La representación del diseño detallado utilizando plantillas y sus efectos en la calidad del software

Silvana Moreno[1], Vanessa Casella[1], Martín Solari[2], and Diego Vallespir[1]

[1] Universidad de la República, Uruguay
{smoreno,vcasella,dvallesp}@fing.edu.uy
[2] Universidad ORT Uruguay
{martin.solari}@ort.edu.uy

Resumen El diseño de software es una práctica que busca contribuir en la construcción de software de calidad. Durante el diseño se traducen los requisitos en una representación del software cuya calidad puede ser evaluada. Para los estudiantes de pregrado, el diseño es difícil de entender y realizar. De hecho, construir un buen diseño parece requerir de un cierto nivel de desarrollo cognitivo que pocos estudiantes alcanzan. El objetivo de este trabajo es conocer el efecto en la calidad del software cuando estudiantes de pregrado utilizan plantillas para representar el diseño detallado. Conducimos un estudio empírico donde los estudiantes desarrollan 8 proyectos siguiendo un proceso definido y registrando datos de la ejecución del mismo en una herramienta. Encontramos que el uso de plantillas de diseño no mejoró la calidad del código medido como la densidad de defectos en la fase de prueba unitaria. Tampoco el uso de plantillas logró reducir la cantidad de *code smells* presentes en el código analizado.

Palabras claves: diseño detallado · calidad de diseño · estudiantes

1 Introducción

El diseño de software es uno de los componentes más importantes para asegurar el éxito de un sistema de software [13]. Este contiene dos actividades entre el análisis de requisitos y la construcción del software: el diseño arquitectónico y el diseño detallado. Durante el diseño arquitectónico se estructuran e identifican los componentes de alto nivel. Durante el diseño detallado se especifica cada componente en detalle [3]. Este trabajo está enfocado en el diseño detallado.

El diseño es una disciplina difícil de comprender para los estudiantes de pregrado, y el éxito (i.e. construir un buen diseño) parece requerir un cierto nivel de desarrollo cognitivo que pocos estudiantes alcanzan [6, 13, 19]. La capacidad de los estudiantes para construir un buen diseño está relacionada con la capacidad de abtracción, comprensión, razonamiento y procesamiento de la información [17, 18, 26].

Construir software de calidad es cada vez más importante. Dependemos del software en nuestra vida diaria y, cada vez más, dependemos de su correcto

funcionamiento. Un diseño de software de calidad permite construir software de calidad; con menos defectos y más mantenible. Los profesionales de la industria son conscientes de la importancia de la calidad del diseño de software y, entre otros, utilizan prácticas de *clean code* (código limpio), revisiones y herramientas para contribuir en este sentido [4, 11, 29].

Conocer cómo diseñan los estudiantes de pregrado es de interés de varios autores [7–9, 20, 30]. La mayoría de sus estudios encontraron que los estudiantes no logran producir un buen diseño de software. Algunos de los problemas detectados son la falta de consistencia entre los artefactos de diseño y el código, diseños incompletos y la falta de entendimiento de qué tipo de información incluir al diseñar software [8, 9, 20].

En este trabajo estudiamos la práctica del diseño de software en estudiantes de pregrado, que ya han aprobado cursos de programación y diseño. Conducimos un experimento en el contexto de cursos con el objetivo de estudiar el efecto en la calidad del software que tiene la representación del diseño utilizando plantillas específicas.

El documento se estructura así: la sección 2 plantea los trabajos relacionados; las secciones 3 y 4 presentan el método experimental; la sección 5 presenta los resultados y en la 6 se realiza su discusión; las amenazas a la validez se mencionan en la sección 7; y la sección 8 presenta las conclusiones y trabajo a futuro.

2 Trabajos relacionados

El diseño de software es una actividad importante para garantizar la calidad de un sistema de software [13, 24]. Implica identificar y describir de forma abstracta el sistema de software y sus relaciones. Buenos diseños ayudan al desarrollo de software robusto, mantenible y con pocos errores [23, 28]. El diseño detallado de software es una actividad creativa, que puede realizarse de distintas formas: en la mente del desarrollador de software, en un bosquejo en papel, mediante diagramas, utilizando lenguajes o herramientas tanto formales como informales. El estándar internacional ISO 9126 propone un modelo con métricas de calidad de software internas y externas. Las internas son aquellas que no dependen de la ejecución del software (medidas estáticas), mientras que las externas son aquellas aplicables al software en ejecución [15].

En los últimos años, el uso de prácticas de *clean code* y de herramientas ha contribuido en la mejora de la calidad del diseño [29]. Los *code smells*, *anti patterns* y *design flaws* son algunos de los indicadores utilizados actualmente para medir la calidad del diseño [4, 11]. SonarQube [5] y FindBugs [2] son algunas de las herramientas utilizadas para medir la calidad del código a través de la detección de "*bad smells*".

Las prácticas actuales de la industria hacen necesario que los graduados tengan la capacidad de entender y de construir diseños de software. Sin embargo, los estudiantes de pregrado tienen dificultades para diseñar. Construir un buen diseño requiere un cierto nivel de desarrollo cognitivo que pocos estudiantes alcanzan [6, 13, 19]. Estos desarrollos cognitivos tienen que ver con la capaci-

dad para lograr dimensionar el software, la capacidad de reconocer patrones de diseño, estilos y datos relacionados, así como la capacidad lógica y deductiva para descomponer el sistema en subsistemas y componentes [13].

De hecho, para los estudiantes aprender a diseñar es más difícil que aprender a programar. Esta dificultad ocurre porque para la mayoría de los lenguajes de programación, los estudiantes obtienen retroalimentación del compilador y errores en tiempo de ejecución. Sin embargo, esto no ocurre con el diseño [16].

El diseño orientado a objetos (OO) es uno de los enfoques de diseño más utilizados en la industria y uno de los temas que normalmente se enseña en las universidades [10]. Los diagramas y lenguajes de modelado OO permiten modelar aspectos estáticos y dinámicos del sistema. Varios estudios empíricos analizan la comprensión y los beneficios del uso de diagramas utilizando el lenguaje de modelado UML [1, 12, 31]. Los diagramas UML ayudan a entender el diseño y a mantener el código fuente.

Sin embargo, en varios estudios, los estudiantes de pregrado no logran obtener beneficios en el diseño utilizando estos diagramas [12, 31]. Gravino encuentra que los estudiantes que utilizan diagramas UML para diseñar no logran mejoras significativas en tareas de comprensión de código fuente comparado con los estudiantes que no lo utilizan. Además, los estudiantes que utilizan diagramas usan el doble del tiempo en la misma tarea de comprensión de código fuente respecto a los estudiantes que no los utilizan. Al analizar el factor experiencia, encuentra que los estudiantes mas experimentados logran una mejora en la comprensión del código fuente [12, 27].

Para los profesionales de la industria el uso de UML continúa siendo una resistencia [29]. Una encuesta realizada a 50 profesionales del software indica que aunque la calidad del software es un aspecto importante, el uso de UML es selectivo (informal, sólo por un tiempo, luego se descarta) y con baja frecuencia [22].

Las habilidades de diseño que tienen los estudiantes de pregrado son reportadas por un conjunto de estudios que examinan los artefactos producidos por estos para conocer cómo diseñan software [7–9, 20, 30]. Estos estudios utilizan la misma especificación de requerimientos con diferentes enfoques: diseños producidos de forma individual, diseños realizados de forma grupal y diseños producidos en diferentes niveles de formación. En general, los autores coinciden en que los estudiantes de pregrado no son capaces de diseñar un sistema de software. La falta de consistencia entre los artefactos de diseño y el código, los diseños incompletos y la falta de entendimiento de qué tipo de información incluir al diseñar software son algunas de las dificultades reportadas [8, 9, 20].

En un trabajo anterior [21], estudiamos cuánto esfuerzo dedican los estudiantes al diseño de software y cuál es su percepción respecto al diseño y a la enseñanza de esta disciplina durante la carrera de grado en computación de la Universidad de la República. El análisis realizado muestra que los estudiantes dedican al menos 4 veces más de tiempo a la codificación que al diseño de software. Además, pocos estudiantes se dan cuenta que no dedican el tiempo suficiente en etapas tempranas del desarrollo.

Creemos, al igual que Loftus et al. [20], que los estudiantes no saben qué hacer cuando tienen que diseñar software. Además, varios autores analizaron los artefactos producidos y coinciden en que los estudiantes no saben diseñar [7–9, 20, 30]. Esto motivó el trabajo presentado aquí; que propone brindar a los estudiantes plantillas de diseño como herramienta de apoyo a la representación del diseño. A diferencia de Gravino y Torchiano que analizan los beneficios del uso de diagramas en la comprensión del código [12, 31], nuestro enfoque analiza el efecto del uso de plantillas en la calidad del software. Analizamos la calidad desde el punto de vista de los defectos y de los *code smells* en el código. El foco de nuestra investigación es el diseño orientado a objetos y detallado (a nivel de clase y componente).

3 Método

Nos interesa conocer el efecto del diseño en la calidad del software cuando se exige a los estudiantes de pregrado la representación del diseño utilizando un conjunto específico de plantillas. Para ello, llevamos adelante tres experimentos en el contexto de un curso de la carrera de grado en computación de la Universidad de la República, durante los años 2015, 2016 y 2017. En los experimentos participaron estudiantes avanzados de la carrera de grado de informática de la Facultad de Ingeniería, que ya tenían aprobados los cursos donde se enseña diseño detallado de software: principios de diseño, artefactos y diagramas de diseño, UML, patrones de diseño, etc.

3.1 Contexto del curso

El curso tiene todos los años el mismo formato. Comienza con dos clases teóricas donde se enseña el proceso de desarrollo que van a utilizar en el curso (que denominamos proceso base) y se explica la dinámica del trabajo práctico. El proceso base es un proceso definido y disciplinado que ayuda a realizar mejor el trabajo. El trabajo práctico consiste en que cada estudiante desarrolle 8 proyectos siguiendo el proceso base y registrando datos de la ejecución del proceso en una herramienta. Los estudiantes realizan los proyectos de forma individual y consecutiva.

Los proyectos son asignados semanalmente a los estudiantes por parte de un docente tutor. La asignación consiste en el envío de los requerimientos del proyecto. La entrega de cada estudiante consiste en el código que resuelve el problema, los casos de pruebas ejecutados y los datos registrados en la herramienta. Los estudiantes realizan los proyectos en sus casas y tienen un docente asignado que será el responsable de asignar, corregir y evacuar de cada proyecto. Antes de comenzar el primer proyecto cada estudiante debe elegir el lenguaje de programación a utilizar a lo largo del curso. Nuestro interés es recolectar datos de la ejecución del proceso de desarrollo con el uso de un lenguaje que el estudiante domina y no datos del aprendizaje de un lenguaje.

Las fases del proceso, los scripts y el registro de datos (logs) se presentan en la figura 1. El proceso base consiste de las siguientes fases: planificación, diseño, codificación, compilación, pruebas unitarias (PU) y postmortem. Para seguir el proceso base se brinda un conjunto de scripts. Los scripts son una guía que establece las entradas, salidas y actividades a realizar en cada fase. Los scripts ayudan al estudiante a encaminar las actividades del desarrollo pero sin exigir cómo deben realizarse. Por ejemplo, el script para la fase de diseño, tiene como entrada los requerimientos y establece como actividad central producir un diseño que satisfaga los mismos (no se establece cómo debe producirse ni cómo debe representarse dicho diseño). Los datos que se registran son el tiempo dedicado a cada fase del proceso, los defectos detectados y removidos en cada fase y el tamaño en locs del programa construido.

Figura 1. Proceso base, scripts y logs

Los proyectos son todos de naturaleza matemática, de baja y similar complejidad. Los percentiles 5 y 95 de los datos recolectados para todos los estudiantes a lo largo de los 8 proyectos son 26 locs y 242 locs respectivamente. La fase de diseño se refiere al diseño detallado; i.e. identificar clases, atributos, operaciones, escenarios del programa (o componente), cambios de estado y/o pseudo-código. Cada experimento se corresponde con una instancia del curso y los estudiantes que participan de una instancia no vuelven a participar de una instancia posterior.

3.2 Objetivos y diseño experimental

Los experimentos tienen como objetivo conocer el efecto en la calidad del software cuando los estudiantes representan su diseño utilizando un conjunto específico de plantillas de diseño. La completitud de las cuatro plantillas (Operacional, Funcional, Lógica, Estado) permiten describir el diseño de forma completa y precisa [14]. Las plantillas son documentos con una estructura predefinida en las cuales los estudiantes deben representar el diseño realizado.

Las plantillas permiten describir desde la operación del sistema (Operacional) hasta el seudocódigo de cada método (Lógica). A continuación se presenta una breve descripción de cada una.

– Plantilla operacional: especifica la interacción del sistema y los usuarios. El contenido puede verse como similar a los casos de uso.

– Plantilla funcional: se especifica el comportamiento de las invocaciones y retornos del programa (variables, funciones, clases y métodos). La figura 2 presenta un ejemplo de uso de esta plantilla para una clase del proyecto 6.
– Plantilla lógica: en esta plantilla se registra el seudocódigo de cada método que aparece en las plantillas funcionales.
– Plantilla de estado: define las transiciones y las condiciones de los estados internos del programa. El contenido es similar a los diagramas de máquinas de estado.

Student	Student X	**Date**	07/05/17
Program	Problema 6	**Program #**	
Instructor	Instructor Y	**Language**	Java

Class Name	CalculoValorX
Parent Class	

Attributes

Declaration	**Description**
P:double	Valor de la integral.
Dof:int	Grados de libertad.
XIni: double	X inicial a probar(se toma 1,0)
DebeAjustar:boolean	Dice si se debe ajustar el valor de d
EsMenor:boolean	Dice si el valor de la integral calculada con el nuevo x es menor al valor de p.
Xprueba:double	Valor de x a probar.

Items

Declaration	**Description**
calcularX(p:double, dof:int, xIni:double)	Se encarga de calcular el valor de x, la primera vez usando Simpson y en caso que esta primera vez no sea la correcta usando el método recursivo recalcularX
ajustarD(debeAjustar:boolean, esMenor:boolean)	Ajusta el valor de d. Si es menor true entonces el valor retornado es positivo, en caso contrario es negativo. Si debeAjustar, entonces d = 0,5/2, de lo contrario es 0,5
recalcularX(p:double, dof:int, xPrueba:double, esMenor:boolean)	Se encarga de realizar el cálculo de x recursivamente hasta encontrar el valor correcto.

Figura 2. Plantilla Funcional

El diseño de este experimento es de medidas repetidas y de un factor (la representación del diseño de software) con dos alternativas: utilizando plantillas y sin utilizar plantillas. Las variables de respuesta consideradas en este experimento son la calidad interna y externa del software. Para evaluar la calidad externa consideramos la densidad de defectos en la fase de pruebas unitarias del proceso base. Es decir, se cuentan la cantidad de defectos encontrados en dicha fase y se divide entre las locs del proyecto. Conocer la densidad de defectos en esta fase del proceso de desarrollo da una idea de los defectos remanentes en el software[25]. Para evaluar la calidad interna analizamos los *code smells* en los que incurren los estudiantes. Conocer la cantidad de *code smells* presentes en el

código fuente del producto nos da una idea de los costos de mantenibilidad a futuro [11].

4 Operación

Los experimentos se ejecutaron en 2015, 2016 y 2017 y participaron 25, 17 y 19 estudiantes respectivamente. Todos los estudiantes aplican el proceso base en los primeros cuatro proyectos y registran sus datos en la herramienta. Durante los primeros cuatro proyectos la fase de diseño no exige la entrega de la representación del diseño. Una vez finalizado el proyecto 4 se divide a los estudiantes en dos grupos mediante un sorteo; denominamos los grupos como "con representación de diseño utilizando plantillas" (*conPRD*) y "sin plantillas para la representación de diseño" (*sinPRD*). El cuadro 1 presenta el diseño del experimento.

Cuadro 1. Diseño experimento

Grupo	**proyecto 1 to 4**	**proyecto 5 to 8**
conPRD	proceso base	proceso base + rep. diseño con plantillas
sinPRD	proceso base	proceso base

El grupo *sinPRD* es el grupo de control, que continúa aplicando el proceso base a lo largo de los proyectos 5 al 8. El grupo *conPRD* asiste a una clase donde se presentan las cuatro plantillas que permiten representar el diseño; que deben utilizar para representar su diseño a partir del ejercicio 5 y hasta el 8. La entrega de la representación del diseño utilizando plantillas es obligatoria (salvo la plantilla de estado que es opcional). Cuando el estudiante entrega el proyecto, el docente asignado chequea (entre otras cosas) que las plantillas sean consistentes con el código. De esta forma se mitiga que el estudiante diseñe una solución y luego codifique otra. Sin embargo, no se controla que el diseño sea completo y verificable.

5 Resultados y Análisis

Con el objetivo de estudiar el efecto de la representación del diseño con plantillas en la calidad analizamos la calidad desde los puntos de vista externo e interno.

5.1 Calidad Externa

Para analizar la calidad externa definimos las siguientes hipótesis nula y alternativa:

H0: El uso de plantillas de diseño no modifica la densidad de defectos en la fase de pruebas del proceso base.

H1: El uso de plantillas de diseño modifica la densidad de defectos en la fase de pruebas del proceso base.

Proponemos analizar la calidad externa de dos formas: intra grupo y entre grupos. Entre grupos se refiere a conocer si existe una diferencia significativa en la calidad del software entre el grupo *conPRD* y el grupo *sinPRD*. Es decir, conocer si hay una diferencia en la calidad entre el grupo que utiliza las plantillas y el grupo que no. Para este análisis, plantemos las siguientes hipótesis de investigación:

H0: Mediana(Densidad def. PU i) = Mediana (Densidad def. PU j)
H1: Mediana(Densidad def. PU i) > Mediana (Densidad def. PU j)
siendo i, j los estudiantes de los grupos conPRD y sinPRD respectivamente.

Cada muestra corresponde a la densidad de defectos en PU de cada estudiante considerando los programas 5 a 8. Se calcula de la siguiente manera:

$$\frac{1000 * \sum_{n=5}^{8} \#defectosPU_n}{\sum_{n=5}^{8} \#LOC_n} \tag{1}$$

Las muestras son independientes porque se corresponden a estudiantes diferentes, por lo que aplicamos la prueba de Mann-Whitney. Los resultados indican un valor de W = 354, p-value = 0.1656, con lo cual no podemos rechazar la hipótesis nula (significancia = 0.05). Por lo tanto, no podemos afirmar que los estudiantes que utilizaron las plantillas logran desarrollar productos con menos defectos que los estudiantes que no utilizaron los mismos.

El resultado anterior indica que no hay una diferencia estadística entre los estudiantes que usaron las plantillas de los que no. Sin embargo, también queremos conocer si los estudiantes que utilizaron plantillas mejoraron la calidad del software desarrollado. Para conocer esto analizamos la densidad de defectos en PU en los proyectos 1 al 4 y en los proyectos 5 al 8 para el grupo *conPRD*. Estudiar el comportamiento del mismo grupo nos permite conocer si hay un cambio en la calidad del software a partir del proyecto 5 (que es cuando se introducen las plantillas).

Debido a la complejidad del proyecto 2 respecto al resto de los proyectos decidimos no incluir los datos de este proyecto en el análisis. La naturaleza del proyecto 2 y los análisis reportados en nuestro estudio anterior [21] muestran que es un proyecto con una dificultad superior al resto y por lo tanto no comparable. Planteamos las siguientes hipótesis de investigación:

H0: Mediana(Densidad def. PU i) = Mediana (Densidad def. PU j)
H1: Mediana(Densidad def. PU i) > Mediana (Densidad def. PU j)
siendo i los estudiantes del grupo *conPRD* durante los proyectos 1, 3 y 4; y j los mismos estudiantes del grupo *conPRD* durante los proyectos 5 al 8.

En este caso las muestras están apareadas, por lo que aplicamos la prueba de Wilcoxon (*signed rank test*) para muestras apareadas. Los resultados indican un valor de V = 138 y p-value = 0.1438. Dado que el p-value es mayor a 0.05 (valor de la significancia) no es posible rechazar la hipótesis nula. Esto indica que no podemos afirmar que los estudiantes mejoran la calidad del software que producen al utilizar plantillas de diseño.

5.2 Calidad Interna

Para evaluar la calidad interna realizamos un análisis de los *code smells* en los que incurren los estudiantes al desarrollar los proyectos del curso. La intención de este análisis es conocer si el uso de plantillas de diseño evita que los estudiantes cometan ciertos *code smells*.

Los tipos de *code smell* dependen del lenguaje de programación. Dado que los estudiantes pueden elegir el lenguaje en el cual desarrollar los proyectos, hay que realizar este análisis teniendo en cuenta los distintos lenguajes utilizados. Con el objetivo de realizar un análisis inicial, y que aportara valor a nuestro estudio, se seleccionaron los estudiantes que desarrollaron los proyectos con los lenguajes Java, C#, C, C++ y Ruby; descartándose los desarrollados con PHP y Python. Agregar Python y PHP reducía mucho la cantidad de *code smells* comunes con los otros lenguajes, por ende, se descartaron ambos lenguajes para este análisis inicial. Esto dejó para el análisis un total de 45 estudiantes, 19 del año 2015, 14 del 2016 y 12 del 2017.

Para detectar los *code smell* se utilizó la herramienta SonarQube[1], ya que es una herramienta de software libre para una gran variedad de lenguajes de programación, que presenta actualizaciones constantes por la comunidad y una documentación muy amplia, entre otros.

Seleccionamos 16 *code smells* para el análisis. Estos cumplen que son comunes para los lenguajes de programación que seleccionamos y que son detectables por SonarQube. Los *code smells* son: 1) Las declaraciones "if ... else if" deben terminar con la cláusula "else", 2) Las declaraciones "switch"/"case" no deben estar anidadas, 3) Las declaraciones "switch"/"case" no deben tener demasiadas cláusulas "case"/"when", 4) La complejidad cognitiva de las funciones o métodos no debe ser demasiado alta, 5) Las declaraciones colapsables "if" deben fusionarse, 6) Las declaraciones de flujo de control "if", "for", "while", "switch" y "try" no deben anidarse demasiado, 7) Las expresiones no deben ser demasiado complejas, 8) Los archivos no deben tener demasiadas líneas de código, 9) Las funciones o métodos no deben tener demasiadas líneas de código, 10) Las funciones o métodos no deben tener demasiados parámetros, 11) Las líneas de código no deben ser demasiado largas, 12) Las funciones o métodos no deben ser vacíos, 13) Las declaraciones deben estar en líneas separadas, 14) Dos ramas en una estructura condicional no deben tener exactamente la misma implementación, 15) Los parámetros de una función o método no utilizados deben

[1] http://www.sonarqube.org

eliminarse, 16) Las variables locales no utilizadas deben eliminarse. Por motivos de espacio del artículo no se brinda una descripción más detallada de cada uno.

El Cuadro 2 presenta el porcentaje de estudiantes que al menos incurrió en un *code smell*, segmentado por proyecto (del 1 al 8) y por grupo (*sinPRD* y *conPRD*). Los *code smell* 3, 8 y 12 no están presentes en ninguno de los proyectos analizados.

Cuadro 2. Porcentaje de estudiantes que incurren en al menos un code smell por tipo de code smell y grupo.

Code smell	**Grupo**	Proyecto							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	*sinPRD*	4%	29%	0%	4%	13%	13%	4%	13%
	conPRD	19%	19%	10%	0%	5%	5%	5%	5%
2	*sinPRD*	0%	0%	0%	0%	0%	0%	0%	0%
	conPRD	0%	0%	0%	0%	0%	0%	0%	5%
4	*sinPRD*	8%	58%	0%	13%	30%	46%	29%	50%
	conPRD	24%	43%	5%	10%	10%	43%	24%	95%
5	*sinPRD*	4%	21%	0%	0%	0%	0%	0%	0%
	conPRD	0%	24%	10%	0%	0%	5%	0%	5%
6	*sinPRD*	13%	63%	8%	29%	30%	38%	13%	42%
	conPRD	38%	67%	29%	29%	33%	52%	57%	62%
7	*sinPRD*	0%	25%	0%	0%	0%	4%	8%	0%
	conPRD	0%	19%	0%	0%	0%	5%	0%	5%
9	*sinPRD*	0%	4%	8%	17%	10%	21%	21%	67%
	conPRD	0%	10%	19%	14%	10%	29%	38%	71%
10	*sinPRD*	0%	0%	0%	0%	0%	0%	8%	54%
	conPRD	0%	0%	5%	0%	0%	0%	19%	38%
11	*sinPRD*	4%	46%	42%	8%	40%	4%	46%	75%
	conPRD	0%	29%	29%	0%	14%	5%	24%	62%
13	*sinPRD*	0%	0%	0%	0%	10%	0%	0%	4%
	conPRD	5%	0%	5%	0%	0%	0%	5%	19%
14	*sinPRD*	0%	8%	0%	0%	10%	0%	0%	0%
	conPRD	0%	0%	0%	0%	0%	0%	0%	0%
15	*sinPRD*	0%	0%	8%	4%	20%	0%	13%	17%
	conPRD	0%	0%	0%	0%	5%	0%	0%	0%
16	*sinPRD*	8%	13%	8%	8%	40%	8%	17%	29%
	conPRD	5%	5%	10%	10%	0%	0%	10%	10%

Al analizar el cuadro entre los grupos *sinPRD* y *conPRD* a partir del programa 5 (a partir del uso de plantillas) se desprende una gran variabilidad, tanto si se mira por proyecto como si se mira por *code smell*.

Para los code smell 4, 7, 10 y 13 se observa que para ciertos proyectos un grupo está mejor y para ciertos otros proyectos está mejor el otro grupo. Para los *code smell* 1, 2, 5, 6, 9 y 14 sucede que la diferencia es muy chica entre los

grupos. En definitiva, para ninguno de estos *code smells* se observan cambios al usar las plantillas.

Para el caso del *code smell* 11 se observa un porcentaje muy menor en los proyectos 5 y 7 y menor en el proyecto 8 por parte del grupo que utiliza las plantillas. En el proyecto 6 ambos grupos se comportan casi igual. Desde el punto de vista de las plantillas quizás sea la plantilla de seudocódigo la que esté ayudando a los estudiantes a disminuir la introducción de este *code smell*.

Los *code smell* 15 y 16 tienen un comportamiento similar. Para ambos casos el grupo *conPRD* casi no incurre en ellos mientras que el grupo *sinPRD* incurre y a veces en un alto porcentaje. El 15 refiere a parámetros no utilizados en los métodos y el 16 a variables locales no utilizadas. Claramente estos tipos de *code smells* pueden ser evitados con buenos diseños de software. Desde el punto de vista del uso de las plantillas, quizás el desarrollo de seudocódigo (plantilla Lógica) y la plantilla Funcional estén evitando que los estudiantes del grupo *conPRD* incurran en estos *code smells*. De todas formas, es necesario analizar manualmente las plantillas entregadas por los estudiantes y tener entrevistas con ellos para conocer mejor si esto puede estar sucediendo por los motivos descritos. Esto aún no se ha realizado.

Sin embargo, al analizar el cuadro, pero solamente considerando los datos del grupo *conPRD* a lo largo de los 8 proyectos, no vemos que el uso de plantillas mejore la calidad interna.

Vale la pena notar que este grupo normalmente no incurrió (o lo hizo en muy bajo porcentaje) en los *code smells* 15 y 16. Observando los proyectos 1 a 4 y 5 a 8 por separado, no vemos una diferencia entre ellos. Es decir, el comportamiento de este grupo antes de usar plantillas y durante su uso no cambia para estos *code smells*. Entonces, la diferencia presentada en el análisis anterior entre los grupos *conPRD* y *sinPRD* no responde al uso de las plantillas.

Algo similar sucede con el *code smell* 11. Los resultados no presentan una disminución de este *code smell* al utilizarse las plantillas.

Se puede observar que en el proyecto 8 aumenta significativamente el porcentaje de ocurrencia de los *code smells* 4, 9 y 10 para ambos grupos. Este aumento hace pensar que el proyecto 8 resulta mas complejo para los estudiantes. Estos tres *code smells* indican que el código desarrollado es demasiado complejo y largo para su comprensión. Es decir, el uso de las plantillas no ayudó a los estudiantes a elaborar un diseño menos complejo y entendible.

Juntando ambos análisis se desprende que el uso de las plantillas no mejora la calidad interna. En particular, el uso de las plantillas no tiene un efecto en los *code smells* en los que incurren los estudiantes al desarrollar software.

6 Discusión

En el contexto de nuestro experimento encontramos que la representación del diseño utilizando plantillas no ayudó a desarrollar productos de software de mayor calidad. Los resultados arrojan que el uso de plantillas no mejoró en la cantidad de defectos que contiene el código desarrollado (medido como densidad

de defectos en PU) ni mejoró la calidad interna (medido como *code smells* en el código). Estos resultados se relacionan con los reportados por Gravino [12], donde el uso de diagramas UML no logró mejoras en la comprensión de código fuente respecto al no uso de los mismos.

El uso de plantillas para representar el diseño no tuvo un efecto positivo en la calidad. Los estudiantes que las usaron no mejoraron la misma. Esto puede deberse a varios factores que deberemos analizar en el futuro. Podría ser, entre otros motivos, que no están habituados a estas plantillas y por ende no se obtuvo el beneficio esperado, podría ser que simplemente completaron las plantillas pero no se preocuparon en ese momento por pensar y desarrollar un diseño de calidad o podría ser que los estudiantes no saben qué y cómo diseñar.

Si bien falta más análisis, nosotros coincidimos con varios autores en que los estudiantes de pregrado tienen dificultades para diseñar y no parecen entender qué tipo de información incluir para diseñar software [8, 9, 20].

7 Amenazas a la validez

La mayoría de los estudios empíricos se ven amenazados por la forma en que la investigación se lleva a cabo. En esta sección se describen las amenazas a la validez que hemos detectado.

Investigar con estudiantes implica varias amenazas. Por un lado, el hecho que el contexto del experimento sea un curso implica que el estudiante se desenvuelva de forma distinta al contexto profesional. Intentamos minimizar esta amenaza con un curso sin calificación, es decir el estudiante aprueba o reprueba. Además hacemos hincapié en la importancia de seguir y registrar el proceso tal cual fue y dejando claro que no se evaluará por los resultados, defectos encontrados o esfuerzos dedicados. Además, para el análisis realizamos una agregación de los datos de los tres cursos; sabiendo que distintos cursos pueden influir en los datos recolectados por ser un modelo jerárquico. Intentamos disminuir esta amenaza mediante el uso de un proceso definido y disciplinado que siguen los estudiantes y manteniendo el mismo material y los mismos docentes a lo largo de los 3 cursos.

Por otro lado, los estudiantes realizan los proyectos desde sus casas lo que provoca un control limitado por parte de los investigadores. Intentamos reducir esta amenaza asignando un tutor a cada estudiante de forma de supervisar y atender las consultas durante la realización de proyectos. Además, el tutor revisa cada proyecto una vez finalizado y envía feedback correcciones a sus estudiantes.

Por último, la cantidad de estudiantes en el estudio constituye una amenaza a la conclusión estadística. Participaron 61 estudiantes durante las 3 ejecuciones. Esto provoca la realización del análisis estadístico utilizando test no paramétricos cuya potencia estadística es menor a los test paramétricos.

8 Conclusiones

Este trabajo es un paso más hacia la comprensión de la práctica del diseño de software. Los resultados de nuestro experimento muestran que los estudiantes

de pregrado no mejoran la calidad del software cuando utilizan plantillas para representar el diseño. Analizamos la calidad del software desde los puntos de vista interno y externo. Por un lado, comprobamos estadísticamente que el uso de plantillas para representar el diseño no mejora la calidad externa del software medida como la densidad de defectos en las pruebas unitarias. Desde el punto de la calidad interna, el uso de las plantillas no tiene un efecto positivo significativo en los *code smells* en los que incurren los estudiantes al desarrollar software.

Estos resultados, sumados al reportado previamente (los estudiantes dedican cuatro veces menos esfuerzo al diseño que a la codificación) [21], nos generan nuevas interrogantes sobre la práctica del diseño de software: ¿Qué diseñan habitualmente los estudiantes?, ¿Qué tipo de información incluyen al diseñar?, ¿Es posible que realicen sus diseños mentalmente sin representarlos?, ¿Conocen el efecto de un buen diseño en la calidad del software? Como trabajo a futuro proponemos analizar los diseños realizados en las plantillas para conocer qué diseñan, y poder detectar problemas. Por otro lado, nos interesa ejecutar un nuevo experimento que nos permita analizar qué diseñan los estudiantes sin la exigencia de entregar una representación definida por los investigadores. Estudiar el comportamiento habitual del estudiante al diseñar software nos permitirá detectar posibles problemas en las prácticas de diseño y proponer mejoras en la enseñanza que permitan la construcción de software de calidad.

Referencias

1. Arisholm, E., Briand, L.C., Hove, S.E., Labiche, Y.: The impact of uml documentation on software maintenance: an experimental evaluation. IEEE Transactions on Software Engineering **32**(6) (2006)
2. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J.: Using static analysis to find bugs. IEEE software **25**(5) (2008)
3. Bourque, P., Fairley, R.E.: Guide to the Software Engineering Body of Knowledge - SWEBOK v3.0. IEEE CS, 2014 version edn. (2014)
4. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998)
5. Campbell, G.A., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co, 2013 version edn. (2013)
6. Carrington, D., K Kim, S.: Teaching software design with open source software. In: 33rd Annual Frontiers in Education, 2003. FIE 2003. (2003)
7. Chen, T.Y., Cooper, S., McCartney, R., Schwartzman, L.: The (relative) importance of software design criteria. In: ITiCSE (2005)
8. Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Zander, C.: Can graduating students design software systems? In: SIGCSE'06. ACM (2006)
9. Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Zander, C.: Categorizing student software designs: Methods, results, and implications. Computer science education **16**(3) (2006)
10. Flores, P., Medinilla, N.: Conceptions of the students around object-oriented design: A case study. In: XII Jornadas Iberoamericanas de Ingenieria de Software e Ingeniería del Conocimiento (2017)

11. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
12. Gravino, C., Scanniello, G., Tortora, G.: Source-code comprehension tasks supported by uml design models: Results from a controlled experiment and a differentiated replication. Journal of Visual Languages & Computing **28** (2015)
13. Hu, C.: The nature of software design and its teaching: an exposition. ACM Inroads **4**(2) (2013)
14. Humphrey, W.S.: A discipline for software engineering. Addison-Wesley Longman Publishing Co., Inc. (1995)
15. ISO/IEC: ISO/IEC 9126. Software engineering – Product quality. ISO/IEC (2008)
16. Karasneh, B., Jolak, R., Chaudron, M.R.V.: Using examples for teaching software design: An experiment using a repository of uml class diagrams. In: 2015 Asia-Pacific Software Engineering Conference (APSEC) (2015)
17. Kramer, J.: Is abstraction the key to computing? Commun. ACM **50** (2007)
18. Leung, F., Bolloju, N.: Analyzing the quality of domain models developed by novice systems analysts. In: 38th Hawaii International Conference on System Sciences (2005)
19. Linder, S.P., Abbott, D., Fromberger, M.J.: An instructional scaffolding approach to teaching software design. Journal of Computing Sciences in Colleges **21** (2006)
20. Loftus, C., Thomas, L., Zander, C.: Can graduating students design: revisited. In: Proceedings of the 42nd ACM technical symposium on Computer science education. ACM (2011)
21. Moreno Silvana, V.D.: ¿los estudiantes de pregrado son capaces de diseñar software? estudio de la relación entre el tiempo de codificación y el tiempo de diseño en el desarrollo de software. In: Conferencia Iberoamericana de Ingeniería de Software 2018 (2018)
22. Petre, M.: Uml in practice. International Conference on Software Engineeringn **35** (2013)
23. Pierce, K., Deneen, L., Shute, G.: Teaching software design in the freshman year. In: Software Engineering Education. Springer Berlin Heidelberg (1991)
24. Richard N. Taylor: Proc. 33rd International Conference on Software Engineering (ICSE 2011). ACM (2011)
25. Schneidewind, N.F., Keller, T.W.: Applying reliability models to the space shuttle. IEEE Software **9**(4) (1992)
26. Siau, K., Tan, X.: Improving the quality of conceptual modeling using cognitive mapping techniques. Data & Knowledge Engineering **55**(3) (2005), quality in conceptual modeling
27. Soh, Z., Sharafi, Z., Van den Plas, B., Cepeda Porras, G., Guéhéneuc, Y.G., Antoniol, G.: Professional status and expertise for uml class diagram comprehension: An empirical study. In: IEEE International Conference on Program Comprehension (2012)
28. Sommerville, I.: Software Engineering. Pearson (2016)
29. Stevenson, J., Wood, M.: Recognising object-oriented software design quality: a practitioner-based questionnaire survey. Software Quality Journal **26** (2018)
30. Tenenberg, J.: Students designing software: a multi-national, multi-institutional study. Informatics in Education **4** (2005)
31. Torchiano, M., Scanniello, G., Ricca, F., Reggio, G., Leotta, M.: Do uml object diagrams affect design comprehensibility? results from a family of four controlled experiments. Journal of Visual Languages & Computing **41** (2017)