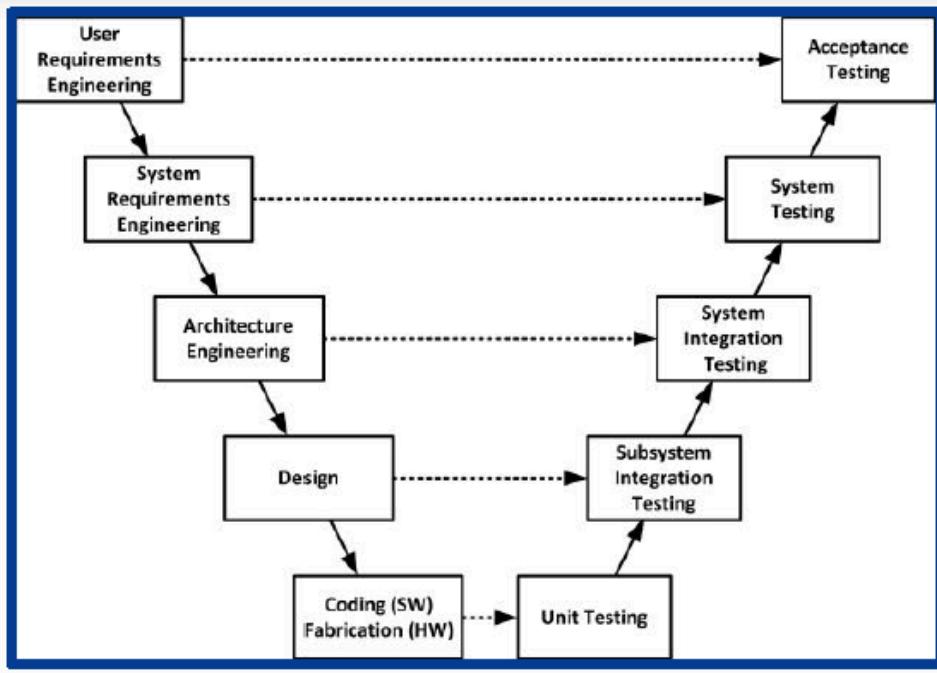
El objetivo es empezar a diseñar las condiciones de prueba lo antes posible.

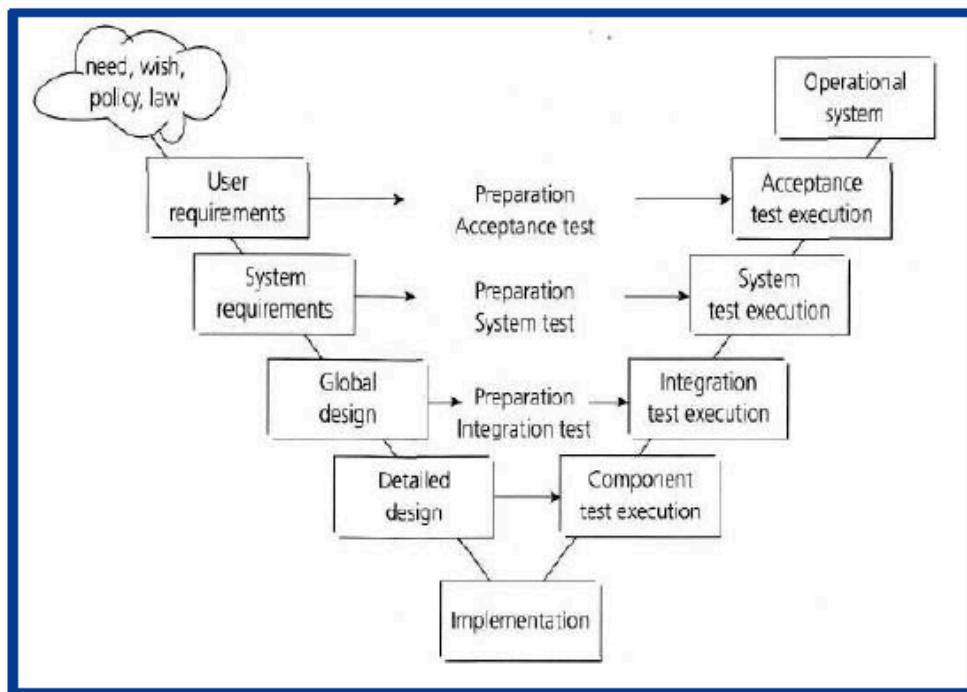
No hace falta esperar a que todo el producto esté listo y andando para empezar a probar. Si tengo los requerimientos del usuario, ya puedo empezar a diseñar la prueba de aceptación.

Si tengo diagrama de arquitectura, ya puedo planear estrategia de integración.

- Al escribir la condición de prueba, genero un requerimiento que me puedo reducir los tiempos de codeo. Puedo detectar requerimientos incompletos.

Cuando está definido...	...Ya se pueden diseñar estas pruebas
Requerimientos de usuario	Aceptación
Requerimientos de sistema	Pruebas no funcionales de sistema
Arquitectura/diseño del sistema	Integración
Diseño de bajo nivel (de componente)	De Componente
Diseño de módulo/Código	Pruebas unitarias





Scripted/exploratory and unscripted testing

Exploratory Testing / Ad hoc testing

Es el aprendizaje, diseño y ejecución de la prueba en forma simultánea.

- Es “unscripted”. No se documenta todas las condiciones y casos previamente.
- El tester es responsable en todo momento de la decisión del camino a tomar. Va decidiendo tácticamente lo que es mejor en cada paso según el conocimiento que va adquiriendo de la ejecución de un test.
- Unscripted no significa sin preparación. Apunta a tener más variantes y no tener restricciones del “script driven”

Scripted vs unscripted

	Script Driven Testing	Unscripted Testing
Info	<ul style="list-style-type: none"> • Primero se confeccionan las condiciones y casos - para luego ser ejecutados. 	<ul style="list-style-type: none"> • Se confeccionan las condiciones y casos a medida que se aprende de las distintas

	Script Driven Testing	Unscripted Testing
	<ul style="list-style-type: none"> ○ Creación de condiciones: creativa, requiere más experiencia ○ Ejecución y reporte: operativo, requiere menos skill ● En cada ciclo de prueba se suelen repetir las condiciones y casos, y no se retroalimentan 	<ul style="list-style-type: none"> ejecuciones, obteniendo ventajas de lo aprendido. ● Las nuevas ideas ocurren "on the fly". 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
Ventajas	<ul style="list-style-type: none"> ● Puede ser objeto de revisión entre pares (otros testers o usuarios finales) ● Reutilizable para una nueva ejecución ● Es medible (cantidad de condiciones y casos) 	<ul style="list-style-type: none"> ● Se pueden variar los test sobre la marcha según lo que se considere más apropiado ● El trabajo creativo se hace durante la ejecución (no antes) ● Mayor cobertura de situaciones sobre posibilidades difíciles de anticipar
Desventajas	<ul style="list-style-type: none"> ● Laborioso ● La etapa más creativa es el diseño de las condiciones y casos y no la ejecución 	<ul style="list-style-type: none"> ● Puede perderse la capacidad de "reproducir" si no se sigue un orden (plan o charter) al probar. (Plan != script) ● Muy dependiente de las personas <ul style="list-style-type: none"> ○ Creatividad / memoria / intuición ○ Testers pueden ser expertos en el negocio (SMEs), en la plataforma tecnológica, conocer el producto antes, etc. ○ Esto puede jugar a favor o en contra

¿Cuándo es apropiado el ET?

- 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

Etapas básicas de la Exploration Testing

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

¿Cómo empezar?

Hay distintos enfoques

- Haciendo un draft de la arquitectura
- Brainstorming de 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 (HICCUP) - permite explicar los errores detectados cuando no se están comparando con requerimientos escritos. Ver [Paper: Testing without a Map](#) / [Paper: Exploratory Testing](#)

Armado del “charter”

Un charter define la “misión” de la sesión de testing (qué testear, cómo testear, y que tipo de defectos se buscan). **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 documentos. Poner foco en el consumo de recursos y el tiempo de respuesta

Conclusión

- ET es una forma de encarar el testing. Muchos testers han aplicado y aplican ET
- ET es dependiente del tester (skills / experiencia / “personalidad”)
- ET es dependiente del conocimiento que se obtiene a medida que se ejecuta la prueba
- A la hora de elegir “script based testing” o “unscripted based testing”, se debe ver cuál conviene dependiendo del contexto o mixear ambos.

Métricas del testing

- Productividad de Diseño (Ej: Condiciones construidas por unidad de tiempo)
- Productividad de Ejecución (Ej: Casos ejecutados por unidad de tiempo)
- Eficacia (Pre Release). Qué tan buenos o no somos en lo que hacemos. Ejemplo: (Incidentes Reportados Aceptados / Total Incidentes Reportados x 100). Es interesante medir también la eficacia en la resolución de defectos
- Eficacia (Post Release). Cuánto evité que se filtrara al usuario. Ej: [Fallas Reportadas / (Fallas Reportadas + Fallas Rep. User)]
- “Calidad” del Desarrollo (Ej: Índice de Severidad por Defectos Reportados)
- Release Readiness (Ej: Índice de Severidad por Defectos Abiertos, es decir defectos no resueltos - me dice si está lista o preparada para producción.)

Preguntas de parcial: ¿Qué pasa en una auditoría de proceso si no cumple con la métrica de Release Readiness?

Si la métrica de Release Readiness da mal, debería no hacerse un release. Si se hace el release igual, la auditoría de proceso va a mostrar que hubo una falla ahí.

Testing Automation

Las pruebas automáticas tienen que ser un complemento de las manuales.

- No intentar eliminar al testing manual
- No buscar suplantar a los testers
- La automatización no debe ser una meta, sino una alternativa

Las automatizaciones abarcan un amplio espectro del proceso de testing (desde la gestión de incidentes a las pruebas de regresión pasando por la generación de casos de prueba).

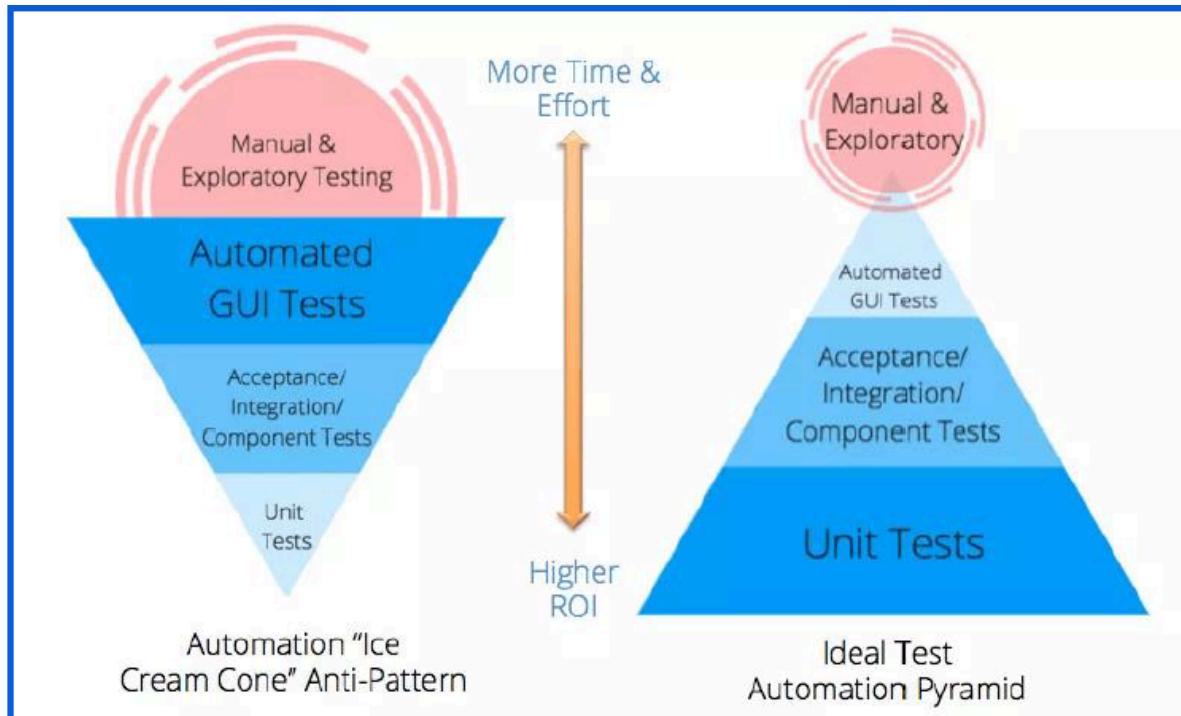
Automatizar lleva tiempo y esfuerzo; y un mantenimiento a medida que nuestro producto cambie.

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

Clasificación de Herramientas de Testing

- Herramientas para gestionar las pruebas y los test cases.
 - Gestión de pruebas, de requerimientos, de incidentes, y 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.

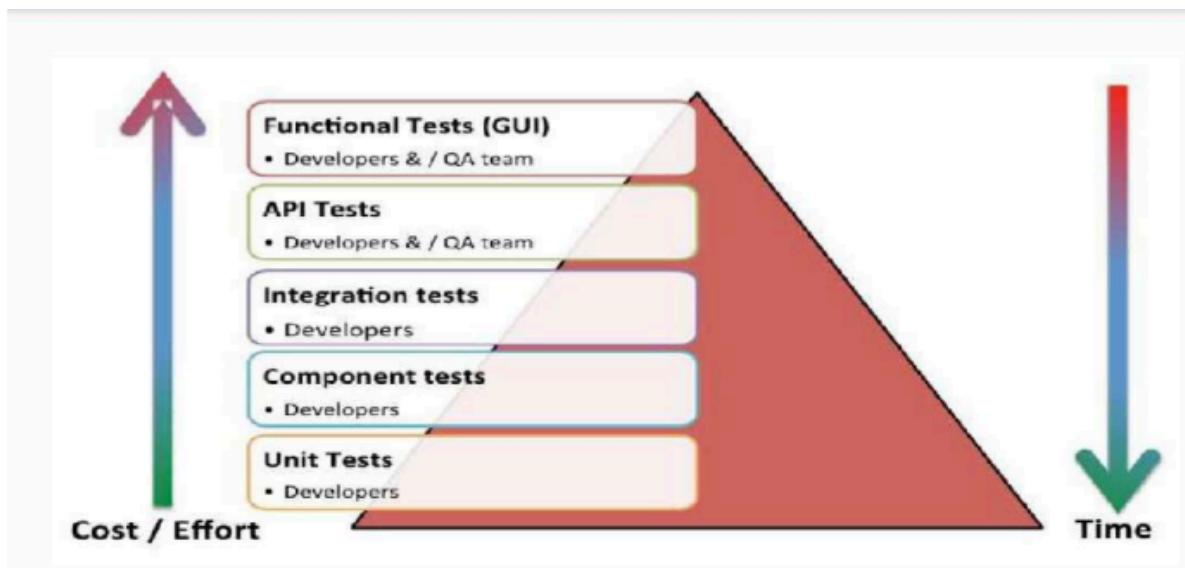
¿Qué automatizar?



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.

Lo que se busca es reducir la necesidad de testeo manual, por ende se recomienda usar la pirámide ideal (más tests en la base de la pirámide, menos tests en la punta, lleva a más ROI y menos esfuerzo en escribir tests y testear manualmente).

Pirámide de testing de Kohn (de otros años)



Ventajas y desventajas

Ventajas

- Cobertura
- Bajo costo de ejecución
- Multiplicabilidad
- Independencia del Tester
- Consistencia (si falla, falla siempre)
- Reuso
- Rapidez

Desventajas

- En algunos casos, el costo de "licenciamiento" / "alquiler" es elevado
- Costo de desarrollo y mantenimientos
- Las fallas pueden deberse a la automatización

Proceso

Antes de arrancar

- Analizar madurez de mi "testing manual"
 - ¿Cuánto tiempo llevo testeando?
 - Tengo un proceso definido? ¿Funciona? Estoy conforme? Lo reviso y lo mejoro?
 - Tengo historia / métricas guardadas? (cantidad de ciclos, esfuerzo incurrido, ventana de tiempo, costos)

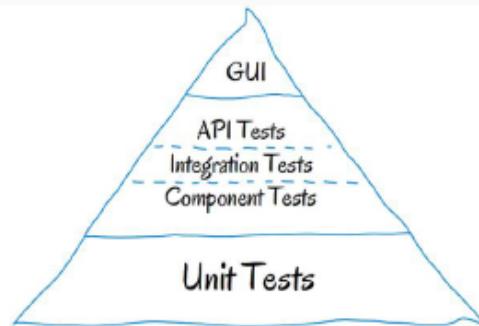
- Capitalicé las “lessons learned”
- ¿Resguardo mis activos? (conocimiento, condiciones y casos de prueba, evidencias, incidentes)

Objetivo

- Identificar la motivación
- Expectativas
- ¿Cuál es mi driver?
 - Umbral de Calidad Not “Good Enough”
 - Más profundidad / Más cobertura / Más repeticiones / Troubleshooting
 - Costo
 - Ventana de tiempo
 - Equipo
 - ¿Son varios de los anteriores?
- ¿Tengo restricciones?
- ¿Tengo flexibilidad?

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



El Caso de Negocio

La presión por el ROI. Usualmente: 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 funciona como soporte y para liberar recursos (NO reemplaza el tester)

Conclusiones

- Acumular experiencia en testing manual
- Setear expectativas
- No esperar resultados de manera inmediata
- No pretender automatizar todo!!
 - Sólo lo que conviene
 - Además, hay cosas que necesitan verificación visual
- No perder vista que las condiciones y casos son el activo
 - La automatización por sí sola no encuentra fallas
- Independizarse lo más posible de una herramienta en particular
- Separar el equipo de automatización del equipo de ejecución
 - Contar con los perfiles adecuados para automatizar
- Analizar proyecto a proyecto (c/situación es particular)
- Esperar cualquier evaluación de automatizar a contar con un producto/aplicación con cierta estabilidad

- Cuidado con la volatilidad de la interfaz de usuario
- Automatizar es un proyecto más y su entregable es una aplicación más ○ Y como toda aplicación queremos que sea mantenible (... y demás atributos de calidad ...)
- Mas todo lo que ya conocemos de cualquier desarrollo ...
- Automatizar debe seguir un proceso ⇒ Medir y Evaluar

Paper: Testing without a Map, de Michael Bolton (obligatorio) + Using Heuristics Test Oracles de Michael Kelly.

M Bolton habla sobre su trabajo con James Bach, del próximo paper. M. Kelly enfoca la heurística como una forma de comunicar/justificar mejor por qué algo que no están en una especificación de requerimientos podría ser un bug.

Pretender testear sólo con especificaciones escritas y completas tiene varios problemas:

- No siempre se las tiene (evaluar si sirve para un fin, testear algo viejo y parcheado)
- que estén completas depende del contexto
- muchas cosas “completas” dejan cosas implícitas

Al explorar en un testeo, se parte de dos conocimientos:

- Oráculos: principio o mecanismo que nos permite reconocer si un software funciona de acuerdo a los criterios de alguien.
- Heurísticas: una guía provisional y falible por medio de la que se puede investigar un problema.

Bolton sugiere la heurística-oráculo HICCUPS. Un producto debe ser consistente con:

- **History:** La funcionalidad actual debería ser consistente con la pasada, salvo que haya un buen motivo para que cambie. ¿Algo cambió sin aviso?
- **Image:** La imagen que debe proyectar el producto debe ser acorde a la que se quiere transmitir como organización. ¿Se ve profesional?
- **Comparable Product:** Se usa productos similares como estándar de comparación. ¿El producto similar tiene este comportamiento?
- **Claims:** ¿Coincide con lo que se ha dicho sobre el producto?
- **User Expectation:** El producto se debe comportar de forma consistente con lo que podría querer o esperar razonablemente un usuario.

- **Product:** Una funcionalidad del software debería ser consistente con otras funciones comparables dentro del mismo. ¿Hay cambios en la terminología / cómo funciona / look and feel dentro del mismo producto?
- **Purpose:** El comportamiento de una función debe ser consistente con su objetivo aparente. Por ejemplo: valores negativos en el tamaño de letra en Word
- **Statutes:** El producto debería respetar requerimientos legales y regulatorios.

Usando estos puntos podemos expresar por qué pensamos que algo es un bug. Acordarnos del paper que mostraba que con esta heurística el tester fue capaz de testear aun sin tener los requerimientos especificados.

El ET permite hacer un testeo que agrega valor rápidamente. Hay que evaluar cuánto agrega hacer una especificación escrita.

Paper: Exploratory Testing Explained, de James Bach (Obligatorio?)

Hacer Scripted Testing puede servir, por ejemplo, si tenemos requerimientos específicos, o si hay tests que se deben ejecutar de la misma forma siempre y servir como benchmark.

ET: Simultaneous learning, test design and test execution.

Diseña los tests mientras se ejecutan, y usa la información obtenida mientras se testea para diseñar nuevos y mejores tests.

La utilidad del ET va aumentando durante el proceso, mientras que la utilidad del scripted testing va disminuyendo, ya que si no encontraste un bug la primera vez que ejecutaste un caso de prueba, las posibilidades de encontrarlo en las siguientes es baja.

Afectan a ET: la misión, el tester, el tiempo que tengamos para ejecutarlo, el producto en sí mismo, cuánto sabe el tester del producto, etc.

Estructura de ET (y sus elementos externos básicos): Over a period of time, a tester interacts with a product to fulfill a testing mission, and reporting results.

Durante el ciclo tenemos que estar alineado a la misión, y ejecutar tests que estén relacionados.

Generalmente empieza con un charter, que te dice la misión y alguna técnica a usar. La puedes elegir el tester o se la puede asignar su test lead/manager. Te dice qué se quiere probar, en general. Es una lista, pero no paso a paso, sino una lista de cosas en general.

El tester tiene que estar entrenado, conocer a la organización.

El resultado es una lista de bugs, y capaz también el material actualizado.

Depende mucho del tester y sus skills:

- Test Design: crear tests que exploren el producto.
- Observadores: hay que observar todo lo que pueda ser inusual o raro.
- Pensamiento crítico: deben poder revisar y explicar su lógica.
- Ideas diversas: pueden usar heurísticas, guías, checklists
- Muchos recursos: herramientas, test data, gente a la que preguntar.

ET sirve para situaciones en las que no es obvio que hay que testear, o si querés ir más allá de los tests obvios. Sirve para:

- dar feedback rápido
- aprender el producto rápido
- diversificar el scripted testing
- revisar el trabajo de otro tester
- investigar un defecto en particular
- improvisar sobre casos de prueba
- interpretar instrucciones vagas
- mejorar casos de prueba ya existentes, o escribir nuevos

Si te trabas con algo, se hace una nota y seguís. No tiene que ser bloqueante. Hay que buscarle la vuelta.

Una estrategia básica de ET es tener un plan de ataque, pero permitirte desviarte por un corto tiempo.

OBLIGATORIO - Test Oracles

By Michael Kelly

Cuando encontramos un bug, hay que convencer al resto de que el defecto es realmente un problema.

El artículo es para aprender a explicar por qué sentimos que algo que nos parece incorrecto es un bug. Así no nos basamos solo en la falta de requerimientos.

No sirve "me parece que tendría que andar así, no de esta otra forma"

HICCUPP test oracles:

-
1. History: ser consistente con versiones pasadas. Si algo cambió, y nadie dijo que debía cambiar, entonces podemos haber encontrado un problema.
 2. Image: no puede dar una mala imagen de la compañía.
 3. Comparable Product: usamos de oráculo a otro producto que sea realmente comparable. 4. Claims: cosas que se dicen del producto, como requerimientos, cosas que dicen los stakeholders, etc.
 5. User expectation: es inconsistente si no hace algo que el usuario esperaría que haga. Hay que tener una idea de quién es el usuario y que esperaría.
 6. Product: algo funciona de alguna manera en una parte, y de otra manera en otra. Podría ser terminología, estética, funcionalidad. El producto sería inconsistente por sí mismo. 7. Purpose: comportamiento contrario a lo que el usuario quiere hacer. Pensar en el objetivo de una feature por ejemplo. Se relaciona con Claims y User expectations.

Usualmente si viola algún oráculo, viola también otros.

1, 3 y 4 les dan más bola. 6 y 7 los managers no le dan tanta bola. 2 y 5 generalmente no les dan bola, entonces se pueden reportar en combinación con otros para que sean resueltos.

Los HICCUPP test oracles son heurísticas, que es una solución que funciona la mayoría de las veces. Osea que puede fallar. Entonces, mejor no basarnos sólo en esto, sino que si hay requerimientos usarlos, citar lo que dicen los usuarios. Usar HICCUPP para agregar data al bug, o explicar otros aspectos de él.

Para imprimir - P1

Dimensiones ISO 25000

- Adecuación Funcional: La capacidad para proporcionar **funciones que satisfacen las necesidades declaradas e implícitas** cuando el producto se usa en condiciones especificadas.
- Fiabilidad (reliability): Capacidad para desempeñar las **funciones especificadas (bajo unas condiciones y periodo de tiempo determinados)**.
- Eficiencia de Desempeño: Desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones.
- Usabilidad : Capacidad del producto para ser entendido, aprendido, usado y resultar atractivo para el usuario (cuando se usa bajo determinadas condiciones).

Characteristics	Sub-characteristics	Measure
Functional Suitability	Functional Completeness	Functional coverage
	Functional Correctness	Functional correctness
	Functional Appropriateness	Functional appropriateness of usage objective Functional appropriateness of the system
Performance Efficiency	Time behavior	Mean response time
		Response time adequacy
		Mean turnaround time
		Turnaround time adequacy
	Resource Utilization	Mean throughput
Compatibility	Capacity	Mean processor utilization
		Mean memory utilization
		Mean I/O devices utilization
	Co-existence	Bandwidth utilization
		Transaction processing capacity
	Interoperability	User access capacity
		User access increase adequacy
	Data formats exchangeability	Co-existence with other products
		Data exchange protocol sufficiency
		External interface adequacy

		Description completeness
	Appropriateness recognisability	Demonstration coverage
		Entry point self-descriptiveness
	Learnability	User guidance completeness
		Entry fields defaults
		Error messages understandability
		Self-explanatory user interface
	Usability	Operability
		Operational consistency
		Message clarity
		Functional customizability
		User interface customizability
		Monitoring capability
		Undo capability
		Understandable categorization of information
		Appearance consistency
		Input device support
		Avoidance of user operation error
		User entry error correction
		User error recoverability
		User interface aesthetics
		Accessibility for users with disabilities
		Supported languages adequacy
	Maturity	Fault correction
		Mean time between failure (MTBF)
		Failure rate
		Test coverage
	Reliability	Availability
		System availability
		Mean down time
		Failure avoidance
		Redundancy of components
		Mean fault notification time
		Recoverability
		Mean recovery time
		Backup data completeness

Security	Confidentiality	Access controllability Data encryption correctness Strength of cryptographic algorithms
	Integrity	Data integrity Internal data corruption prevention
	Non-repudiation	Buffer overflow prevention Digital signature usage
	Accountability	User audit trail completeness System log retention
	Authenticity	Authentication mechanism sufficiency Authentication rules conformity
	Modularity	Coupling of components Cyclomatic complexity adequacy
	Reusability	Reusability assets Coding rules conformity
	Analyzability	System log completeness Diagnosis function effectiveness
	Modifiability	Diagnosis function sufficiency Modification efficiency Modification correctness Modification capability
	Testability	Modification efficiency Test function completeness Autonomous testability Test restartability
Maintainability	Adaptability	Test function completeness Autonomous testability Test restartability Hardware environmental adaptability System software environmental adaptability Operational environment adaptability
	Installability	Hardware environmental adaptability System software environmental adaptability Operational environment adaptability Installation time efficiency Ease of installation
	Replaceability	Installation time efficiency Ease of installation Usage similarity Product quality equivalence Functional inclusiveness Data reusability/import capability

Roles de proyecto vs. SCRUM

Roles de un proyecto	Roles en SCRUM	Explicación
Stakeholders	Stakeholders	Todos los involucrados por el proyecto

Sponsor	~SCRUM Master	Dueño del proyecto. Tiene autoridad para llevar adelante el proyecto.
Usuario Campeón	Product Owner	Experto en el dominio. Define qué es un producto bien hecho.
Usuario directo	-	Operan el sistema
Usuario indirecto	-	Usan el sistema pero no operan
Equipo	Dev team	

CYNEFIN

Marco conceptual para la toma de decisiones.

	Significado	Técnica de decisión	Relación causa-efecto	Acción
Simple	Situación estable, existen reglas probadas para aplicar	Mejores prácticas	Relación clara. Si se hace X, se espera Y	Categorizar sense-categorize-respond
Complicado	La relación causa-efecto requiere análisis o experiencia. Hay muchas respuestas correctas.	Buenas prácticas	Decisión del experto. Se evalúan los hechos.	Analizar sense-analyze-respond
Complejo	La relación causa-efecto sólo puede ser decidida en retrospectiva	Diseños útiles e informativos	Se exploran soluciones, se analizan y se responde al problema	Experimentar probe-sense-respond
Caótico	Relación incierta entre causa y efecto	Cualquier acción es una respuesta adecuada	Se actúa para establecer el orden, se analiza lo que tiene algo de estabilidad, y se responde para transformar el caos en complejidad	Actuar act-sense-respond

Métricas

KANBAN

- Touch time: tiempo neto de trabajo sobre una tarea determinada
- Lead Time: tiempo entre el momento de pedido y el momento de su entrega (desde pedido)
- Cycle Time: tiempo entre el inicio y el final del proceso (para un ítem de trabajo) (desde que se inicia el trabajo después de recibido el pedido)

Para imprimir - P2

Funciones de SCM (según SWEBOk)

1. Administración del Proceso SCM

- a. Contexto Organizacional de SCM
- b. Plan de SCM
- c. Vigilancia del Proceso SCM

2. Identificación de la CS

- a. Identificar ítems a ser controlados
- b. Identificar ítems de 3ros o bibliotecas

3. Control de la CS

- a. Solicitar, evaluar y aprobar cambios al SW
- b. Implementar cambios al SW
- c. Desviaciones y excepciones del proceso

4. SC Status Accounting

- a. Información de Estado de la Gestión de CM
- b. Reportes de Estado de la Configuración

5. Auditoría de la Configuración

- a. Auditoría Funcional
- b. Auditoría Física
- c. Auditoría de Proceso

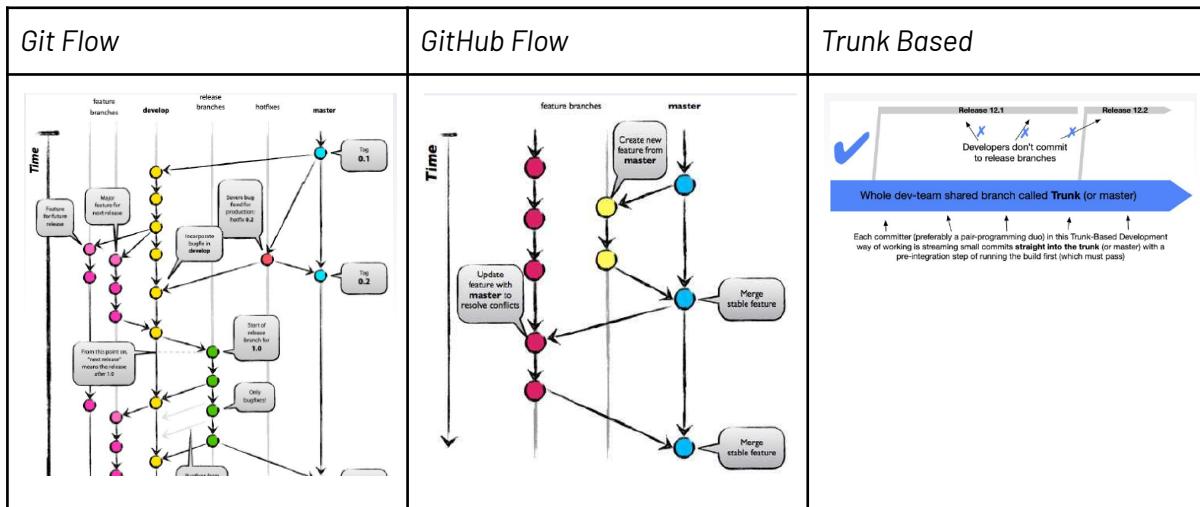
6. Release Management & Delivery

- a. Software Building
- b. Software Releasing

7. SCM Management Tools

- a. Herramientas que dan soporte a diferentes aspectos y niveles de la gestión de configuración.

Branching strategies



Métricas

SCM Change Request	Successful Changes	Cantidad de cambios que han sido aprobados y desplegados en producción sin generar incidentes.
	<i>Unsuccessful changes</i>	Indica: efectividad del proceso de testing
	Changes in Backlog	Cantidad de cambios pendientes de ser analizados. Indica: eficiencia del proceso de aprobación de cambios + velocidad del equipo.
	Emergency Changes	Cantidad de cambios que han sido aprobados por canales de urgencia o emergencia. Indica: si hay hotfixes necesarios
SCM Release Management	Deployment Frequency	Cada cuanto tiempo se despliega software a Producción satisfactoriamente. Medición: Cant. deployments / día Objetivo: Indica cada cuanto tiempo agregamos valor a los clientes
	Change Failure Rate	% de despliegues que llevaron a una degradación del servicio. Medida: Cant. Incidentes / Cant. despliegues % Objetivo: Incrementar la satisfacción de los clientes al reducir el número de downtimes
	Lead Time for Changes	Tiempo que le lleva a un commit llegar a producción [días]. Objetivo: Potencialmente puede indicar la productividad del proceso de desarrollo

	Mean Time To Recovery (MTTR)	Tiempo que le demora a la organización recuperarse luego de un incidente [horas]. Objetivo: Incrementar la satisfacción de los clientes al reducir el número de downtimes
Mantenibilidad	Complejidad ciclomática	Medición cuantitativa de la complejidad lógica de un programa. Cantidad de caminos independientes que trae el código. Cómo calcularla: $V(g) = A - N + 2$ donde A = aristas, N = nodos - La complejidad varía entre [1, +infinito) A mayor complejidad ciclomática, mayor cantidad de casos de prueba tiene que haber. Menor mantenibilidad.
Testing	Productividad de diseño	Ej: Condiciones construidas por unidad de tiempo
	Productividad de Ejecución	Ej: Casos ejecutados por unidad de tiempo
	Eficacia (Pre Release)	Qué tan buenos o no somos en lo que hacemos. Ejemplo: (Incidentes Reportados Aceptados / Total Incidentes Reportados).
	Eficacia (Post Release)	Cuánto evité que se filtrara al usuario. Ej: [Fallas Reportadas / (Fallas Reportadas + Fallas Rep. User)]
	"Calidad" del Desarrollo	Ej: Índice de Severidad por Defectos Reportados
	Release Readiness	Ej: Índice de Severidad por Defectos Abiertos, es decir defectos no resueltos - me dice si está lista o preparada para producción.

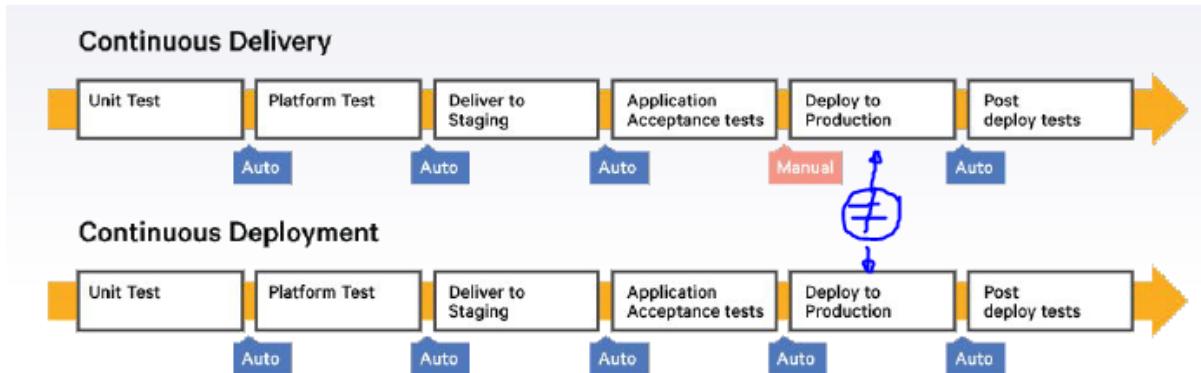
Tipos de Auditorías

1. Auditoria Funcional:
 - a. Es una evaluación del producto software que se realiza para verificar, vía testing, inspección, demostración o análisis de resultados, **que el producto ha cumplido los requerimientos especificados en la línea base de documentación funcional.**
 - b. En las auditorías funcionales, básicamente se verifica que una configuración dada **cumpla con alguna especificación de requerimientos** previamente definida.
2. Auditoria Física
 - a. **Evalúa que los elementos modificados de la configuración sean consistentes con las especificaciones técnicas de los cambios en la versión.**
 - b. Esta auditoría también evalúa los cambios realizados en el código y en los diferentes ambientes
3. Auditoria de Proceso
 - a. Evalúa que los cambios realizados al producto software **hayan seguido el proceso definido para los mismos.**
 - b. El objetivo es confirmar que todos los cambios fueron solicitados, aprobados e implementados de acuerdo al proceso definido en la compañía.
 - c. Ejemplo: Al identificar una línea base se identifica su composición (qué ítems la conforman). Las inconsistencias se reportan y se determina si el defecto corresponde a la configuración misma o a la documentación.

Tipos de Build

1. Local builds: El developer lo realiza localmente en su entorno de desarrollo, corre Pruebas Unitarias (UT)
2. Integration builds: su objetivo es generar el entorno completo para pruebas de integración
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 hay una nueva versión para liberar (lo decide un administrador o el mismo sistema de integración si se hace deployment continuo)

CD vs CD



Filosofía de trabajo de devops: CALMS

1. **Cultura:** Ser dueños del cambio para mejorar la colaboración y comunicación.
2. **Automatización:** Eliminar el trabajo manual y repetitivo lleva a procesos repetibles y sistemas confiables, reduciendo el error humano.
3. Lean: Remover la burocracia para tener ciclos más cortos y menos desperdicio
4. Métricas: Medir todo, usar datos para refinar los ciclos.
5. Sharing: Compartir experiencias de éxito y fallas para que otros puedan aprender.

HICCUPS.

- History: La funcionalidad actual debería ser consistente con la pasada, salvo que haya un buen motivo para que cambie. ¿Algo cambió sin aviso?
- Image: La imagen que debe proyectar el producto debe ser acorde a la que se quiere transmitir como organización. ¿Se ve profesional?
- Comparable Product: Se usa productos similares como estándar de comparación. ¿El producto similar tiene este comportamiento?
- Claims: ¿Coincide con lo que se ha dicho sobre el producto?
- User Expectation: El producto se debe comportar de forma consistente con lo que podría querer o esperar razonablemente un usuario.
- Product: Una funcionalidad del software debería ser consistente con otras funciones comparables dentro del mismo. ¿Hay cambios en la terminología / cómo funciona / look and feel dentro del mismo producto?
- Purpose: El comportamiento de una función debe ser consistente con su objetivo aparente. Por ejemplo: valores negativos en el tamaño de letra en Word
- Statutes: El producto debería respetar requerimientos legales y regulatorios.

Testing - Conceptos a diferenciar

- Incidente: toda ocurrencia que sucede durante la ejecución de una prueba que requiere investigación.
- Equivocación: Acción humana que produce un resultado incorrecto.
- Defecto: Paso, proceso o definición de dato incorrecto, o ausencia de cierta característica
- Falla: Resultado de ejecución incorrecto. Es algo producido por el SW distinto al resultado esperado.
- Depuración: Proceso para eliminar un defecto que posee el software. No es una tarea de prueba , sino consecuencia de ella.
- Condiciones de Prueba: descripciones de situaciones que quieren probarse ante las que el sistema debe responder. Crear condiciones es un proceso "creativo".
- Casos de Prueba: lotes de datos necesarios para que se dé una determinada condición de prueba. Crear casos de prueba es un proceso "laborioso".

--

- Una equivocación lleva a uno o más defectos que están presentes en el código
- Un defecto lleva a cero, una o más fallas.
- La falla es la manifestación del defecto.
- Una falla tiene que ver con uno o más defectos.

Testing de caja blanca: Grados de cobertura

- 1er grado: **Cobertura de sentencias.** Prueba cada instrucción (**sólo una rama del if, no ambas**). Indica si todas las líneas fueron ejecutadas con las pruebas. Si alguna no se ejecuta, hay que agregar casos.
- 2o grado: **Cobertura de decisiones.** Prueba cada rama (salida) de un "IF" o "WHILE"
- 3er grado: **Cobertura de condiciones** ([más cobertura que la de decisiones y sentencias](#)). Prueba cada expresión lógica (A AND B) de los IF, WHILE.
- 4o grado: **Prueba del Camino Básico**/complejidad ciclomática. Prueba todos los caminos independientes que pueden ejecutarse en esa pieza de código.
 - Las piezas de código se representan a través de un grafo de flujo

Un mismo conjunto de casos de prueba puede ofrecer distintos grados de cobertura en distintas implementaciones de una función.

