Modelo Propuesto

El modelo propuesto se basa en el trabajo de Maciá, Berná, Sánchez, Lozano, & Fuster (2016), que separa los proyectos a abordar dentro del Smart Campus en los 6 ejes descritos anteriormente. Antes de exponer los proyectos propuestos por eje, es necesario abordar el modelo de trabajo del macroproyecto Ecosistema Smart Campus de la UNIAJC y a sus colaboradores.

Equipo de trabajo Ecosistema Smart Campus-UNIAJC

El proyecto está encabezado por la Magister en Informática y Telecomunicaciones de la Universidad ICESI, Ana Milena Rojas Calero, docente de carrera, perteneciente a la Facultad de Ingenierías e investigadora del grupo Grintic. Junto con ella se encuentran trabajando durante la fase I: Mg. Hugo Alberto González López, rector de la institución; Mg. Juan Carlos Cruz Ardila, decano de investigaciones y PhD. Leandro Flórez Aristizábal, director del grupo de investigación Grintic. Durante la misma fase, de manera no oficial, participó el MSc. Manuel Alejandro Pastrana, profesor de la misma facultad, quien en segunda fase pasa a ser parte del equipo de trabajo.

En este equipo, se encuentra además un grupo de becarios, quienes participaron durante las dos etapas del proyecto. A continuación, se mencionan como agradecimiento a su valioso esfuerzo:

- Angie Melissa Tobar Bolaños
- Camilo Andrés Muñoz González
- Fernando Andrés Cifuentes Calderón
- Jonathan Arley Rodríguez Pacheco
- Manuela Cardona Arias
- Nasly Katherine Escobar Ortiz
- Nicolás Bermúdez Rodríguez
- Nicolás Grisales Villanueva
- Nicolás Ortiz Acevedo
- Raúl Andrés Hernández Ocampo
- Ronal Andrés Tamayo Zapata
- Rubens Díaz Pullí

Proceso de desarrollo de software Ecosistema Smart Campus

La metodología de desarrollo de software de Smart Campus toma, a criterio de los investigadores, la mejor configuración de roles, técnicas, eventos, artefactos, herramientas y buenas prácticas de diversos marcos de trabajo como XP, SCRUM, DevOps y de metodologías como PMI para la gestión de proyectos. Inicialmente, en el trabajo de Pastrana, Ordoñez, Rojas, & Ordoñez (2019) Springer Nature Singapore Pte Ltd. *Compliance with the client requirements is a key factor for the success of projects. However, this compliance is not always easy to monitor and verify. SCRUM framework flexibility makes it possible to include a variety of techniques and good practices from other methods to help software development teams to achieve their goals. Some of these tools can track historical changes in the code versioning* (El cumplimiento de los requisitos del cliente es un factor clave para el éxito de los proyectos. Sin embargo, este cumplimiento no siempre es fácil de monitorear y verificar. La flexibilidad del marco SCRUM permite incluir una variedad de técnicas y buenas prácticas de otros métodos para ayudar a los equipos de desarrollo de software a lograr sus objetivos. Algunas de estas herramientas pueden rastrear cambios históricos en el control de versiones del código), se describe el proceso de desarrollo basado en scrum, que utiliza las buenas prácticas de XP orientadas a la implementación de algunas herramientas que garantizan la calidad preventiva mediante el enfoque DevOps y, finalmente, se describen los artefactos y prácticas tomadas de PMI para garantizar el éxito de los proyectos.

Scrum es un marco de trabajo muy útil a la hora de ejecutar proyectos de todos los tamaños y niveles de complejidad. Se centra específicamente en generar una cultura de trabajo colaborativo, donde la comunicación entre todos los interesados es el eje fundamental para obtener retroalimentación lo más rápido posible y aprender constantemente como lograr mejores resultados. A su vez, aunque es muy fácil de contar como funciona, es un poco complejo de implementar (Schwaber & Sutherland, 2017). Cada equipo tiene una configuración cultural, una experiencia propia y una forma de trabajo. Por lo mismo, puede resultar complicado implementar todo scrum dentro de un proceso de desarrollo. Por lo anterior, Schwaber & Sutherland (2017) denominan a scrum como un marco de trabajo y no una metodología, debido a que quien busca la implementación de scrum dentro de un proceso de desarrollo, debe tener la suficiente habilidad para determinar qué incluir o no dentro del proceso por cada fase del ciclo de desarrollo descrito por Pressman & Maxim (2015). A continuación, se detallarán los roles, artefactos, eventos y prácticas tomadas de scrum y las buenas prácticas de desarrollo de software de XP.

Roles dentro del proyecto

El Ecosistema Smart Campus adopta los roles basados en scrum, con responsabilidades específicas bien definidas, de acuerdo a lo descrito por Schwaber & Sutherland (2017) y un rol extra, que es el gerente de proyecto, quien cumple con orientar el proyecto a fin de que este tenga los resultados esperados de acuerdo a la visión institucional. A continuación, se describen cada uno:

Product Owner

Según el enfoque de Schwaber & Sutherland (2017), este rol es ejecutado por quien tiene más clara la visión del producto. La responsabilidad principal es la de representar a todos los stakeholder a través de su opinión y visión. Centra su esfuerzo en ayudar a que todos los integrantes del proyecto entiendan las necesidades funcionales a resolver, la importancia o prioridad que tienen para el cliente; ayudar a pactar los compromisos, ordenar el trabajo, a realizar y proveer a los otros roles de todo lo requerido para que puedan realizar el proyecto de manera adecuada cuando lo que necesitan está del lado del cliente. Este rol no es un gerente de proyecto, ni un dueño del equipo. Tan sólo es quien ayuda a orientar la visión para que se cumpla.

Scrum Master

Es un integrante del scrum development team, con una responsabilidad específica: "eliminar cualquier impedimento para lograr las metas del sprint", como indican Schwaber & Sutherland (2017). Este rol es quien mejor conoce scrum y hace que se cumpla. Se encarga de organizar todos los eventos de scrum (sprint, planning meeting, daily scrum, sprint review, sprint retrospective) y apoyar a que se cumplan adecuadamente. Apoya al equipo con sus problemáticas como solicitudes de información al cliente, aclaración de dudas, pero también con cosas simples como ubicación cómoda del espacio de trabajo, material para trabajar, e incluso preparación técnica. Adicionalmente, corrobora que los artefactos fundamentales de scrum se construyan de manera adecuada (product backlog, sprint backlog y burndown chart).

Scrum Development Team

Schwaber & Sutherland (2017) indican que este rol es quien construye el producto con la ayuda de los otros dos roles. Su objetivo principal es lograr el cumplimiento de los objetivos del sprint. Se conforma por equipos auto-gestionables que

organizan su trabajo de manera adecuada y son capaces de trabajar enfocados en compromisos y no en horas. Son los responsables de construir junto con el product owner y bajo la orientación del scrum master los artefactos del proyecto, incluyendo la unidad desplegable para poner en producción.

Gerente del Proyecto

Este rol está enfocado en dirigir de manera adecuada el Proyecto, supervisando el cumplimiento de los compromisos de acuerdo a la planeación entregada por el equipo y supervisando que las entregas cumplan con lo esperado por la institución. Este rol normalmente lo cumple el director de proyecto de grado, quien es uno de los coinvestigadores inscritos dentro del proyecto.

Análisis y planeación

Según Pressman & Maxim (2015), esta etapa consiste en la identificación de las necesidades funcionales de los sistemas a construir, que formalmente se denomina elicitación de requisitos. Aquí los encargados de recopilar la información utilizan diversas técnicas que permiten formalizar lo indagado en un artefacto (documento que cumple con características específicas de acuerdo a un estándar o a una convención).

Las técnicas descritas por el mismo autor son la entrevista, la encuesta, la lluvia de ideas, pero Smart Campus, basado en el trabajo de (Pastrana, Ordóñez, Ordóñez, & Merchan (2017), decide adoptar la técnica inception deck.

Técnica de elicitación de requisitos utilizada

La técnica creada por Rasmusson (2010) consiste en una serie de 10 dinámicas que permiten al auditorio unificar la visión del producto a partir del consenso. A continuación, usando el trabajo de Pastrana et al. (2017), se describe cada etapa:

Initial activity: Who is in the auditorium?

I. Why are we here? And who's in the audience? Estas dos dinámicas están centradas en identificar si todos conocen la razón de la reunión y tienen algún grado de conocimiento sobre la visión del producto. Adicionalmente, confirma que estén los interesados que mayor aporte puedan generar a la reunión.

II. Creating an Elevator Pitch. En caso de no conocerse la visión del producto se debe crear un pequeño discurso que resalte en menos de 5 minutos las ventajas

y beneficios del proyecto, él porqué es necesario y el a quién está dirigido.

III. Design a product box. Esta dinámica permite centrarse en el nombre del producto, una frase que lo identifique y el posible logo. Se parte del símil con la frase: "Si el producto estuviera terminado y puesto en una vitrina de ventas como se vería".

IV. Create a list of "what is not". Esta dinámica está diseñada para delimitar el alcance del proyecto en todos los sentidos posibles, desde funcional hasta tecnológico. Siempre que pensamos en negativo nuestro cerebro instintivamente debe recurrir a identificar que sí es para poder delimitar esta visión.

V. Meet your neighborhood. En caso de que las personas del auditorio no conozcan quienes estarán participando dentro del proyecto, se realiza una breve presentación de cada uno indicando su rol, aspectos a aportar, cosas que no gustan del producto y datos de contacto.

VI. What keeps us up at night? Preocupaciones. Son los riesgos que están desde el inicio del proyecto y deben ser atendidos cuanto antes. Vital para un buen comienzo.

VII. Show the solution: 4 formas de mostrar la solución, donde cada una tiene un enfoque particular.

a. Give the app some personality

b. Let's make it flow

c. Wireframing

d. Stories map

VIII. What's it going to take? Compromisos de los participantes para que el Proyecto salga a flote.

IX. How much will it cost? ¿Qué retos existen en aspectos no económicos? ¿Pruebas de concepto? ¿Investigación? ¿Cambios internos de la empresa? Son preguntas constantes que deben hacerse para resolver este punto.

X. Summary. Se presenta el resumen de los resultados

Artefacto de la etapa de análisis: Product Backlog

El resultado final de la técnica aplicada es el artefacto fundamental para el proceso de desarrollo de software denominado Product Backlog. Según Schwaber & Sutherland (2017), creadores de scrum, este documento recopila la información funcional con el detalle suficiente a través de un modelo de historias de usuario que permite su escritura. Es importante resaltar que para la ingeniería de software no existe un modelo estándar de historias de usuario hasta la fecha, por tal razón, el equipo decide adoptar el modelo de (Cohn, 2004), que a través de la siguiente frase permite fácilmente seguir una estructura para la escritura de la información: yo como [ROL]+ requiero/espero/deseo/necesito [una funcionalidad] +para [motive]+con los siguientes [Criterios de aceptación]. Con base en esto se crea la siguiente estructura base de plantilla para la recopilación de las historias de usuario:

Tabla 1: Modelo inicial de Product Backlog. C.A = Criterios de Aceptación

ID	Rol	Funcionalidad	Motivo	C.A.
	Yo como [Rol]	Requiero [Funcionalidad]	Para [Motivo]	

La identificación del ID permite referenciar rápidamente la HU sin tener que entrar a leer todo el detalle. Esto facilitará ejercicios como la estimación y la priorización en la construcción de la planeación de sprint backlogs. La columna Rol indicará al equipo rápidamente quien ejerce la acción. Es importante no confundir el sistema como un rol ya que este no lo es. La columna funcionalidad expresa siempre la acción a ejecutar. Una regla común a la hora de crear un lenguaje común es que la funcionalidad siempre se exprese como un verbo en infinitivo (terminado en ar, er, ir). Esto facilita garantizar que la historia de usuario es atómica, es decir, que solamente responde por una funcionalidad a la vez. Frecuentemente, una confusión que se da respecto a este aspecto es si se pueden utilizar o no verbos como gestionar que agrupa 4 funcionalidades dentro de sí (crear, modificar, consultar, eliminar). La recomendación que da tanto Cohn (2004) como Pastrana et al. (2017) es que no, debido a que esto incrementa el número de criterios de aceptación haciendo muy difícil de leer y comprender la HU, además de aumentar la complejidad de la misma en el momento de realizar la estimación.

Cohn (2004) sugiere que las HU deben llevar en su estructura una justificación o motivo que lo da el para. Cuando un interesado del proyecto puede justificar

por qué necesita la HU significa que tiene claro su valor de negocio, pero cuando le cuesta trabajo significa que realmente no está muy claro qué quiere hacer y es deber del equipo ayudarle a encontrar dicho valor o descartar la HU. Finalmente, los criterios de aceptación dan el detalle adecuado para poder comprobar si se ha cumplido con una HU. Algunas reglas útiles para escribir criterios de aceptación son: ¿dónde sucede la acción?, ¿qué componentes de pantalla hay?, ¿campos, botones, listas, paneles colapsables, imágenes, etc?, ¿tienen alguna restricción estos campos como tamaño, formato archivos a utilizar, columnas a mostrar en una tabla?, ¿tienen obligatoriedad?, ¿qué mensajes de error y de éxito se deben mostrar? Recordar que no existe un nivel de detalle estándar para los criterios, pero entre más claro sea, más fácil será su implementación.

Una vez este modelo base de la documentación del análisis de requisitos es construido, se da paso a su completitud, adicionándole los campos requeridos para la estimación y priorización bajo técnicas ágiles y un campo extra de observaciones que permite durante la etapa de refinamiento de las historias adicionar comentarios para ser resueltos previos a la estimación.

Tabla 2: Modelo final de Product Backlog. C.A = Criterios de Aceptación. C = Complejidad. P= Priorización. Obs = Observaciones

ID	Rol	Funcionalidad	Motivo	C.A.	C	P	Obs
	Yo como [Rol]	Requiero [Funcionalidad]	Para [Motivo]				

Lo anterior deja entrever que las historias de usuario son susceptibles a refinamiento constante, debido a que están escritas en lenguaje natural y su interpretación puede variar por quien las lee a menos de que se construyan bajo un lenguaje común y pactado previamente entre todos los interesados del proyecto. El impacto de la ambigüedad está contemplado y reducido en alguna medida, a partir del uso de la técnica inception deck y del modelo de historias de usuario utilizado. Adicionalmente, bajo la sugerencia de Pastrana et al. (2017), en algunos proyectos con gran complejidad, el uso de modelamiento de procesos de negocio se utiliza como complemento de las HU para garantizar una mayor claridad de la información.

Adicionalmente, también es posible entender a este punto que la actividad de estimación y priorización es complementaria y posterior a la elicitación de requisitos. Por lo que el equipo primero debe entender el contexto y todo

lo que se requiere previo a afrontar cómo podrían planear los compromisos del proyecto y las entregas parciales. A continuación, se da un ejemplo de una historia de usuario construida para un proyecto aplicando todo lo indicado anteriormente:

Id	Rol	Funcionalidad	Motivo	C:A.	C	P	Obs
	Yo como administrador	Requiero ingresar a la plataforma de manera segura	para garantizar que las acciones del administrador no las puede ejercer ningún otro rol	1. la pantalla contiene el campo usuario. Obligatorio 2. La pantalla contiene el campo contraseña. Obligatorio 3. la pantalla tendrá un boton "Acceder" que permite enviar los datos para ser validados 4. Al presionar el botón, el sistema valida que los campos usuario y contraseña no estén vacíos, en caso contrario, el sistema desplegará el siguiente mensaje "Favor ingresar campos obligatorios". Adicionalmente, los campos dejados en vacío se resaltarán con color rojo y un asterisco que indique son esos los campos a corregir. 5. Cuando los datos son correctos el sistema los envía al servicio de autenticación de smart campus y espera la validación. si es correcta da acceso con el rol administrador. En caso contrario indica el siguiente mensaje: "Usuario o contraseña incorrecto, por favor volver a intentar"			Usuario y contraseña son los mismos de Academusoft

Figura 3. Ejemplo de HU aplicando. C.A = Criterios de Aceptación. C = Complejidad. P= Priorización. Obs = Observaciones

Se recomienda al lector dar una revisada al **anexo a**, adjunto a este libro para que pueda tener un modelo a seguir de acuerdo a la plantilla generada por el equipo de trabajo.

Técnica de estimación de requisitos utilizada

El proceso de estimación bajo la perspectiva de los marcos de trabajo ágil se enfoca en entender que el tiempo es una variable que cambia tanto como los

diversos factores con los que un equipo afronta el proyecto. Aspectos como la experticia del equipo de desarrollo, que tanto se conocen entre sí, que tanto dominan las tecnologías del proyecto, que tan buenas máquinas de desarrollo tienen, que tanta calidad deben incluir, si conocen las tecnologías de calidad, las técnicas y las medidas de calidad, etc., generan variabilidad en el tiempo que toma construir un proyecto. Por tanto, los modelos ágiles indican que las medidas de tiempo que se utilizan en los métodos predictivos intentan predecir algo que sin una continuidad de datos históricos formalizados a través de fórmulas no se podrían predecir.

Para dar un ejemplo simple de la pretensión del agilísimo, se plantea el siguiente escenario.

La distancia entre un punto A cualquiera y B es de 20 km.

a 20 KM B

Si una persona decidiera recorrer esa distancia a pie, caminando podría tomarle unas 3 horas aproximadamente. Pero si dispone de buen estado físico y decide trotar, podría tomarle alrededor de 1,5 horas. La mitad del tiempo inicial. Si decidiera, en lugar de trotar, tomar una bicicleta y recorrer esa misma distancia, probablemente en 45 minutos lo haría. Claro está, si lo hace en un vehículo más rápido como un carro o una moto, su tiempo sería mucho menor.

Así mismo funcionan los proyectos de desarrollo de software. El vehículo sería la tecnología a usar: lenguajes de programación, frameworks, generadores de códigos, IDE o entornos de desarrollo y herramientas de calidad preventiva. Entre más dominio tengamos de estos, el tiempo de desarrollo será menor, pero si desconocemos varios de estos el tiempo de desarrollo será mucho mayor.

Por lo anterior, los modelos ágiles no se centran en calcular el tiempo sino la dificultad a resolver bajo una percepción subjetiva que solamente puede ser dada por el equipo de desarrollo.

Con respecto al tiempo que requieren los modelos tradicionales de gestión de proyectos y los cálculos gerenciales de métricas de control de productividad, hay una solución muy simple y es el uso de compromisos fijos por sprints.

Según Schwaber & Sutherland (2017), un sprint se le denomina a un periodo de tiempo fijo que puede tener una duración de 1, 2, 3 o 4 semanas. Ese periodo de tiempo es en el que se divide el proyecto a realizar y es el mismo durante todo el proyecto, por lo que puede ser inferior de lo mencionado, que un proyecto que está compuesto por varios sprints.

Para poder afrontar la estimación, Ecosistema Smart Campus UNIAJC adopta póker scrum. Primero se explicarán las barajas posibles a utilizar y posteriormente se detallará el proceso de estimación para que pueda ser replicado. La técnica consiste en una baraja con unos rangos de número y 3 cartas comodines. Existen dos tipos de baraja muy utilizados a nivel de la industria del desarrollo de software. La figura 3 refleja la baraja original, que utiliza la serie de Fibonacci para la sucesión de números. Es importante recordar que la serie de Fibonacci indica que el valor siguiente es la suma de los dos anteriores. Los valores reflejan la percepción de complejidad asociada a la funcionalidad o en palabras más simples, en esa escala de valores qué tan difícil considera que es hacer la funcionalidad requerida. Por parte de las cartas comodines tenemos tres opciones importantes a revisar. La primera es el símbolo de pregunta (?). Cuando alguien del equipo utiliza esta carta significa que la funcionalidad expuesta no contiene detalle suficiente para comprenderla y poder dar una apreciación de la estimación. Por otra parte, la carta infinita (∞) significa que la funcionalidad expuesta tiene tanta información que su complejidad tiende a ser muy grande. Ambas cartas sugieren al equipo de desarrollo scrum que deben refinar la historia que se está afrontando y darle suficiente detalle adecuado para la percepción de la estimación. Es importante resaltar que refinar una historia de usuario (HU) puede llevar a dos situaciones muy comunes. La primera es que la HU le falte detalle en los criterios de aceptación. La segunda es que la HU tenga tanto detalle en los criterios que la historia probablemente lleve más de una funcionalidad incluida y, por tanto, para conservar su atomicidad esta debe ser descompuesta en varias HU. Se sugiere tener presente que al descomponer un HU en varias, su estimación no debe dar la suma de la estimación anterior, sino que cada HU nueva se debe estimar de manera independiente, como una funcionalidad sin dependencia de otras.

Figura 4. Baraja de Ppker Scrum basado en Fibonacci.
Fuente: Aplicación scrum poker disponible en Google Play

La figura 4 muestra la baraja más aceptada por la industria. Esta baraja contiene la misma definición de la serie de Fibonacci, pero con un rango más acotado de valores que van del 0 al 100. Esta última baraja es la que es adoptada por el equipo de trabajo y utilizada en los proyectos por parte del Ecosistema Smart Campus UNIAJC.

Figura 5. Baraja de Poker Scrum Standar.
Fuente: Aplicación scrum poker disponible en Google Play

Pasos para la estimación por poker scrum

Una vez comprendida la baraja a utilizar, es importante exponer la técnica a fin de tener un pleno conocimiento del proceso de planeación bajo el modelo de trabajo del equipo de Smart Campus. A continuación, se exponen los pasos requeridos para llevar a cabo la técnica con éxito:

Pasos previos

- El product backlog debe contener todas las historias de usuario refinadas y con el detalle suficiente para que el equipo pueda realizar el proceso.

- El equipo de desarrollo scrum debe leer previo a la reunión el product backlog y anotar todas sus dudas o sugerencias previas a la reunión.

- Si hay sugerencias o dudas estas deben ser expuestas al equipo para ser resueltas previas a la reunión.

- El product owner debe estar disponible para consultas en caso de requerirlo.

Inicio de la estimación

Se debe fijar un moderador de la reunión, que normalmente es el scrum master.

Se leen todas las historias de usuario en su definición épica. Recordar que Cohn (2004) indica que una HU épica es la primera parte de la estructura sugerida por él y que no incluye criterios de aceptación (Yo como [rol]+ requiero [funcionalidad] +Para [motive]).

I. Se selecciona la HU que parece la más fácil de hacer y la más difícil. Esto se denomina fijar el piso y el techo de la estimación.

II. Se leen los criterios de aceptación de las HU piso y techo para corroborar que efectivamente son las más fácil y difícil de realizar. Si todo el equipo está de acuerdo se seleccionan.

III. Se estima la HU más fácil o piso. Para la estimación el moderador corrobora que la HU es completamente comprendida. Si no es comprendida, la historia de usuario debe ser refinada inmediatamente. Si es comprendida, cada integrante del equipo de manera secreta selecciona un valor de la baraja del póker scrum, y no comenta este valor hasta que el moderador lo indique. Esto se hace con el fin de no sesgar la opinión de los demás, sino fomentar la participación de todos y llegar a un consenso a través de la diversidad de opiniones. Cuando el moderador indica que todos muestren su selección, todos los participantes muestran la carta de la baraja que han seleccionado. Aquí pueden suceder varias situaciones. La primera y la ideal es que todos tengan el mismo valor en la votación. En este caso

significa que todos están de acuerdo con que el valor de estimación es ese. El segundo caso es que algunos integrantes tengan un valor y otros otro (división del equipo), por lo que el moderador solicita que se expongan las opiniones de todo para poder llegar a un consenso. Puede suceder que en algunos casos el equipo decida adoptar una de las votaciones como el valor de complejidad o dificultad de la HU. En otros casos, donde aún no se han puesto de acuerdo, se solicita volver a estimar. El moderador debe intentar que el máximo número de votaciones que se haga sobre la misma HU no supere el valor de 3. Hay que tener presente que el valor de complejidad asignado a la HU siempre es el consenso del equipo, nunca se hace promedio de los valores. De esta manera queda asignado el valor de complejidad de la historia de usuario más fácil, que se denomina mínimo de la estimación. No existirá un valor más pequeño que el de esta HU.

IV. De la misma manera que se realizó la estimación para la HU más simple se realiza para la máxima.

V. Una vez fue fijado el valor máximo y mínimo de la estimación, se procede a seleccionar un valor intermedio entre esa escala. A ese valor se le denomina pivote. Cuando un equipo tiene dudas sobre una HU, si está sobre estimada o sub estimada, el equipo debe compararla contra el pivote. Con esto tiene un punto intermedio para poder identificar si la historia se acerca más al pivote o se aleja más del pivote hacia alguno de los extremos.

VI. Con los valores de máximo, mínimo y pivote fijados, se procede a estimar todas las demás HU del product backlog.

VII. Con todas las HU estimadas se suman los pesos de todas las HU para realizar el cálculo del peso del proyecto. En la tabla 3 se puede apreciar un breve ejemplo:

Tabla 3. Ejemplo cálculo del peso del proyecto

ID	Estimación
HU-01	1
HU-02	3
HU-03	8
HU-n	13
Total	25

VIII. Con el peso de todo el proyecto calculado, el equipo de trabajo ya puede proceder a calcular el número de sprints que tomará realizar el proyecto. Para esto lo primero que se hace es fijar el tamaño del sprint. Recordar que de acuerdo a Schwaber & Sutherland (2017), un sprint es un periodo de tiempo que va de 1 hasta máximo 4 semanas. Con el mismo tamaño durante todos los sprints del proyecto. Esto quiere decir que, si se define que los sprint serán de 2 semanas, todos los sprints del proyecto tendrán ese tamaño. Con el tamaño del sprint seleccionado, el equipo debe fijar un compromiso de puntos a cubrir durante ese periodo de tiempo. Para identificar el valor más pequeño de compromiso posible a asumir se sugiere que se multiplique el tamaño del equipo por el valor de la historia de usuario más pequeña y este valor por el número de semanas que implica el sprint. Es decir, que si la historia de usuario más pequeña tiene una complejidad de 2, el equipo de trabajo está compuesto por 5 personas y cada sprint dura dos semanas; un valor mínimo de compromiso podría ser 2*5*2 = 20. Esto implica que cada persona se compromete a realizar dos puntos de complejidad por sprint. Claro está, este es el escenario de compromiso más bajo posible. Otra forma de calcularlo es utilizando una fórmula similar a la anterior, pero sin multiplicar por el número de semanas del sprint y con la historia de usuario más difícil. En este caso, si por ejemplo la estimación máxima da un valor de 20, existen 5 integrantes; el valor del compromiso del sprint será de 100 puntos a cubrir. Una tercera forma de asumir el compromiso, y que es la más utilizada, es colocar un valor por encima del compromiso calculado con la historia de usuario mínima e ir generando un incremento a medida que el equipo aumenta la velocidad de desarrollo.

IX. Calcular el número de sprints. Cuando se divide el tamaño del proyecto por el compromiso se obtiene el número de sprints que tomará hacer el proyecto realidad.

A continuación, se da un ejemplo más detallado de cómo aplicar la fórmula, suponiendo la siguiente tabla como el resultado de la estimación de un proyecto:

Tabla 4. Ejemplo de product backlog estimado sin detalle de las HU.

ID	Estimación
HU-01	5
HU-02	5
HU-03	8
HU-04	13
HU-05	8
HU-06	5
HU-07	5
HU-08	5
HU-09	5
HU-10	8
HU-11	13
HU-12	20
HU-13	5
HU-14	5
HU-15	13
HU-16	20
HU-17	8
HU-18	13
HU-19	13
HU-20	13
HU-21	13
HU-22	13
HU-23	13
HU-24	5
HU-25	5

Peso total del proyecto: 239 puntos.
Peso HU Mínima: 5
Peso HU Máxima: 20
Pivote: 8
Duración de los sprints: 2 semanas
Compromiso del equipo: 36 puntos
Número de sprints para construir el proyecto: Peso total del proyecto/ Compromiso del equipo = 239/36 = 6.63

Recomendaciones sobre la estimación

Dos recomendaciones muy útiles para los equipos de desarrollo que apenas empiezan a abordar la técnica son los siguientes:

1. A la hora de estimar una historia de usuario, pensar que todo lo que se debe realizar es vital. Como mínimo toda historia de usuario está dividida en backend y frontend, es decir, en la lógica que componen la funcionalidad y la pantalla. Para poder construir esa funcionalidad que se está estimando, es necesario realizar una serie de actividades que involucran tener en cuenta: desde cuántos elementos hay en pantalla que capturan o exponen

datos, validaciones en pantalla, mensajes de error o de éxito; el paso de esta información hacia la lógica y el retorno de la respuesta desde la lógica hacia la pantalla, la estructuración de clases de la lógica regida por patrones como MVC, MVP, MVVM, BLoC, etc.; las consultas, si estas pueden ser heredadas de un método genérico o deben ser implementadas de manera nativa; el número de relaciones de las tablas para las consultas, etc. Adicionalmente, el versionamiento, las pruebas unitarias, las pruebas de integración, las revisiones y refinamiento producto del análisis estático de código, las revisiones funcionales que aseguran que la HU está cumplida. Esto hace que la percepción real a la hora de estimar no sea como se acostumbra, pensando sólo en líneas de código, sino pensando en un panorama mucho más completo.

2. Cuando se realiza el cálculo del número de sprints del proyecto y da un valor decimal, como en el ejemplo anterior que da 6.63, lo más recomendable es indicar que el número de sprints es el siguiente valor entero. En este caso 7 sprints tomaría hacer el proyecto. Esto se hace con el fin de no asumir compromisos parciales y utilizar el tiempo restante del compromiso para incrementar las prácticas de calidad.

Técnica de priorización de requisitos utilizada

En los marcos de trabajo ágiles como scrum y XP se abordan diversos modelos de priorización. El objetivo principal de esta labor es dar un orden de atención a las historias de usuario que se realizarán en la distribución por sprints. Uno de los modelos más utilizados en la industria por su simplicidad y que es el adoptado por Ecosistema Smart Campus UNIAJC es MoSCoW.

La palabra MoSCoW utiliza las siglas MSCW para referenciar frases que determinan el orden de importancia a través de la palabra deber en inglés.

M = Must Have = Debe tenerse

S = Should Have = Debería tenerse

C = Could Have = Podría tenerse

W= Won't have by now = No se tendrá por ahora

Estas siglas recuerdan que todas las historias de usuario marcadas con la letra M tienen el más alto grado de prioridad porque, de acuerdo a la visión del equipo junto con el product owner, son parte del core o núcleo del proyecto. Son las que más valor de negocio le aportan al mismo. Por tanto, si o si deben estar lo antes posible. Es decir, se verán reflejadas en los primeros sprints del proyecto. Las historias marcadas son S deberían estar una vez atendidas las de mayor prioridad. No son parte del núcleo, pero son un complemento excelente del mismo. Pueden esperar a sprints posteriores del proyecto. Las historias marcadas como C son un ideal, un plus del proyecto. Son un complemento que incrementa la satisfacción del cliente, pero no son vitales para el proyecto. Normalmente estas HU son abordadas al final del proyecto. Toda HU marcada como W indica que en la fase actual no son contempladas dentro del alcance, pero pueden marcar el camino para una nueva.

Pasos para la priorización por MoSCoW

I. Una vez se ha completado la estimación, el equipo debe volver a leer el product backlog. Esto se realiza como filtro de calidad. Se identifica si hay claridad sobre las HU o existe alguna duda y se resuelve.

II. Junto con el product owner a cada HU se le asigna un valor de acuerdo a la escala mencionada de la técnica (valores posibles M, S, C, W). Continuando con el ejemplo anterior, la tabla 5 muestra cómo se completa el product backlog con los valores de estimación y priorización.

Tabla 5. Ejemplo de product backlog estimado y priorizado sin detalle de las HU.

ID	Estimación	Priorización
HU-01	5	M
HU-02	5	S
HU-03	8	M
HU-04	13	M
HU-05	8	S
HU-06	5	S
HU-07	5	S
HU-08	5	M
HU-09	5	S
HU-10	8	M
HU-11	13	S

Continúa en la página 52

Viene de la página 51

HU-12	20	S
HU-13	5	M
HU-14	5	S
HU-15	13	M
HU-16	20	M
HU-17	8	S
HU-18	13	S
HU-19	13	M
HU-20	13	M
HU-21	13	S
HU-22	13	M
HU-23	13	M
HU-24	5	M
HU-25	5	S

Recomendaciones sobre la priorización

Dos recomendaciones muy útiles para los equipos de desarrollo que apenas empiezan a abordar la técnica son las siguientes:

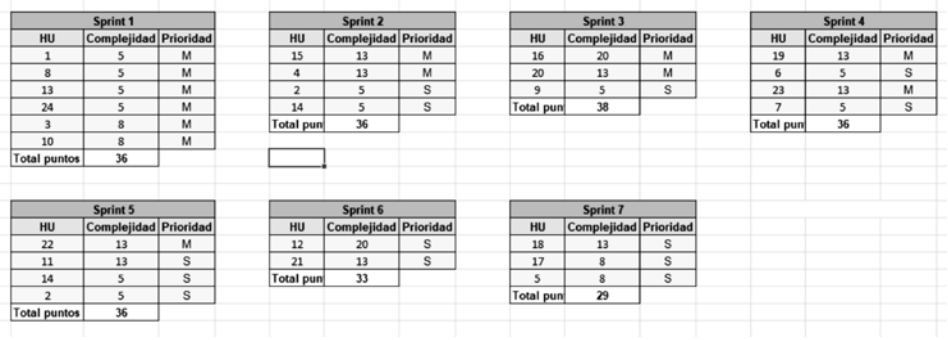
I. No olvidar preguntarle al product owner (PO) si está de acuerdo con la priorización. En algunas ocasiones el PO no podrá acompañar el equipo en la reunión de priorización, por lo que siempre es importante validar posterior a la aplicación de la técnica si está de acuerdo con el orden de atención que se ha dado.

II. No suponer que todas las HU son M. Es frecuente creer que todas las HU son importantes e inmediatas, pero la realidad es que siempre hay cosas que esperan y cosas que efectivamente son inmediatas. Tener esto presente.

Artefacto de la etapa de análisis: Sprint Backlog

La construcción de un sprint backlog supone la división del product backlog en listados más pequeños a realizar por cada sprint. La forma de realizar esta labor es bastante simple. Dado que a estas alturas ya se debe tener el tamaño del sprint, el número de sprints a realizar, el compromiso del equipo y el orden de prioridad de atención de la HU, la división se da casi en un orden natural. Siempre se atienden primero las historias de usuario M, salvo que el número de puntos de complejidad exceda demasiado el compromiso del sprint, en cuyo caso se pueden incluir

historias de complejidad S. La figura 5 muestra un ejemplo resultante de generar el sprint backlog del ejemplo que se ha venido trabajando en esta sección.



Sprint 1		
HU	Complejidad	Prioridad
1	5	M
8	5	M
13	5	M
24	5	M
3	8	M
10	8	M
Total puntos	36	

Sprint 2		
HU	Complejidad	Prioridad
15	13	M
4	13	M
2	5	S
14	5	S
Total pun	36	

Sprint 3		
HU	Complejidad	Prioridad
16	20	M
20	13	M
9	5	S
Total pun	38	

Sprint 4		
HU	Complejidad	Prioridad
19	13	M
6	5	S
23	13	M
7	5	S
Total pun	36	

Sprint 5		
HU	Complejidad	Prioridad
22	13	M
11	13	S
14	5	S
2	5	S
Total puntos	36	

Sprint 6		
HU	Complejidad	Prioridad
12	20	S
21	13	S
Total pun	33	

Sprint 7		
HU	Complejidad	Prioridad
18	13	S
17	8	S
5	8	S
Total pun	29	

Figura 6. Ejemplo Sprint Backlog

Se recomienda revisar el **anexo b** de este documento para tener un modelo a seguir para los sprint backlog.

Diseño de software

El diseño, bajo la perspectiva de Pressman & Maxim (2015), constituye una de las etapas más importantes para la correcta ejecución de un proyecto de software. Si bien, durante la etapa de análisis de requisitos se responde el interrogante; ¿qué se hará?, en la etapa de diseño se responde: ¿cómo se hará? Y para dar respuesta a este interrogante se requiere una metodología de diseño adecuada.

El proyecto Ecosistema Smart Campus, se nutre de la experticia del equipo de investigadores, quienes dada su experiencia profesional han afrontado distintos proyectos con múltiples empresas del país y del exterior. Con base en esto, se ha construido una plantilla del Documento de Arquitectura de Software o DAS, como se le conoce por sus siglas. Este es el artefacto fundamental de esta etapa y contiene todas las decisiones arquitecturales que permiten una correcta codificación del proyecto, basado en la formalización de las decisiones a través de diagramas UML.

Artefacto de la etapa de diseño: Documento de Arquitectura de software-DAS

El artefacto está compuesto de tres elementos principales que permiten organizar la información de manera correcta. A continuación, se amplía en detalle cada una.

Información de contexto

Contiene toda la información introductoria y de contexto del documento y del proyecto. Lo componen una introducción del documento, el propósito, el alcance del proyecto, las definiciones, siglas y abreviaturas, referencias y una perspectiva de vista global donde se detallan que vistas utilizará el documento para formalizar las decisiones.

Decisiones arquitecturales basadas en atributos de calidad y patrones de diseño

Está compuesto por la sección metas y restricciones, donde el equipo de desarrollo especifica los atributos de calidad (estado deseado de comportamiento del software) y los patrones de diseño que lo satisfacen (son soluciones eficientes a problemas conocidos que ya están comprobadas que funcionan de manera correcta). Es importante conocer que los atributos de calidad pueden ser tanto observables como no observables. Esto implica que si son observables puedo verlos especificados en partes del código y su representación en los diagramas UML es posible. De ser no observables, se verá en comportamientos del sistema como la adaptación de la interface gráfica a diferentes tamaños de pantalla. En esta sección se realiza una tabla por los atributos observables y una por los no observables, donde se detalla atributo de calidad a cubrir, descripción del atributo, patrón de diseño que lo resuelve y dónde veo reflejada la implementación del mismo.

La correcta selección de atributos de calidad no es tarea fácil. Aunque existen varios modelos que pueden sugerir posibles atributos recomendados para los proyectos como son la norma ISO/IEC 25000 y el modelo FURPS. Siempre es recomendado evaluar las características de cada proyecto para seleccionar cada atributo de calidad de manera adecuada y no equivocarnos por omisión. Si bien la selección de atributos de calidad es el primer paso, una vez seleccionados debemos entender que cubrir por completo el atributo por medio de patrones de diseño y luego evaluar su resultado con diversos escenarios de calidad puede resultar muy costoso. Por este motivo, siempre es aconsejable que se especifique cuales sub categorías dentro del atributo de calidad se van a cubrir dentro del proyecto.

Un ejemplo que puede ayudar al lector a asimilar más este aspecto es el atributo de calidad: *usabilidad*. Este atributo tiene una gran variedad de elementos que lo componen. Por mencionar algunos de ellos está la facilidad

de aprendizaje, la adaptabilidad, la usabilidad universal, la confiabilidad, el ser compresible, el ser intuitivo, la memorabilidad, entre otros. Cubrir todos estos aspectos es muy difícil, ya que el equipo debería entonces hacer múltiples tipos de pruebas que garanticen la implementación y satisfacción dentro de la solución. Las tablas 6 y 7 muestran el ejemplo de la tabla utilizada dentro de la plantilla.

Tabla 6. Modelo de tabla de atributos de calidad observables

Atributos de calidad observables			
Atributo de calidad	Descripción	Táctica o Patrón de diseño	Aplicación
Atributo de calidad: ·Subcategoría 1 ·Subcategoría 2 ·Subcategoría N	Descripción de las subcategorías	Patrón o patrones de diseño a utilizar para satisfacer cada subcategoría	Se especifica. En lugar del código fuente se ve aplicado.

Tabla 7. Modelo de tabla de atributos de calidad no observables

Atributos de calidad no observables			
Atributo de calidad	Descripción	Táctica o Patrón de diseño	Aplicación
Atributo de calidad: ·Subcategoría 1 ·Subcategoría 2 ·Subcategoría N	Descripción de las subcategorías	Patrón o patrones de diseño a utilizar para satisfacer cada subcategoría.	Se especifica en que momento de la ejecución se puede ver el comportamiento.

Vistas arquitecturales

De acuerdo al trabajo realizado por Kruchten (2006), existen muchos diagramas UML y, para darle un orden, se identificaron características comunes entre ellos y luego fueron agrupados de acuerdo a esas características en vistas. Las vistas están dirigidas para la comprensión específica de un público objetivo y esto facilita saber específicamente qué diagramar.

El primer paso para poder hacer correctamente diagramas UML es entender qué debemos diagramar. Aquí es altamente importante que el equipo haya elaborado las tablas indicadas anteriormente. Los atributos de calidad están especificando las características no funcionales del sistema y los patrones de diseño el cómo resolverlas. La representación de estos patrones a través de UML será lo que conforme el diseño arquitectural del sistema y modele la solución.

Vista Física

De acuerdo con Kruchten (2006), la vista física está encargada de expresar la comunicación entre dispositivos físicos y sus componentes principales. Existen varias maneras de representar con diagramas informales o diagramas UML.

En los diagramas formales a través de la notación UML, se puede representar con nodos los dispositivos físicos que intervienen en la solución, como son los dispositivos móviles o los PC de los usuarios finales, los servidores internos de la UNIAJC y los servidores externos, necesarios por consumos de APIs externas, ver figura 7.

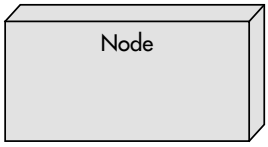


Figura 7. Representación UML de un nodo. Herramienta Visual Paradigm Versión 16

Estos dispositivos cuando permiten que se les hagan peticiones exponen puertos para la comunicación, lo que crea interfaces de comunicación habilitadas entre los nodos. Esto también tiene su representación en UML que puede ser apreciada en la figura 8.

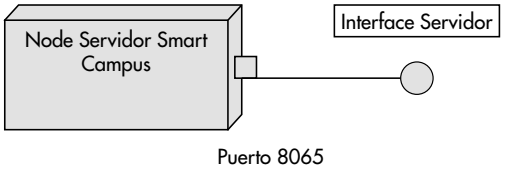


Figura 8. Representación UML de un nodo con un puerto y una interface de comunicación. Herramienta Visual Paradigm Version 16

Finalmente, existen elementos de software representados como componentes que permiten detallar aspectos como servidores de aplicaciones, donde corren las aplicaciones web o las APIs propias del proyecto, bases de datos, sistemas operativos o servicios externos, ver figura 9.

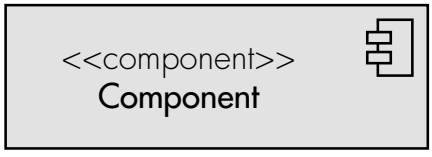


Figura 9. Representación UML de un componente. Herramienta Visual Paradigm Versión 16

La unión de estos elementos permite realizar un diagrama de nodos y componentes orientado a la vista de física. En la figura 10 se puede apreciar un ejemplo más completo sobre la aplicación de los elementos UML mencionados y trabajando en conjunto para reflejar un modelo de solución bajo las condiciones de infraestructura actuales de Smart Campus.

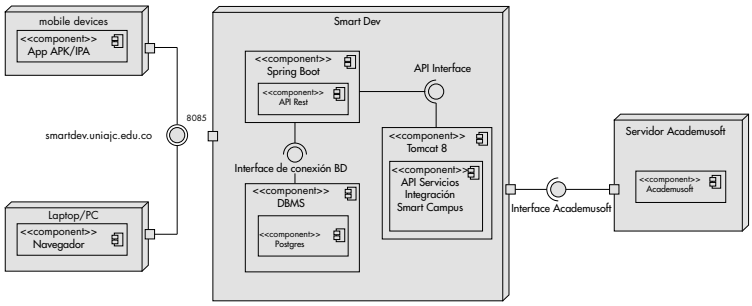


Figura 10. Diagrama de nodos y componentes modelo para proyectos Smart Campus UNIAJC

Vista Lógica

Bajo el enfoque dado por Kruchten (2006), la vista lógica revela al equipo de desarrollo las decisiones arquitecturales principales que dan estructura al aplicativo. Esto quiere decir que la mayor cantidad de patrones de diseño son expuestos aquí, en especial los que se clasifican como estructurales y de comportamiento, aunque algunos creacionales también pueden ser reflejados. Los diagramas principales que pueden ser utilizados por los equipos en esta vista son: el diagrama de clases, el diagrama de comunicación y el diagrama de secuencia. Pero existen otros diagramas como el de paquetes y componentes tanto simple como de capa nivel, que permiten dar un detalle exhaustivo de la solución. Es importante no confundir los diagramas de secuencia y de comunicación que se exponen en esta vista con los de procesos. En esta vista se detallan los elementos que componen la arquitectura y su comunicación entre sí, en cambio en la vista de proceso se detalla a nivel funcional la interacción entre las diversas funcionalidades a construir.

Con respecto a los diagramas de clase es vital comprender que la estructuración de este diagrama no debe exponer un modelo entidad relación con métodos. Por el contrario, debe exponer de manera global toda la estructuración del arquetipo a implementar. A continuación, en la figura 11 se puede apreciar un ejemplo de una arquitectura multicapa para una aplicación web a construir en Java.

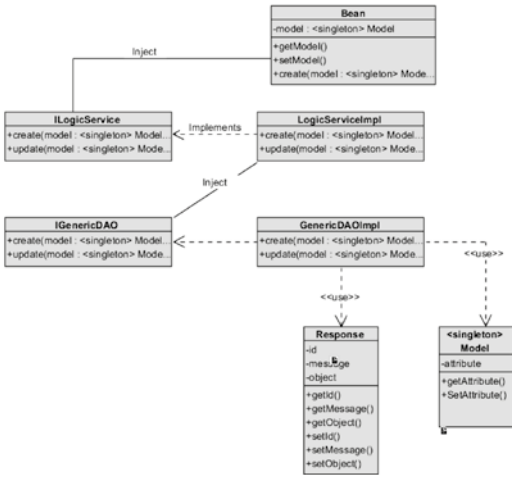


Figura 11. Ejemplo diagrama de clases. Herramienta Visual Paradigm Versión 16

Es importante destacar de la lectura correcta de un diagrama de clase para esta vista, que se debe poder dar una explicación detallada sobre los elementos que lo componen, los atributos de calidad que satisface y los patrones de diseño que implementa. Del diagrama anterior se puede apreciar que la aplicación recibe todas las peticiones a través de una pantalla y esta tiene asociado un bean. Los bean son clases específicas de Java que permiten hacer un binding de los elementos en pantalla con atributos de objetos o variables que se construyen para capturar la información. Los bean de este modelo representan la capa de presentación del aplicativo y se comunican con la siguiente capa, que es la de lógica de negocios a través de la inyección de dependencias a la interface de esa capa. La inyección de dependencias es un patrón de diseño que permite, de manera muy simple, trabajar bajo la definición de los métodos a utilizar sin tener que reservar espacio en memoria para su uso. Únicamente cuando una petición es requerida, se crea el objeto bajo demanda, se utiliza y al terminar su función se elimina de la memoria. Esto incrementa el uso eficiente de recursos del sistema, haciéndolo dinámico. Estas interfaces que son inyectadas solamente en tiempo de ejecución y bajo la condición de que se haya hecho una petición, invocan a su clase de implementación. De la misma manera, las clases que implementan la lógica usan el mismo concepto para llamar por inyección de dependencias a clases de acceso a datos. Estas clases, implementando el patrón de diseño DAO, son las únicas encargadas de administrar las peticiones a la bd. Esto independiza la capa de lógica de la capa de presentación. Haciendo un diseño arquitectural multicapa. Es relevante para la interpretación apreciar que sólo el DAO utiliza al modelo, por lo que la información que viajó encapsulada por toda la aplicación estará dentro de un objeto de transferencia de datos o

DTO. Así mismo, para no tener que crear una clase DAO por cada modelo, se ha utilizado colecciones de Java para generar una clase DAO genérica, la cual solo requiere de la entidad para ejecutar los métodos globales: si se requiere un método especifico, si se requiere la implementación de una clase propia para ese DAO. De igual forma, garantizando que las transacciones son únicas y serializadas, evitando que se sobre escriban, todos los modelos son singleton. Gracias a este modelo, se garantiza que el aplicativo es escalable, mantenible, modificable, confiable y de un rendimiento eficiente.

Otro ejemplo de diagrama a utilizar para esta vista es el de paquetes y componentes, donde el principal objetivo de los paquetes es servir de agrupador de los componentes que integran la arquitectura. En UML los paquetes se representan mediante carpetas. Ver figura 12.

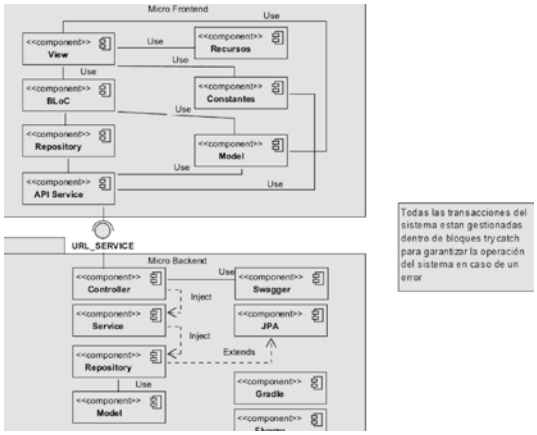


Figura 12. Diagrama de paquetes y componentes. Herramienta Visual Paradigm Versión 16

Es vital para el modelo de trabajo de Smart Campus que cada diagrama vaya acompañado de su respectiva explicación detallada, como se hizo en el caso del diagrama de clases. Se deja al lector la interpretación de la imagen 12.

Vista de Procesos

Esta vista tiene una estrecha relación con el resultado de la etapa de análisis. Aquí se describe el paso a paso funcional del aplicativo mediante la identificación correcta de los roles que intervienen, qué acciones pueden ejecutar y qué reglas de negocio aplican. Los diagramas principales que se utilizan en esta vista, de acuerdo a Kruchten (2006), son los de secuencia, los de colaboración y los de comunicación.

Dentro del Ecosistema Smart Campus, los investigadores siempre están buscando nuevas formas de aproximarse a resultados de alta calidad mediante la innovación y la creatividad aplicada. Así mediante Pastrana, Ordóñez, Ordonez, Thom, & Merchan (2018), uno de los investigadores propone un enfoque diferente utilizando Business Process Modeling (BPM o Modelado de Proceso de Negocio). Esta técnica utiliza una notación específica, llamada Notación BPM o BPMN por sus siglas en inglés. Esta forma de modelado de procesos contiene una semántica propia que permite de manera detallada definir las funcionalidades del sistema, los roles que las pueden ejecutar, las reglas de negocio, las validaciones lógicas y el tipo de tarea a ejecutar. Lo anterior ayuda a unificar la visión del proyecto debido a que BPMN solo tiene una única interpretación al ser un lenguaje gráfico y, por tanto, reduce en gran medida la ambigüedad producto de escribir las historias de usuario en lenguaje natural.

Business Process Modeling – BPM

Responde a una forma de modelado que facilita la descomposición de un proyecto en los procesos que lo componen, a fin de poder entender plenamente su complejidad, los elementos que lo componen, los roles que interactúan dentro de las funcionalidades y su comunicación entre sí. Su forma de representación es como un workflow o flujo de trabajo. Posee como restricciones que un proceso sólo puede tener un estado de inicio y uno de fin; todas las tareas llevan semántica asociada, todas las compuertas llevan semántica asociada y compuerta que abre debe ser cerrada previa al fin del modelado. A continuación, se detallan los elementos que componen la notación utilizando BPMN y la herramienta Bizagi Modeler.

Proceso

Se representa como un contenedor de todos los elementos que internamente detallarán su lógica. Es el elemento principal para iniciar el modelado. La figura 13 muestra su representación en la herramienta Bizagi.

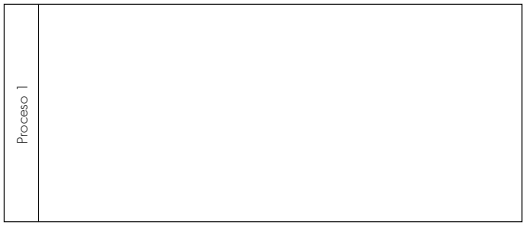


Figura 13. Representación de un proceso en BPMN. Herramienta Bizagi

En el lado izquierdo se aprecia que es donde se indica el nombre del proceso. Es importante mencionar que un proyecto puede tener tantos procesos como sea requerido en el nivel de detalle.

Lane o Carril

Representa el rol que ejecutará la acción. Un contendor de procesos puede tener tantos roles como sea necesario. Su representación modifica al contendor de proceso agregando una línea horizontal que separa el carril para indicar que esas acciones sólo son ejecutadas por ese rol. La figura 14 detalla la representación mencionada con un rol y la figura 15 con dos roles.

Figura 14. Representación Lane BPMN con un rol. Herramienta Bizagi

Figura 15. Representación Lane BPMN con dos roles. Herramienta Bizagi

Estado de inicio

Indica el inicio de un proceso. Se representa con una bola de color verde que es situada en el lane o carril que comienza la acción, ver figura 16. Sólo puede existir un estado de inicio por proceso.

Figura 16. Representación Estado de inicio BPMN con dos roles. Herramienta Bizagi

Estado de fin

Indica el final de un proceso. Se representa con una bola de color rojo que es situada en el lane o carril que finaliza la acción, ver figura 17. Sólo puede existir un estado de fin en un proceso. Asimismo sólo puede haber estado de fin si hay estado de inicio.

Figura 17. Representación Estado de fin BPMN con dos roles. Herramienta Bizagi

Tarea

Es un elemento fundamental dentro de la representación de las funcionalidades del proceso. Detalla qué acción se va a ejecutar por el rol en donde está puesta. Siempre inicia con un verbo en infinitivo, es decir, terminado en ar, er, ir. Esto se hace para indicar acción. Las tareas tienen una particularidad y es que poseen semántica que indica el tipo de acción a ejecutar, resaltando si es realizada por el usuario (ver figura 18); por el sistema (ver figura 19); si es manual (ver figura 20); si es una tarea de envío de mensaje (ver figura 21) o de recepción del mensaje (ver figura 22); si es un script (ver figura 23) o si es una regla de negocio (ver figura 24). Lo anterior se hace para dar claridad en los elementos utilizados, en la comunicación entre ellos y en la comunicación entre los roles.

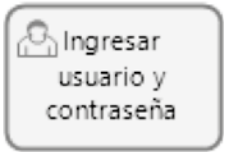


Figura 18. Tarea realizada por el usuario, notación BPMN. Herramienta Bizagi

Figura 19. Tarea realizada por el sistema, notación BPMN. Herramienta Bizagi

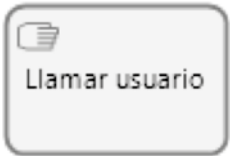


Figura 20. Tarea realizada de forma manual, notación BPMN. Herramienta Bizagi

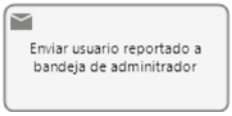


Figura 21. Tarea de envio de mensaje, notación BPMN. Herramienta Bizagi

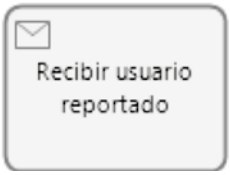


Figura 22. Tarea de recepción de mensaje en notación BPMN. Herramienta Bizagi

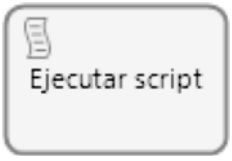


Figura 23. Tarea de ejecución script, notación BPMN. Herramienta Bizagi

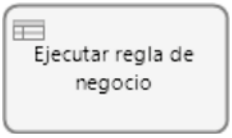


Figura 24. Tarea de ejecución regla de negocio, notación BPMN. Herramienta Bizagi

Compuerta

Las compuertas detallan una regla de negocio o validación a realizarse. Indican una divergencia o convergencia de varios caminos del proceso. A continuación, se da un breve detalle de cada una:

- **Exclusiva:** funciona como una disyunción, es decir, genera una validación donde se toma una alternativa u otra, ver figura 25.

- **Paralela:** funciona como un conector donde dos alternativas o caminos del proceso se ejecutan al tiempo, ver figura 26.

- **Inclusiva:** funciona como una disyunción, es decir, genera que dos condiciones deban ser cumplidas antes de continuar con el flujo normal del proceso, ver figura 27.

- **Compuerta basada en eventos:** en un proceso donde aparece esta compuerta, se indica que hasta no disparar un evento que la invoque, los caminos que desprenden de esta no estarán accesibles para el proceso. Ver figura 28.

- **Compuerta exclusiva basada en eventos:** funciona como un conector donde hay varias opciones con posibilidad de ser escogidos. Solamente cuando se da la selección de uno en específico, los demás caminos se inactivan y no son ejecutados. Un ejemplo simple para su uso son las opciones de menú. Aunque un menú puede tener múltiples opciones, el usuario sólo puede acceder a una al tiempo. Ver figura 29.

- **Compuerta paralela basada en eventos:** funciona como un conector donde al ser disparado por un evento varios caminos son ejecutados al tiempo. Si el evento no es iniciado, estos caminos no estarán disponibles. Ver figura 30.

- **Compuerta compleja:** se utiliza de manera convergente o divergente. Su uso radica en la complejidad de un proceso, donde no se ha podido controlar, por ejemplo, la dependencia de continuidad del proceso en la validación de 2 de 3 respuestas. Ver figura 31.

Figura 25. Compuerta exclusiva, BMPN. Herramienta Bizagi

Figura 26. Compuerta paralela, BMPN. Herramienta Bizagi

Figura 27. Compuerta inclusiva, BMPN. Herramienta Bizagi

Figura 28. Compuerta basada en eventos, BMPN. Herramienta Bizagi

Figura 29. Compuerta exclusiva basada en eventos, BMPN. Herramienta Bizagi

Figura 30. Compuerta paralela basada en eventos, BMPN. Herramienta Bizagi

Figura 31. Compuerta compleja, BMPN. Herramienta Bizagi

Subproceso

Se representa como un contenedor de proceso. Ayuda a disminuir la complejidad implícita en una visión global y sirve como punto de partida para descomponer desde una macrovisión a un detalle exhaustivo. La figura 32 muestra su representación en la herramienta Bizagi.

Figura 32. Subproceso, BPMN. Herramienta Bizagi

Evento intermedio

Se representa un evento que está en medio de un proceso, por ejemplo, una comunicación asíncrona de mensajes o una espera de tiempo requerida hasta disparar el servicio. Los dos eventos intermedios más utilizados son el de mensaje, ver figura 33 y el de espera, ver figura 34.

Figura 33. Evento intermedio de mensaje, BPMN. Herramienta Bizagi

Figura 34. Evento intermedio de espera, BPMN. Herramienta Bizagi

El ejemplo de la figura 35 ejemplifica el uso del evento intermedio de mensajes.

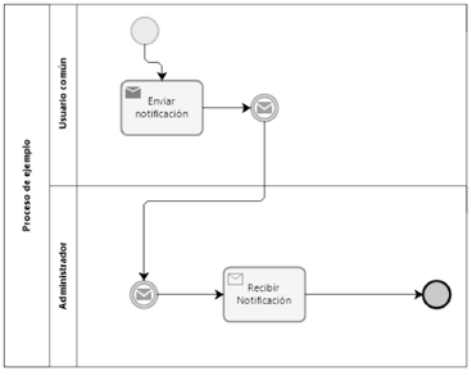


Figura 35. Ejemplo uso evento intermedio, BPMN. Herramienta Bizagi

La integración de estos componentes permite modelar a un detalle completo todos los procesos que integran un proyecto. Esta técnica permite fácilmente validar con todos los stakeholders la comprensión del proyecto y la visión del mismo, además de permitir refinarlo en caso de ser necesario.

Es muy recomendado el uso de esta vista cuando el proyecto a afrontar tiene una complejidad significativa y un número de roles que interviene mayor a 3.

Por ejemplo, en el trabajo de grado de Delgado & Gutiérrez (2020) se construyó una solución para la gestión de las solicitudes de la Facultad de Ingenierías. Una de las principales causas para esta solicitud es que el proyecto tenía mucha tarea manual con intervención de varias personas y pocas alertas tempranas de gestión y control.

Para hacer un correcto análisis de la solución y complementarlo con el diseño de la solución, el equipo de trabajo recurrió al uso del modelamiento de procesos de negocio que le permitiera comprender dónde estaban los mayores problemas y modelar una solución que optimizara de manera significativa el proceso. La figura 36 muestra un ejemplo de la vista global utilizando una gran parte de los elementos mencionados a este punto.

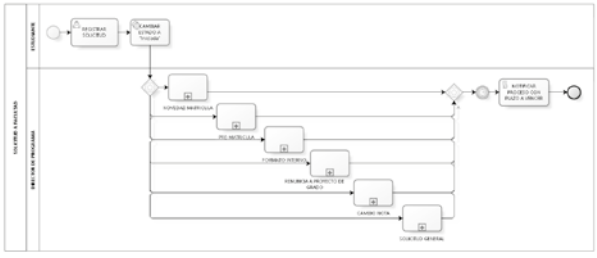


Figura 36. Macro proceso solución sistema de solicitudes Facultad de Ingenierías. Modelado con Bizagi. Fuente: proyecto de grado Delgado & Gutiérrez (2020)

Vista de Despliegue

De acuerdo con Kruchten (2006), la vista de despliegue es muy similar a la vista física. Ambas se encargan de expresar la comunicación entre dispositivos físicos y sus componentes principales. Pero su principal diferencia radica en el detalle interno que tienen los componentes principales de la aplicación. En este diagrama es mucho más detallada la estructuración y se puede percibir una mayor cantidad de patrones de diseño implementados. La figura 37 da un claro ejemplo de esta vista.

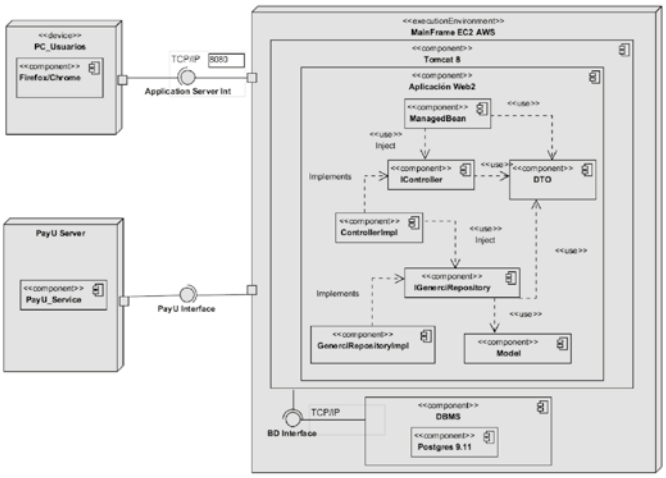


Figura 37. Ejemplo diagrama de despliegue vista de despliegue. Herramienta visual paradigm versión 16

Se recomienda al lector revisar el anexo c adjunto a este documento, para tener una guía más completa de la plantilla utilizada dentro de los proyectos.

Desarrollo de software y calidad

Esta etapa permite materializar el *¿qué se hará?*, descrito en el análisis de requisitos y el ¿cómo se hará?, especificado en el diseño de la solución mediante el código fuente. Aquí, el scrum development team, de manera organizada y de acuerdo con la planeación, trabaja de manera iterativa incremental para construir unidades de código que permiten hacer un entregable 100% funcional. Estas unidades desplegables conforman una parte del proyecto que, de sprint a sprint, suman al total de compromisos, resolviendo las HU de cada fase. La base fundamental para que el equipo pueda concentrarse únicamente en programar son las decisiones tomadas en el diseño. Gracias a esas decisiones, el equipo de desarrollo sabe exactamente a qué acudir dentro del lenguaje de programación, utilizando el máximo potencial de los frameworks disponibles.

Si las fases de análisis y diseño han sido ejecutadas adecuadamente, se minimiza el reproceso producto de dudas o de decisiones no tomadas a la hora de programar, incrementando la productividad del equipo de desarrollo. Claro está, esto no es suficiente para garantizar que la calidad de cada entregable sea la más alta posible, como indica Pastrana et al. (2019), y por ello, se hace necesario generar una serie de mecanismos y prácticas que resalten la mejora continua producto de la calidad preventiva aplicada sprint a sprint.

Tradicionalmente los proyectos de desarrollo de software separan la etapa de construcción del proyecto de la de calidad, como indica el enfoque inicial de Pressman & Maxim (2015), en el que primero se construye el producto y luego se revisa que tanta calidad tiene para someterlo a un proceso de correcciones y revisiones hasta que cumpla con el nivel aceptado por el revisor. Este enfoque puede resultar costoso dependiendo de la cantidad de reproceso a asumir y la cantidad de tiempo disponible en el cronograma de actividades pactadas con el product owner. Siguiendo el otro enfoque, el agilista, la calidad y el desarrollo deben ir de la mano en todo momento, como lo indica Beck (1999).

Desde la visión de los 12 principios y los 4 valores de Fowler & Highsmith (2001), lo deseable para todo proceso de desarrollo de software es tener un perfecto equilibrio entre los compromisos y la vida personal, esto sólo es posible alcanzarlo si existe un ritmo sostenible de desarrollo de software, donde los mecanismos de alerta temprana permitan al equipo identificar sus buenas y malas prácticas de desarrollo, logrando así, sprint a sprint, mejorar su forma de trabajo y obtener un resultado de calidad más alto y con menor reproceso. Enfoque que coincide con el de Pastrana et al. (2019), quienes centrados en este ideal elaboran una identificación de las principales buenas prácticas de desarrollo detalladas por Beck (1999), pero dándole forma a través de herramientas específicas para poder estandarizar el proceso de desarrollo de software, propio de Smart Campus.

Buenas prácticas de desarrollo de XP adoptadas

Acorde al trabajo realizado por Pastrana et al. (2019), Extremme Programming o XP, como comúnmente se le conoce, describe 12 buenas prácticas con el ánimo de que los equipos de desarrollo puedan obtener información rápida de cómo están llevando la construcción de su aplicativo y puedan corregir a tiempo posibles brechas que con el paso del tiempo se incrementen y terminen siendo un problema para el producto ya puesto en producción. Según Beck (1999), las buenas prácticas recomendadas son:

1. El juego de la planificación: para todo equipo de desarrollo es fundamental realizar una correcta planeación de su trabajo, por lo que es fundamental tener claro los compromisos por entrega, las responsabilidades de cada uno de los integrantes del equipo y los objetivos a cumplir de acuerdo a los roles que desempeñen los integrantes.

2. Entregas pequeñas: se debe trabajar por periodos de tiempo controlados no mayores a 4 semanas. En concordancia con la definición de scrum a esto se le denomina sprint y tiene como objetivo poder controlar de manera eficiente la evolución del producto.

3. Metáfora: una metáfora es una explicación de algo por medio de una representación diferente con la que guarda una relación. Las abreviaciones o nomenclaturas pactadas en un proyecto entre todos los integrantes del equipo son un ejemplo de esto. Para aclararlo un poco más, por ejemplo, el equipo de desarrollo puede indicar que tbl significa tabla y crear esta representación como algo común con el equipo de soporte del proyecto. Esta práctica puede llegar a ser muy difícil de implementar.

4. Diseño simple: arquitecturalmente un proyecto debe mantener el diseño de su solución lo más simple posible para poder controlar los cambios estructurales, de comportamiento o creacionales de una manera más eficiente y mantener en orden el crecimiento evolutivo del proyecto.

5. Refactoring: todo proceso de desarrollo de software tiene implícito algún grado de reproceso, producto de malas prácticas o de errores humanos. Por lo anterior, es necesario tener presente siempre un tiempo adecuado para realizar estas correcciones o mejoras posibles que permitan disminuir la deuda técnica.

6. Pair Programming o programación en parejas: es una práctica poco utilizada debido a la complejidad de la misma. Se utiliza principalmente cuando queremos nivelar la curva técnica del equipo al tiempo que supervisamos la calidad del entregable. En esta práctica dos personas del equipo trabajan juntos la misma historia de usuario. Primero, uno de los integrantes toma el control del desarrollo y empieza la codificación, a esta persona se le denomina navegante, ya que es quien está directamente afectando la unidad de código. Su acompañante es quien aprende del trabajo que este está realizando y le ayuda a comprobar la concordancia de lo que se escribe a nivel de código frente al documento de arquitectura de software y a los criterios de aceptación de la historia de usuario. Esto permite que, al tiempo, la unidad de código sea puesta a prueba constantemente.

7. Propiedad colectiva: tanto la información del proyecto, como el código fuente debe estar disponible en todo momento para cualquiera de los

integrantes del equipo. Por lo anterior, deben existir mecanismos que permitan garantizar esta práctica en todo momento.

8. Integración y despliegue continuo: esta práctica apoya al equipo como una alerta temprana que permite identificar si el incremento que se está realizando afecta de manera negativa a la unidad desplegable generando impedimentos para alcanzar los objetivos del sprint. Aquí las herramientas que se implemente propenden por identificar de manera inmediata si se ha producido un error sintáctico o técnico que impida avanzar y notificarlo de manera inmediata para que todo el equipo esté enterado y tome acciones correctivas de manera inmediata.

9. Semana de 40 horas: un equipo cansado es un equipo que empieza a generar errores de manera inconsciente. Esto crea no sólo problemas de rendimiento sino también de calidad. Por lo anterior, no se recomienda que un equipo esté dedicado a un proyecto más de 40 horas por semana.

10. Cliente in situ: esta práctica invita a que, si es posible, se trabaje lo más cerca posible del cliente, preferiblemente en el mismo lugar, así se garantiza la comunicación constante y cara a cara. En caso de no ser posible, al menos se recomienda tener contacto constante con el product owner del proyecto.

11. Estándar de programación: el secreto de mantener la misma visión de desarrollo es que todo el equipo hable el mismo idioma. Por lo anterior, no se debe trabajar un proyecto de desarrollo sin haber construido ese idioma común que represente el nombrado de clases, variables, constantes, objetos, tablas, etc.

12. Pruebas: representan la comprobación de que tanto funcional como técnicamente el software está listo para responder a las necesidades del proyecto. Por lo anterior, el equipo de desarrollo debe generar todos los mecanismos necesarios para garantizar el sello de calidad de manera preventiva y no correctiva. Mecanismos como las pruebas unitarias, pruebas de integración, revisiones por pares, son obligatorias antes de determinar que una historia de usuario ha sido finalizada.

Si bien, de acuerdo a la explicación de este mismo capítulo, algunas de estas prácticas recomendadas por Beck (1999), como son *el juego de la*

planificación, entregas pequeñas, diseño simple y pruebas ya son tomadas en cuenta dentro del proceso de scrum, descrito por Schwaber & Sutherland (2017), existen otras que resultan en un complemento ideal que ayuda a reforzar el proceso de aseguramiento de calidad mediante el aprendizaje y la mejora continua, como indica Pastrana et al. (2019). Por ejemplo, *la propiedad colectiva del código* toma alta relevancia para los equipos de desarrollo de software mediante su implementación a través de un versionador de código. Esta herramienta permite administrar en un repositorio central para todo el equipo, todos los cambios generados producto de la evolución del entregable. Aquí el histórico del código fuente es importante no sólo para identificar la trazabilidad histórica, sino también para garantizar que el código fuente está disponible para todo el equipo de trabajo en cualquier momento. Adicionalmente, brinda la oportunidad de recuperar rápidamente un cambio errado que afecte el código de todos los participantes simplemente con devolverse a la versión anterior estable.

Otra de las prácticas que toma mucha importancia, según Pastrana et al. (2019), es *la integración continua*. Las herramientas de integración continua permiten generar junto con el versionador de código fuente una sinergia ideal para revisar si los cambios evolutivos del código están siendo guardados sin problemas sintácticos o si, por el contrario, estos cambios están afectando la unidad desplegable. Una alerta temprana de este tipo ayuda al equipo a controlar la evolución del aplicativo de manera adecuada, impidiendo que pase mucho tiempo entre la construcción de una funcionalidad y su empaquetado. Así se identifica de manera inmediata si los cambios son adecuados o no y el equipo entra a corregir de manera eficiente. En complemento a lo anterior, una forma adecuada de conservar una mantenibilidad ideal del proyecto es tener *estándar de código* definido para el equipo de trabajo. Esto da lugar a una única forma de escribir el código, a la vez que la curva de aprendizaje está asociada únicamente a entender la lógica implícita en la funcionalidad y no a entender la sintaxis con la que fue construida. Los proyectos de Smart Campus para hacer uso de esta práctica implementan un analizador estático de código que garantice el cumplimento del estándar internacional de codificación, asociado a las buenas prácticas de programación conocidas para los lenguajes orientados a objetos más comunes. Esta herramienta también va de la mano del versionador de software y del integrador continuo.

Lo anterior busca como objetivo, mantener un ritmo sostenible bajo una jornada eficiente de máximo 40 horas a la semana.

Ambiente de calidad preventiva

Con base en lo anterior, se puede identificar claramente que el objetivo de Smart Campus dentro de la etapa de desarrollo y calidad, es hacer lo más eficiente este aspecto mediante la implementación adecuada de un ambiente de calidad preventiva que entregue al equipo de desarrollo alertas tempranas de mejora y que, a su vez, le permita al equipo mejorar continuamente su cultura de desarrollo.

Alineado el trabajo de Pastrana et al. (2019)Springer Nature Singapore Pte Ltd. Compliance with the client requirements is a key factor for the success of projects. However, this compliance is not always easy to monitor and verify. SCRUM framework flexibility makes it possible to include a variety of techniques and good practices from other methods to help software development teams to achieve their goals. Some of these tools can track historical changes in the code (versioning con Pastrana, Ordóñez, & Cobos (2019), un esquema de calidad preventiva adecuado debe estar orientado al modelo DevOps, que permite garantizar que en todo momento la evolución y operación del proyecto está controlado y medido a través de mecanismos que implementen las buenas prácticas mencionadas. Así, el equipo de trabajo diseña un ambiente compuesto por un versionador de código, un integrador continuo y un analizador estático de código, ver figura 38.

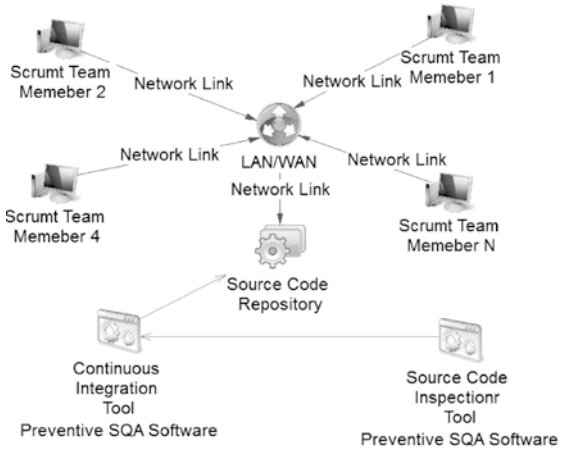


Figura 38. Esquema ambiente de calidad preventiva

Cada vez que uno de los integrantes del equipo de desarrollo suba un cambio al versionador, el integrador continuo detectará este cambio del histórico. Al saber que hay una nueva versión, utilizando el esquema configurado para el proyecto,

primero deberá correr las pruebas unitarias y, si todas salen bien, procederá a generar la unidad desplegable. Esto se hace con el ánimo de garantizar que existe una mínima revisión de calidad previa por parte del desarrollador de software. Si el intento de generar la unidad desplegable falla, la herramienta notifica al equipo que ese cambio subido presenta un error y que debe ser corregido a tiempo. En caso contrario, notifica al analizador estático de código que puede hacer su revisión.

Actualmente, el versionador utilizado por Smart Campus es gitLab, el analizador estático de código es Sonar y el integrador continuo es Jenkins.

Despliegue

El Ecosistema Smart Campus cuenta con el apoyo institucional del departamento de tecnologías DITIC. Gracias al apoyo, el proyecto logra tener un espacio dentro del data center de la institución y hacer la adquisición de un servidor Dell PowerEdge 2900 con las siguientes características:

- Disco duro: 280 GB
- Memoria RAM: 8 GB
- Procesador: Intel Xeon(R) CPU E5420 @ 2.50GHz Max Vel 3600 MHz de cuatro núcleos
- Sistema operativo: Centos 7
- Dominio del servidor: smartdev.uniajc.edu.co

Este servidor es puesto a disposición de todos los proyectos de desarrollo de software que así lo requieran, para realizar actividades de pruebas y simulaciones de despliegue final que permitan garantizar, desde el sprint 0 de cada proyecto, que no habrá inconvenientes a la hora de la puesta en marcha de cada aplicativo. Los siguientes servidores de aplicaciones y base de datos se encuentran dentro de este equipo:

- Apache tomcat 7.0.59
- Apache tomcat 8.0.9
- Apache tomcat 8.5.30
- PostgreSQL 9.5
- PHP 5.6.40

- Java 1.8.0_222
- Nodejs 9.11

Así mismo, el ecosistema cuenta con un ambiente final de producción, servidor denominado smartcampus, que es administrado por DITIC y en donde se despliegan las aplicaciones producto de los resultados de los diversos proyectos. Para el despliegue de las aplicaciones, todas han pasado por un proceso de calidad preventiva y mejora continua, como se describió en la sección anterior, antes de ser aprobadas por el equipo de desarrollo de Smart Campus.

Dentro de este servidor se ha generado la configuración de ambientes de despliegue a partir del uso de contenedores docker. Los contenedores funcionan dentro del sistema operativo aprovechando sus características, pero creando un entorno aislado dentro del él que permite realizar instalaciones y parametrizaciones correspondientes al despliegue, que por fuera de ese contexto no interfieren con otras configuraciones que estén dentro del sistema operativo. De esta manera, fácilmente podemos poner a cohabitar distintas versiones de un lenguaje de programación sin mayor problema, como por ejemplo tener java 7, java 8, php 5 y php 7, todos en la misma máquina sin configuraciones complejas y aisladas unas de otras, pero con la posibilidad de que las aplicaciones interactúen entre sí, de ser requerido.

Así, actualmente el servidor marca HP ProLiant DL120 Gen9 cuenta con las siguientes características:

- Disco duro: 2 discos duros de 1 TB cada uno.
- Memoria RAM: 8 GB DDR4 FB-DIMM con velocidad 2400 MT/s
- Procesador: Intel(R) Xeon(R) CPU E5-2603 v4 @1.70GHz Max Vel 4000 MHz de seis núcleos
- Sistema operativo: Centos 7
- Dominio del servidor: smartcampus.uniajc.edu.co

Las aplicaciones para despliegue instaladas principalmente en esta máquina son:

Apache tomcat 7.0.88 - port-http:8888/ port-http:8443
PostgreSQL 9.5
Java 1.8.0_222
Apache/2.4.6 httpd

Los contenedores desplegados dentro de esta maquinan son:

- Php7
- Php56
- SonarQube
- GitLab
- Jenkins
- MySql

Proceso de gestión de proyectos Ecosistema Smart Campus

La gestión de proyectos dentro del Ecosistema Smart Campus se da orientada por algunos de los lineamientos y recomendaciones del PMBOK (2017), donde los elementos principales del control y seguimiento de la información se encuentran dados por:

- Acta de inicio: artefacto que delimita el alcance del proyecto, determina las personas involucradas y fija el acuerdo de entregas por sprint, para tener un cronograma tentativo a seguir, salvo una eventualidad justificada que por mutuo acuerdo permita mover una fecha de entrega.
- Acta de confidencialidad: documento de aspecto legal que detalla los compromisos adquiridos legalmente, sobre la responsabilidad del acceso y la manipulación de la información presente en el proyecto.
- Matriz de riesgos: artefacto que permite la correcta identificación de riesgos desde el inicio del proyecto y determina las posibles formas de impedirlo o mitigarlo. Es un documento dinámico que sprint a sprint debe estar siendo alimentado para mantener un seguimiento sobre posibles eventualidades.
- Matriz de interesados: artefacto que detalla la información de contacto de las personas involucradas en el proyecto. Sirve de mecanismo de identificación del canal de comunicación y siempre está disponible para todos los interesados.
- Seguimiento plan o informe diario del proyecto: artefacto que permite evidenciar el avance del proyecto sprint a sprint, determinando si se está avanzando a la velocidad deseada y conforme a las metas del sprint. El cálculo de la velocidad del equipo se da mediante la métrica burndown chart, característica del marco de trabajo ágil scrum.