
Tesis de Maestría

# $PSP_{VDC}$: Propuesta de adaptación del Proceso de Software Personal para incorporar Diseño por Contrato Verificado

Silvana Moreno

Directores de Tesis: Álvaro Tasistro y Diego Vallespir

Director de Estudios: María Urquhart

Programa de Maestría en Informática

Junio de 2013

**UNIVERSIDAD DE LA REPÚBLICA**
**FACULTAD DE INGENIERÍA**

El tribunal docente integrado por los abajo firmantes aprueba la Tesis de Investigación:

**$PSP_{VDC}$: Propuesta de adaptación del Proceso de Software Personal para incorporar Diseño por Contrato Verificado**

**Autor:** Ing. Silvana Moreno
**Director de Tesis:** Dr. Álvaro Tasistro y Dr. Diego Vallespir
**Director Académico:** Dr. María Urquhart
**Carrera:** Maestría en Informática - PEDECIBA
**Calificación:** _____

**TRIBUNAL**

Dra. Juliana Herbert (Revisora) _____

Dra. Cristina Cornes _____

M.Sc. Omar Viera _____

Montevideo, _____

# Contenido

# Capítulo 1.

## 1.1 Introducción

El software se ha vuelto muy común e importante en el mundo moderno actual. El tamaño, la complejidad de las aplicaciones, las tasas apresuradas de entregas de proyectos, las características de los equipos de desarrollo, entre otros, hacen que los productos de software contengan defectos. Estos defectos pueden ocasionar fallas en el software mientras este se está ejecutando [Sommerville 10].

La búsqueda de desarrollar software cero defectos ha dado lugar a un gran número de procesos y metodologías de desarrollo. El *Personal Software Process* (PSP) es uno de estos procesos. El PSP aplica disciplina de proceso y gestión cuantitativa al trabajo individual del ingeniero de software. Promueve la utilización de prácticas específicas durante todas las etapas del desarrollo con el objetivo de mejorar la calidad del producto y aumentar la productividad del individuo [Humphrey 05, Humphrey 06].

Por otro lado, los métodos formales son un conjunto de técnicas para especificar, desarrollar y verificar sistemas software mediante el uso del lenguaje matemático. Consisten en demostrar matemáticamente que los programas producidos cumplen sus especificaciones. El diseño por contrato (DbC) es una técnica para el diseño de los componentes de un sistema de software, mediante el establecimiento de las condiciones (pre y post condiciones) y el comportamiento en un lenguaje formal. Cuando las técnicas y herramientas incorporadas permiten demostrar que se cumplen los requisitos establecidos, estamos en presencia de un método formal generalmente llamado V*erified Design by Contract* (VDbC).

En esta tesis se construye una adaptación al PSP que incorpora VDbC con el objetivo de mejorar, en comparación con el uso del PSP sin VDbC, la calidad de los productos y mantener o mejorar la productividad del individuo. Denominamos $PSP_{VDC}$ a nuestra propuesta. Esperamos de esta forma aportar en la búsqueda de procesos que ayuden a desarrollar productos software muy cercanos a cero defectos.

## 1.2 Contexto

El PSP se divide en fases, como se observa en la figura 1.1. Comienza cuando el ingeniero recibe los requerimientos para un programa o componente de software y termina en el momento en que se libera el mismo. Las fases son: *Planning*, *Design*, *Design review*, *Code*, *Code review*, *Compile*, *Unit Test* y *Post-mortem*.[1]

[1] Se decide mantener algunas palabras en inglés a lo largo de la tesis debido a que son términos comúnmente utilizados y entendidos en esta disciplina.

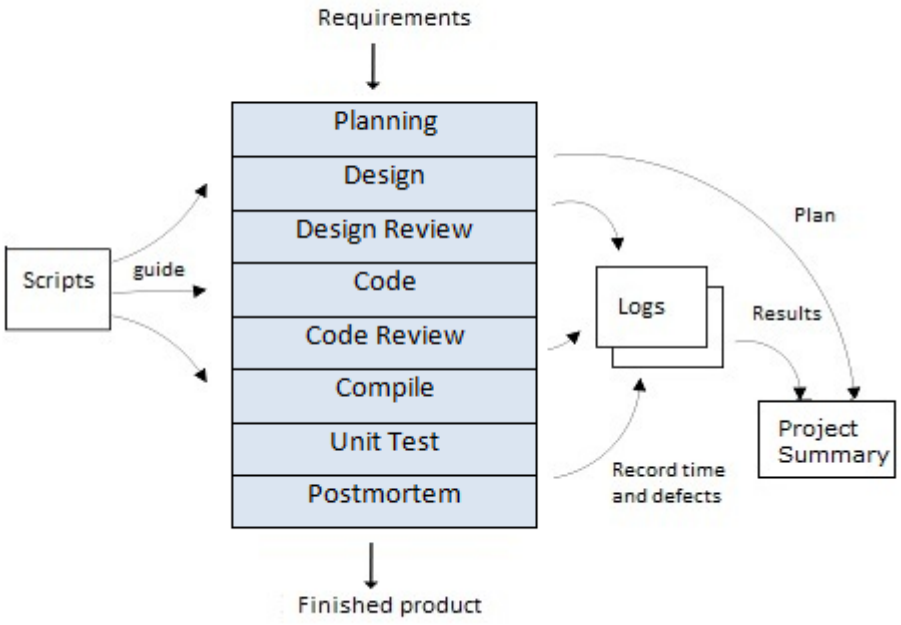



*Figura 1.1 – Fases del PSP*

El PSP se enseña a través de un curso. Durante el curso, los ingenieros construyen programas al mismo tiempo que aprenden a planificar, desarrollar y seguir las prácticas propuestas por el PSP. Se han recogido y analizado estadísticamente los datos sobre la productividad de los ingenieros durante el curso en varias ocasiones. Los resultados revelan que el PSP reduce la cantidad de defectos por línea de código (densidad de defectos) que llegan hasta la fase de *Unit Test* [Hayes 97] y, por lo tanto, la aplicación del PSP mejora significativamente la calidad del producto. La sección 2.2 del Capítulo 2 presenta con mayor profundidad el PSP.

Los Métodos Formales, a su vez, participan de la misma estrategia metodológica como la del PSP, es decir, haciendo énfasis en los procedimientos de desarrollo (opuesto a depender del testing y la depuración). Establecen un requisito en este sentido, que consiste en demostrar matemáticamente que los programas producidos cumplen sus especificaciones. Diseño por contrato (DbC) es una técnica ideada (y patentada) por Bertrand Meyer para el diseño de los componentes de un sistema de software, mediante el establecimiento de las condiciones (pre y post condiciones) en un lenguaje formal [Meyer 92].

Cuando las técnicas y herramientas incorporadas permiten realizar la demostración de la concordancia de cada pieza de software con su especificación estamos en presencia de un método formal generalmente llamado Diseño por Contrato Verificado (VDbC). La sección 2.3 del Capítulo 2 presenta con mayor profundidad VDbC y el Anexo B presenta un ejemplo del uso de VDbC utilizando el lenguaje JML.

# 1.3 Objetivos

Nuestro objetivo general es proponer un proceso de desarrollo de software que mejore la calidad de los productos. Específicamente en esta tesis, se pretende construir una adaptación al PSP, denominada $PSP_{VDC}$ que incorpora el enfoque de VDbC. Se propone también como objetivo la planificación y el diseño de experimentos controlados que permitan comprobar si $PSP_{VDC}$ mejora la calidad de los productos y mantiene o mejora la productividad individual con respecto a PSP.

A continuación se presentan los objetivos con las preguntas de investigación asociadas:

Objetivo 1 - Conocer adaptaciones ya realizadas al PSP que incorporen métodos formales.

Pregunta de investigación 1: ¿Existen adaptaciones al PSP que incorporan métodos formales?

Objetivo 2 – Construir una adaptación al PSP que incorpore VDbC.

Preguntas de investigación 2: ¿Es adaptable el PSP para incorporar VDbC? ¿Cuál sería una adaptación razonable?

Objetivo 3 – Planificar y diseñar un estudio empírico para comparar la calidad y productividad de $PSP_{VDC}$ con respecto a PSP.

Preguntas de investigación 3: ¿Cómo sería una planificación adecuada para comparar los procesos? ¿Cuál sería un diseño adecuado?

## 1.4 Adaptaciones al PSP

Con el fin de responder a la primera pregunta de investigación realizamos una revisión sistemática de la literatura que busca conocer las adaptaciones al PSP que ya fueron documentadas. En particular nos interesa conocer las adaptaciones que incorporan (o que fueron realizadas para incorporar) métodos formales.

Una revisión sistemática de la literatura es un medio de identificación, evaluación e interpretación de toda la información que se disponga relativa a una pregunta de investigación, área temática o fenómeno de interés [Kitchenham 07].

La revisión sistemática realizada como parte de esta tesis describe el motivo de la revisión, las preguntas de investigación, la estrategia de búsqueda, los criterios de inclusión/exclusión, el control de calidad, la extracción y síntesis de los datos y los resultados respetando la forma de presentar revisiones propuesta por Kitchenham [Kitchenham 07].

Se encontraron tres trabajos que adaptan el PSP para incorporar métodos formales y dos que adaptan al PSP pero sin incorporar métodos formales. De los trabajos que incorporan métodos formales, dos de ellos utilizan el lenguaje formal VDM y el otro el lenguaje formal B. De los trabajos que no incorporan métodos formales, uno de ellos adapta el PSP para incorporar el método de programación de a pares (*pair programming*) y el otro integra al PSP técnicas de desarrollo de software dirigido por modelos.

Todas las propuestas encontradas modifican *scripts*, *templates* y formularios para dar soporte a los nuevos procesos. Algunas optan por mantener las mismas fases de desarrollo del PSP agregando actividades, mientras que otras incorporan nuevas fases al proceso.

Los resultados de la revisión sistemática se presentan en el Capítulo 3.

## 1.5 El $PSP_{VDC}$

Para responder a la segunda pregunta investigamos cómo integrar *Verified Design by Contract* (VDbC) al PSP. Como resultado de la investigación desarrollamos una adaptación al PSP, denominada $PSP_{VDC.}$ El objetivo de esta adaptación es reducir el número de defectos por línea de código que llega a la fase de *Unit Test* (en comparación con el uso del PSP) preservando la productividad del ingeniero.

$PSP_{VDC}$ incorpora nuevas fases y modifica otras, así como agrega nuevos *scripts* y *check lists* al PSP. La figura 1.2 ilustra las fases del $PSP_{VDC}$ y las compara con el PSP. Las flechas rojas mapean las fases nuevas del $PSP_{VDC}$ que se corresponden con actividades específicas dentro de otras fases del PSP. Específicamente en $PSP_{VDC}$ se agregan las fases:

- *Test Case Construct:* se lleva a cabo luego de la revisión de diseño. Durante esta fase se construye el conjunto de casos de prueba a utilizar para testear el programa
- *Formal Specification:* durante esta fase se especifican formalmente los contratos (*Design by Contract*). Las pre- y post- condiciones de los métodos y el invariante de la clase son especificados en un lenguaje formal.
- *Formal Specification Review:* la revisión de la especificación formal tiene el propósito de detectar los defectos inyectados durante la especificación formal.
- *Formal Specification Compile:* la compilación consiste en el chequeo automático de la sintaxis de la especificación formal.
- *Pseudo code:* durante esta fase se escribe el pseudo código de los métodos de las clases.
- *Pseudo code Review*: en esta fase se revisa el pseudo código anteriormente producido
- *Proof:* La idea general de esta fase es complementar el código con una prueba formal. El objetivo es que la prueba provea evidencia de la correctitud del código con respecto a las especificaciones formales (*Verified Design by Contract*). Esta prueba debe ser llevada a cabo con la ayuda de una herramienta.

El $PSP_{VDC}$ se presenta de forma completa en el Capítulo 2.

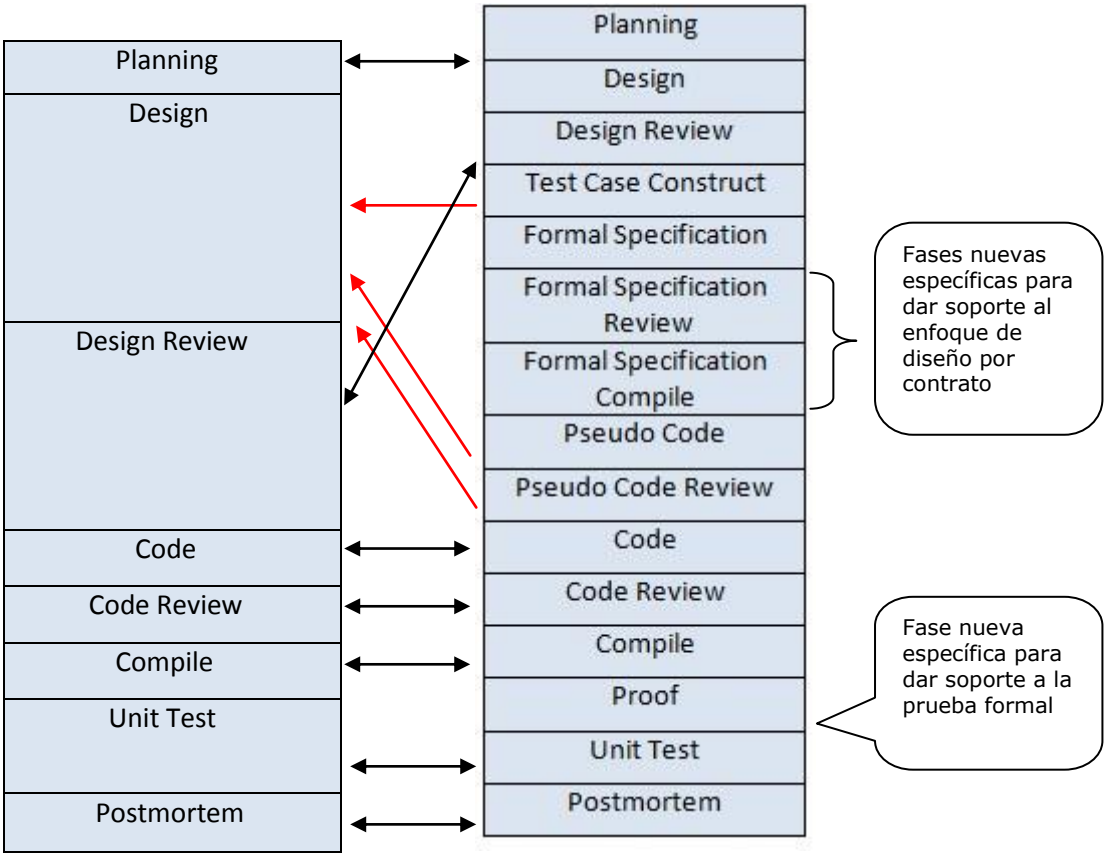



*Figura 1.2 – Las fases de PSP y $PSP_{VDC}$*

# 1.6 Experimentos Controlados

Para lograr el objetivo de investigación 3 planteamos una propuesta para comparar el $PSP_{VDC}$ con el PSP mediante la realización de experimentos controlados. Los experimentos controlados son una técnica de investigación en la cual se quiere tener control del estudio y del entorno en el que éste se lleva a cabo[2].

Debido a que el $PSP_{VDC}$ puede tener dificultades en su aprendizaje, definimos 2 niveles que permiten introducir los conceptos del $PSP_{VDC}$ gradualmente. El nivel 1 denominado $PSP_{VDC}$ *light* y el nivel 2 denominado $PSP_{VDC}$ *full*. El $PSP_{VDC}$ *light* no utiliza la prueba de programas mientras que el $PSP_{VDC}$ *full* sí. Estos niveles permiten armar un curso que introduzca gradualmente las prácticas del $PSP_{VDC}$, explicando entre otras cosas en el nivel 1 lo correspondiente a especificaciones formales y en el nivel 2 los conceptos de las pruebas.

Proponemos la planificación y el diseño de dos experimentos controlados, denominados experimento *light* y experimento *full* respectivamente. El objetivo de los experimentos es conocer y comparar la calidad de los productos y productividad que se obtiene al aplicar ambos procesos. Definimos calidad como la densidad de defectos encontrados en la fase de *Unit Test* (cantidad de defectos/LOCS), y la productividad como la cantidad de líneas de código producidas por hora (LOCS/horas).

Los experimentos se llevarán a cabo en el contexto de la Universidad de la República como parte de una materia de Facultad.

En el experimento *light* se comparan el $PSP_{VDC}$ *light* y el PSP y en el experimento *full* se comparan el $PSP_{VDC}$ *full* y el PSP. Durante el experimento *light* un grupo de estudiantes aplica PSP o $PSP_{VDC}$ *light* a un conjunto de ejercicios. La mitad de los estudiantes aplicará el PSP y la otra mitad el $PSP_{VDC}$ *light* sobre los mismos ejercicios. Los estudiantes deberán ser previamente entrenados. Los entrenamientos consisten en capacitar a los estudiantes en el PSP, el $PSP_{VDC}$ *light* y métodos formales antes de la ejecución del experimento.

De forma similar, durante el experimento *full* un grupo de estudiantes aplica PSP o $PSP_{VDC}$ *full* a un conjunto de ejercicios. La mitad de los estudiantes aplica PSP y la otra mitad $PSP_{VDC}$ *full* sobre los mismos ejercicios. El experimento $PSP_{VDC}$ *full* se pretende realizar un año después que el experimento $PSP_{VDC}$ *light* y con estudiantes diferentes. Los estudiantes deberán ser entrenados en PSP, $PSP_{VDC}$ *full*, métodos formales y técnicas de verificación de programas.

Se definen y planifican ambos experimentos. Se establecen los objetivos, los diseños experimentales, los test de hipótesis, se planifica el armado de los cursos y los entrenamientos a los sujetos. La ejecución y el análisis de datos quedan por fuera del trabajo de esta tesis. Sin embargo, se propone el plan de la ejecución que se desea llevar adelante y cómo se pretenden analizar los resultados.

# 1.7 Publicaciones

En el transcurso de esta tesis se publicaron artículos en revistas (arbitradas y no arbitradas), en conferencias y reportes técnicos.

La primera propuesta de la adaptación al PSP que incorpora diseño por contrato fue publicada en el TSP Symposium 2012.

- Silvana Moreno, Álvaro Tasistro, Diego Vallespir, William Nichols. $PSP_{DC}$: An Adaptation of the PSP to Incorporate Verified Design by Contract in Proceed-

[2] El Anexo A, sección "Experimentos Formales" presenta los conceptos básicos de los experimentos controlados en ingeniería de software.

ings TSP Symposium 2012: Delivering agility with discipline (Special Report Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2012-SR-015), pp.41—50, Saint Petersburg, Florida, EEUU, Setiembre 2012.

La propuesta final, que incorpora *Verified Design by Contract* al PSP se publica como reporte técnico del SEI. Este artículo se corresponde con el capítulo 2 de la tesis.

- Silvana Moreno, Álvaro Tasistro, Diego Vallespir, William Nichols. $PSP_{VDC}$: An Adaptation of the PSP to Incorporate Verified Design by Contract. Technical Report, CMU/SEI-2013-TR-005, ESC-TR-2013-005. Abril 2013.

La revisión sistemática de la literatura sobre adaptaciones al PSP se publica como Reporte Técnico del PEDECIBA.

- "Systematic Literature Review of PSP Adaptations", Reporte Técnico InCo/Pedeciba-2013 TR:XXX.

Se genera una nueva versión del Reporte Técnico "Conceptos de Ingeniería de Software Empírica" incorporando el método de investigación de Estudio de Casos y los trabajos de investigación realizados por el Grupo de Ingeniería de Software.

- "Conceptos de Ingeniería de Software Empírica", versión 2.0. Reporte Técnico InCo/Pedeciba-2013 TR: 13-01.

Se presentó un artículo a la revista Milveinticuatro (no arbitrada) que aún no se ha publicado.

- Silvana Moreno, Diego Vallespir, Álvaro Tasistro. "Pruebas Unitarias en Java: Diseño por Contratos", Milveinticuatro

## 1.8 Estructura del Documento

El documento de tesis contiene un total de 5 capítulos y 2 anexos. En el capítulo 1 se presenta la introducción.

El capítulo 2 presenta de forma completa el $PSP_{VDC}$. Este capítulo es el Reporte Técnico publicado en el Software Engineer Institute (SEI) y está escrito en inglés. El capítulo 3 presenta la revisión sistemática realizada para conocer las distintas adaptaciones existentes del PSP. Este capítulo es el Reporte Técnico PEDECIBA-InCo y también está en inglés.

En el capítulo 4 se proponen los experimentos controlados para comprar el $PSP_{VDC}$ con el PSP. El capítulo 5 contiene las conclusiones y el trabajo a futuro.

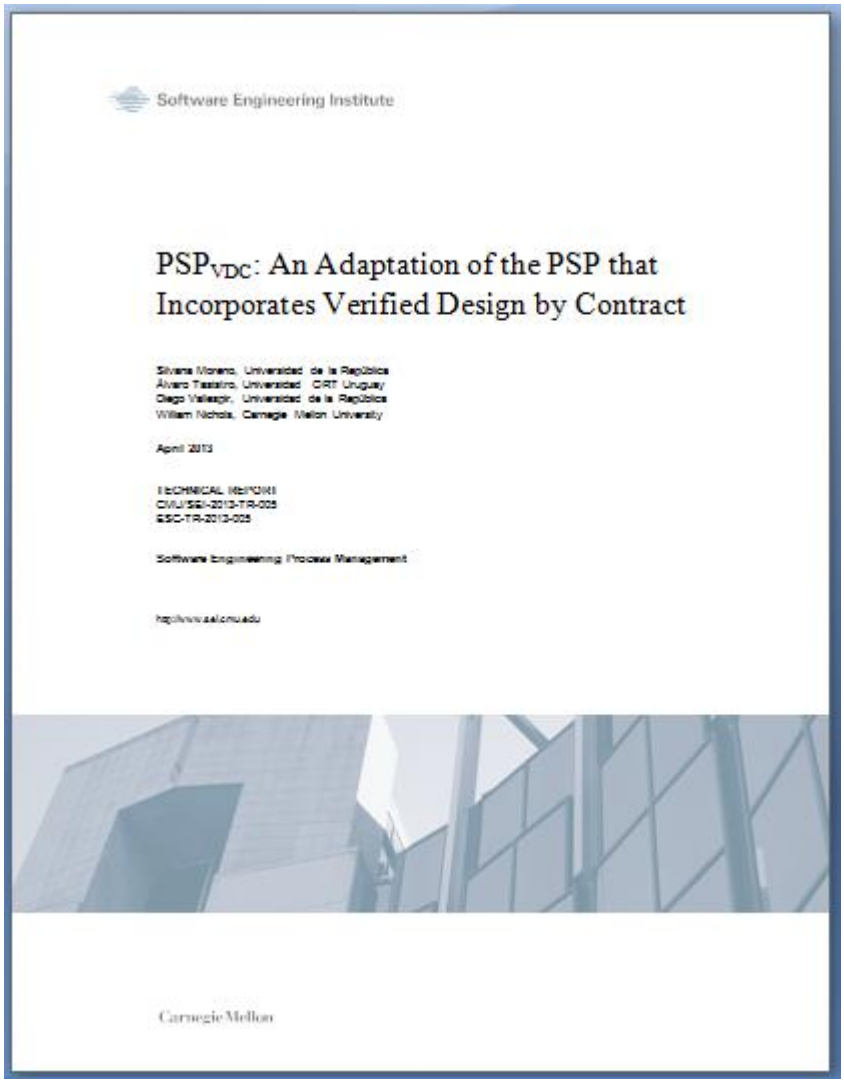
El anexo A presenta el RT de "Conceptos de Ingeniería de Software Empírica" y el anexo B presenta el artículo de "Pruebas Unitarias en Java: Diseño por Contratos".

Debido a que dos capítulos de esta tesis son Reportes Técnicos publicados en inglés la tesis contiene 2 capítulos en inglés y el resto en español.

# Capítulo 2.

## $PSP_{VDC}$: An Adaptation of the PSP to Incorporate Verified Design by Contract

Silvana Moreno, Álvaro Tasistro, Diego Vallespir, William Nichols



# 2.1 Introducción

Software increases in size and complexity each year and plays a larger role in many aspects of our lives. Because the development of software is a creative and intellectual activity performed by human beings, it is normal for the development team to make mistakes, both because of the complexity of the software and because of human nature itself. These mistakes often end up as defects in software products and can cause severe damage when the software is executed. Research on developing defect-free software has led to the development of many processes and methods that aim to detect defects before the product is delivered to the users.

The Personal Software Process (PSP) incorporates process discipline and quantitative management into the software engineer's individual development work. It promotes the exercise of careful procedures during all stages of development with the aim of increasing the individual's productivity and achieving high quality final products [Humphrey 2005, Humphrey 2006].

The PSP course progressively teaches engineers planning, development, and process assessment practices as they build actual programs. Performance data from students in this course has been collected and statistically analyzed, and the results show that PSP substantially reduces the amount of defects per lines of code that survive until the Unit Testing phase [Hayes 1997] [Rombach 2007], indicating that employment of PSP significantly improves product quality.

Still, removing defects at the Unit Testing phase costs five to seven times more than removing them in earlier phases of the PSP [Vallespir 2011, Vallespir 2012]. Because 38% of the injected defects are still present at Unit Testing, opportunities exist for improvement in the early detection of defects using TSP.

Formal methods use the same methodological strategy as the PSP: emphasizing care in development procedures as opposed to relying on testing and debugging. They also establish the radical requirement of proving mathematically that the programs produced satisfy their specifications. Design by Contract (DbC) is a technique devised and patented by Bertrand Meyer for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language [Meyer 1992]. The formal languages that are used for DbC are seamlessly integrated into the programming language to allow specified conditions to be evaluated at runtime, with violations of these conditions managed with exception handling. When appropriate techniques and tools are incorporated to prove that the components satisfy the established requirements, the method is called Verified Design by Contract (VDbC).

In this paper we propose a way to integrate VDbC into PSP to reduce the amount of defects present at the Unit Testing phase, while at the same time preserving or improving productivity. The resulting adaptation of the PSP, called $PSP_{VDC}$, incorporates new phases, modifies others, and adds new scripts and checklists to the infrastructure. Specifically, the phases of Formal Specification, Formal Specification Review, Formal Specification Compile, Test Case Construct, Pseudo Code, Pseudo Code Review, and Proof are added. At a later stage, controlled experiments will be conducted for obtaining results about the improvements achieved by our adaptation. We expect that such experiments will motivate further adjustments to the process so that it eventually becomes practical enough to be employed in industry.

We know of only three works in the literature that propose a combination of PSP and formal methods. Babar and Potter [Babar 2005] combine Abrial's B Method with PSP into B-PSP. They add the phases of Specification, Auto Prover, Animation, and Proof. A new set of defect types is added and logs are modified so as to incorporate data extracted from the B machine's structure. The goal of this work is to provide the individual B developers with a paradigm of measurement and evalua-

tion that promotes reflection on the practice of the B method, inculcating the habit of recognizing causes of defects injected to help prevent them in the future. In comparison to B, our chosen formal method is significantly lighter and so, we expect, easier to incorporate into industrial practice.

Suzumori, Kaiya, and Kaijiri proposed the combination of VDM and PSP [Suzumori 2003]. In their method, the Design phase is modified incorporating the formal specification in the VDM-SL language. New phases are also added: VDM-SL Review, Syntax Check, Type Check, and Validation. A prototype course requiring each student to carry out nine exercises applying VDM on the PSP was designed. After this work was concluded, the research was discontinued for reasons internal to the organization.[3]

Contemporaneously to our work, Kusakabe, Omori, and Araki proposed combining PSP and VDM with the goal of avoiding the injection of defects at the design phase [Kusakabe 2012]. They use automated tools (VDMTools) for syntax checking, type checking, interpretation, and generation of proof obligations. For evaluating the resulting process they had an engineer apply ordinary PSP to the course materials of PSP for Engineers I, then apply the combination of PSP and VDM to a few exercises in the course material of PSP for Engineers II. The experimental results show a successful reduction of the number of defects, without decreased productivity. However, they note that proficiency in the programming language and software development skills might affect the results.

The rest of this paper describes the PSP and $PSP_{VDC}$ methods and is structured as follows: Section 2 provides a general description of PSP, while Section 3 gives a general description of formal methods—VDbC in particular. Section 4 presents the adaptation of PSP to incorporate VDbC. Finally, Section 5 describes conclusions and further work.

## 2.2 Personal Software Process

The PSP was proposed in 1995 by Watts Humphrey at the Software Engineering Institute (SEI). It is aimed at increasing the quality of the products manufactured by individual professionals by improving their personal methods of software development. It takes into account diverse aspects of the software process, including planning, quality control, cost estimation, and productivity.

The PSP is divided into phases, as shown in Figure 1. A project begins with the requirements for a software module and ends when the software is released. The phases are: Planning, Design, Design Review, Code, Code Review, Compile, Unit Test, and Postmortem.

[3] As communicated by the authors via e-mail.

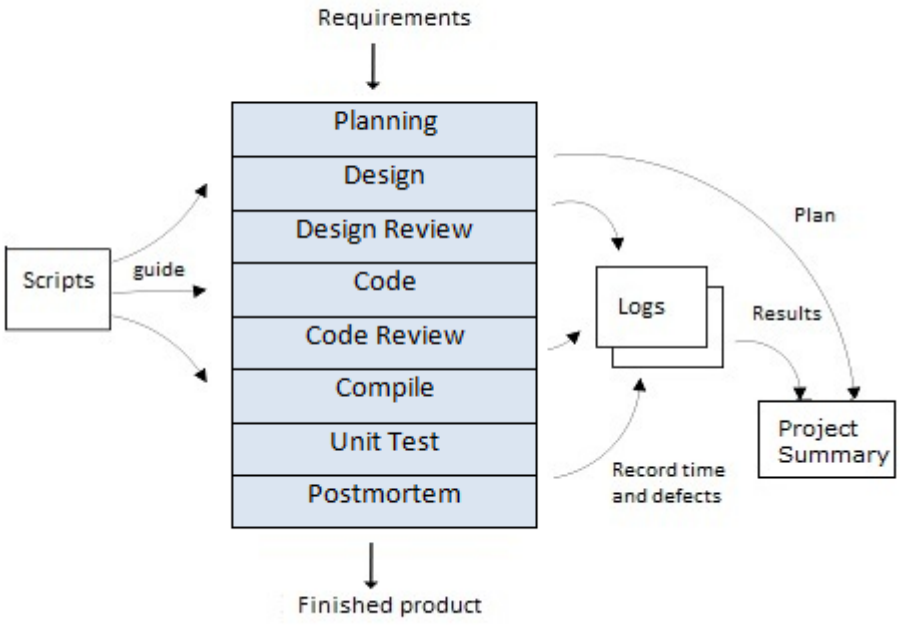



*Figure 1: Phases of the PSP*

In the PSP, all tasks and activities to be performed during software development are defined in a set of documents called scripts. Scripts dictate the course of the work and are to be followed in a disciplined manner. They also facilitate the collection of data about the software process, including time spent at each phase, defects detected at each phase, time spent in detection and correction, the phase at which each defect is detected and removed, and the classification of defects into types. This data is collected into logs and used to evaluate the quality of the process through the employment of indicators like defect density, review rate, and yield. All these measurements render a highly instrumented process, which is ideal for the realization of empirical studies [Wohlin 00]. The scripts used in PSP include the Process Script, Planning Script, Development Script, Design Review Script, Code Review Script, and Postmortem Script. Every script is composed of a purpose, a set of entry criteria, the activities to perform, and the expected outcomes (i.e., exit criteria).

The Process Script, shown in Table 1, contains a general program for the activities of Planning, Development, and Postmortem. The Development activity, in turn, consists of the phases Design, Design Review, Code, Code Review, Compile, and Unit Testing. Therefore, the Process Script describes the whole process.

*Table 1: Process Script*

**Process Script**

| **Purpose** | To guide the development of module-level programs |
|---|---|
| **Entry Criteria** | - Problem description<br>- PSP Project Plan Summary form<br>- Size Estimating template<br>- Historical size and time data (estimated and actual)<br>- Time and Defect Recording logs<br>- Defect Type, Coding, and Size Counting standards |

| | |
|---|---|
| | - Stopwatch (optional) |

| Step | Activities | Description |
|---|---|---|
| 1 | Planning | - Produce or obtain a requirements statement.<br>- Use the PROBE method to estimate the added and modified size and the size prediction interval of this program.<br>- Complete the Size Estimating template.<br>- Use the PROBE method to estimate the required development time and the time prediction interval.<br>- Complete a Task Planning template.<br>- Complete a Schedule Planning template.<br>- Enter the plan data in the Project Plan Summary form.<br>- Complete the Time Recording log. |
| 2 | Development | - Design the program.<br>- Document the design in the design templates.<br>- Review the design and fix and log all defects found.<br>- Implement the design.<br>- Review the code and fix and log all defects found.<br>- Compile the program and fix and log all defects found.<br>- Test the program and fix and log all defects found.<br>- Complete the Time Recording log. |
| 3 | Postmortem | Complete the Project Plan Summary form with actual time, defect, and size data. |

| | |
|---|---|
| **Exit Criteria** | - A thoroughly tested program<br>- Completed Project Plan Summary form with estimated and actual data<br>- Completed Size Estimating and Task and Schedule Planning templates<br>- Completed Design templates<br>- Completed Design Review and Code Review checklists<br>- Completed Test Report template<br>- Completed Process Improvement Proposal (PIP) forms<br>- Completed Time and Defect Recording logs |

The Planning Script, shown in Table 2, describes the Planning Phase. The goals of this phase are to arrive at a precise definition of the product to be constructed, estimate its size, and, on the basis of personal productivity, estimate the time required for construction. As a method of estimation, PSP uses PROxy Based Estimation (PROBE) [Humphrey 05], which, by employing linear regression on historical data, yields an estimated size in lines of code (LOCs) and estimated time in minutes.

*Table 2: Planning Script*

## Planning Script

| | |
|---|---|
| **Purpose** | To guide the PSP planning process |
| **Entry Criteria** | - Problem description<br>- PSP Project Plan Summary form<br>- Size Estimating, Task Planning, and Schedule Planning templates<br>- Historical size and time data (estimated and actual)<br>- Time Recording log |

| Step | Activities | Description |
|---|---|---|
| 1 | Program Requirements | - Produce or obtain a requirements statement for the program.<br>- Ensure that the requirements statement is clear and unambiguous.<br>- Resolve any questions. |
| 2 | Size | - Produce a program conceptual design. |

| | | |
|---|---|---|
| | Estimate | - Use the PROBE method to estimate the added and modified size of this program.<br>- Complete the Size Estimating template and Project Plan Summary form.<br>- Calculate the 70% size prediction interval. (Note: This step is completed in the SEI student workbook.) |
| 3 | Resource Estimate | - Use the PROBE method to estimate the time required to develop this program.<br>- Calculate the 70% size prediction interval. (Note: This step is completed in the SEI student workbook)<br>- Using the "to-date %" from the most recently developed program as a guide, distribute the development time over the planned project phases. (Note: This step is completed in the SEI student workbook.) |
| 4 | Task and Schedule Planning | For projects lasting several days or more, complete the Task Planning and Schedule Planning templates. |
| 5 | Defect Estimate | - Based on your to-date data on defects per added and modified size unit, estimate the total defects to be found in this program.<br>- Based on your "to-date %" data, estimate the number of defects to be injected and removed by phase. |

| | |
|---|---|
| **Exit Criteria** | - Documented requirements statement<br>- Program conceptual design<br>- Completed Size Estimating template<br>- For projects lasting several days or more, completed Task and Schedule Planning templates<br>- Completed Project Plan Summary form with estimated program size, development time, and defect data, and the time and size prediction intervals<br>- Completed Time Recording log |

The Development Script, shown in Table 3, describes the activities to be carried out at the phases of Design, Design Review, Coding, Code Review, Compilation, and Unit Test.

The Design phase consists of designing the program in a complete and unambiguous manner. PSP makes use of four templates to provide documentation of the design in four dimensions: static, dynamic, internal, and external. In particular, the operational specification template describes the interaction between user and system (i.e., the dynamic-external view). The functional specification template allows the definition of the structural features to be provided by the software product, among them classes and inheritance, externally visible attributes, and relations to other classes or parts (i.e., the dynamic-external and static-external views). The state specification template describes the set of states of the program, the transitions between states, and the actions to be taken at each transition (i.e., the dynamic-internal view). Finally, the logic template specifies the internal logic of the program (i.e., the static-internal view) in a concise and convenient way. Pseudo code is appropriate for this task.

Once the design is completed, PSP proceeds to the Design Review phase, described in the Design Review Script in Table 4. Reviews allow you to find defects prior to the first compilation or test. The Design Review phase includes the following checks, among others: that all requirements are taken into account, that the flow and structure of the program are adequate, and that methods and variables are used correctly.

During the Code phase the program is constructed, employing a programming language and a coding standard.

After this phase, a review of the code is carried out, making use of the Code Review Script shown in Table 5. Code review is a very effective

and inexpensive method for finding defects [Hayes 1997, Vallespir 2012]. Both design and code reviews are carried out with the use of checklists, which are created and maintained by each individual engineer taking into account the defects that he/she usually introduces.

After Code Review is the Compile phase, which is the translation of the source program into machine language using a compiler. The phase involves correcting defects detected by the compiler.

The Unit Test phase consists of the execution of the test cases specified during the Design phase. The defects detected at Unit Test allow the quality of the product to be assessed. In PSP, a program is considered to be of adequate quality if it contains 5 or fewer defects per KLOC at Unit Test.

*Table 3: Development Script*

**Development Script**

| **Purpose** | To guide the development of small programs |
|---|---|
| **Entry Criteria** | - Requirements statement<br>- Project Plan Summary form with estimated program size and development time<br>- For projects lasting several days or more, completed Task Planning and Schedule Planning templates<br>- Time and Defect Recording logs<br>- Defect Type standard and Coding standard |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Design | - Review the requirements and produce an external specification to meet them.<br>- Complete Functional and Operational Specification templates to record this specification.<br>- Produce a design to meet this specification.<br>- Record the design in Functional, Operational, State, and Logic Specification templates.<br>- Record in the Defect Recording log any requirements defects found.<br>- Record time in the Time Recording log. |
| 2 | Design Review | - Follow the Design Review script and checklist and review the design.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 3 | Code | - Implement the design following the Coding standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |
| 4 | Code Review | - Follow the Code Review script and checklist and review the code.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 5 | Compile | - Compile the program until there are no compile errors.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 6 | Test | - Test until all tests run without error.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log.<br>- Complete a Test Report template on the tests conducted and the results obtained. |

| | |
|---|---|
| **Exit Criteria** | - A thoroughly tested program that conforms to the Coding standard<br>- Completed Design templates<br>- Completed Design Review and Code Review checklists<br>- Completed Test Report template<br>- Completed Time and Defect Recording logs |

*Table 4: Design Review Script*

# Design Review Script

| | |
|---|---|
| **Purpose** | To guide you in reviewing detailed designs |
| **Entry Criteria** | - Completed program design documented with the PSP Design templates<br>- Design Review checklist<br>- Design standard<br>- Defect Type standard<br>- Time and Defect Recording logs |
| **General** | Where the design was previously verified, check that the analyses:<br>- covered all of the design<br>- were updated for all design changes<br>- are correct<br>- are clear and complete |

| Step | Activities | Description |
|---|---|---|
| 1 | Preparation | - Examine the program and checklist and decide on a review strategy.<br>- Examine the program to identify its state machines, internal loops, and variable and system limits.<br>- Use a trace table or other analytical method to verify the correctness of the design. |
| 2 | Review | - Follow the Design Review checklist.<br>- Review the entire program for each checklist category; do not try to review for more than one category at a time!<br>- Check off each item as you complete it.<br>- Complete a separate checklist for each product or product segment reviewed. |
| 3 | Fix Check | - Check each defect fix for correctness.<br>- Re-review all changes.<br>- Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space. |

| | |
|---|---|
| **Exit Criteria** | - A fully reviewed detailed design<br>- One or more Design Review checklists for every design reviewed<br>- Documented design analysis results<br>- All identified defects fixed and all fixes checked<br>- Completed Time and Defect Recording logs |

*Table 5: Code Review Script*

# Code Review Script

| | |
|---|---|
| **Purpose** | To guide you in reviewing programs |
| **Entry Criteria** | - A completed and reviewed program design<br>- Source program listing<br>- Code Review checklist<br>- Coding standard<br>- Defect Type standard<br>- Time and Defect Recording logs |

| General | Do the code review with a source-code listing; do not review on the screen! |
|---|---|

| Step | Activities | Description |
|---|---|---|
| 1 | Review | - Follow the Code Review checklist.<br>- Review the entire program for each checklist category; do not try to review for more than one category at a time!<br>- Check off each item as it is completed.<br>- For multiple procedures or programs, complete a separate checklist for each. |
| 2 | Correct | - Correct all defects.<br>- If the correction cannot be completed, abort the review and return to the prior process phase.<br>- To facilitate defect analysis, record all of the data specified in the Defect Recording log instructions for every defect. |
| 3 | Check | - Check each defect fix for correctness.<br>- Re-review all design changes.<br>- Record any fix defects as new defects and, where you know the number of the defect with the incorrect fix, enter it in the fix defect space. |

| Exit Criteria | - A fully reviewed source program<br>- One or more Code Review checklists for every program reviewed<br>- All identified defects fixed<br>- Completed Time and Defect Recording logs |
|---|---|

Finally, the Postmortem Script, shown in Table 6, describes the activities of the Postmortem phase, which includes an assessment of both process and product and an analysis of the injected defects, noting the phases at which they were removed. Analyzing the process and understanding where and why mistakes are committed allows developers to improve their own processes and outputs.

*Table 6: Postmortem Script*

## Postmortem Script

| Purpose | To guide the PSP postmortem process |
|---|---|
| Entry Criteria | - Problem description and requirements statement<br>- Project Plan Summary form with program size, development time, and defect data<br>- For projects lasting several days or more, completed Task Planning and Schedule Planning templates<br>- Completed Test Report template<br>- Completed Design templates<br>- Completed Design Review and Code Review checklists<br>- Completed Time and Defect Recording logs<br>- A tested and running program that conforms to the coding and size counting standards |

| Step | Activities | Description |
|---|---|---|
| 1 | Defect Recording | - Review the Project Plan Summary to verify that all of the defects found in each phase were recorded.<br>- Using your best recollection, record any omitted defects. |
| 2 | Defect Data Consistency | - Check that the data on every defect in the Defect Recording log is accurate and complete.<br>- Verify that the numbers of defects injected and removed per phase are reasonable and correct. |

| | | |
|---|---|---|
| | | - Determine the process yield and verify that the value is reasonable and correct.<br>- Using your best recollection, correct any missing or incorrect defect data. |
| 3 | Size | - Count the size of the completed program.<br>- Determine the size of the base, deleted, modified, base additions, reused, new reusable code, and added parts.<br>- Enter these data in the Size Estimating template.<br>- Determine the total program size.<br>- Enter this data in the Project Plan Summary form. |
| 4 | Time | - Review the completed Time Recording log for errors or omissions.<br>- Using your best recollection, correct any missing or incomplete time data. |

| | |
|---|---|
| **Exit Criteria** | - A thoroughly tested program that conforms to the coding and size counting standards<br>- Completed Design templates<br>- Completed Design Review and Code Review checklists<br>- Completed Test Report template<br>- Completed Project Plan Summary form<br>- Completed PIP forms describing process problems, improvement suggestions, and lessons learned<br>- Completed Time and Defect Recording logs |

# 2.3 Formal Methods

Formal methods hold fast to the tenet that programs should be *proven* to satisfy their specifications. Proof is the mathematical activity of arriving at knowledge deductively, starting with postulated, supposed, or self-evident principles and performing successive inferences, each of which extracts a conclusion out of previously arrived-at premises.

In applying this practice to programming, the first principle is the semantics of programs. Semantics allows us to understand program code and know what each part of the program actually computes. This makes it possible, in principle, to deductively ascertain that the computations carried out by the program satisfy certain properties. Among these properties are input-output relationships or patterns of behavior that constitute a precise formulation of the functional specification of the program or system at hand.

Formal logic, at least in its contemporary mathematical variety, strives to formulate artificial languages that frame the mathematical activity. According to this aim, there should be a language for expressing every conceivable mathematical proposition and also a language for expressing proofs, so that a proposition is provable in this language if and only if it is actually true. This latter desirable property of the language is called its correctness. This kind of research began in 1879 with Frege for the purpose of making it undisputable whether a proposition was correctly proven or not [Frege 1967]. Indeed, the whole point of devising artificial languages was to make it possible to automatically check whether a proposition or a proof was correctly written in the language. The proofs were to be accepted on purely syntactic (i.e., formal) grounds and, given the "good" property of correctness of the language, that was enough to ensure the truth of the asserted propositions.

Frege's own language turned out to be not correct and shortly after its failure the whole enterprise of formal logic took a different direction, shifting toward the study of artificial languages as mathematical objects in order to prove their correctness by elementary means. This new course was also destined to failure.

The overall outcome is nevertheless very convenient from an engineering viewpoint. Using the technology we now have available, we can go back to Frege's programs and develop formal proofs semi-automatically. The proof systems (or languages) are still reliable, although they are not complete (i.e., not every true proposition will be provable). But this is no harm in practice and the systems are perfectly expressive from an engineering perspective. All these advances allow us to define formal methods in software engineering as a discipline based on the use of formal languages and related tools for expressing specifications and carrying out proofs of correctness of programs.

Note that the semi-automatic process of program correctness proof is a kind of static checking. We can think of it as an extension of compilation, which not only checks syntax but also properties of functional behavior. Therefore it is convenient to employ the general idea of a semi-automatic verifying compiler to characterize the functionality of the tools employed within a formal methods framework.

Design by Contract (DbC) is a methodology for designing software based on the idea that specifications of software components arise, like business contracts, from agreements between a user and a supplier, who establish the terms of use and performance of the components. That is to say, specifications oblige (and enable) both the user and the supplier to certain conditions of use and a corresponding behavior of the component in question.

In particular, DbC has been proposed in the framework of object-oriented design (and specifically in the language Eiffel) and therefore the software components to be considered are usually classes. The corresponding specifications are pre- and post-conditions to methods, establishing respectively their terms of use and corresponding outcomes, as well as invariants of the class (i.e., conditions to be verified by every visible state of an instance of the class). In the original DbC proposal, all the specifications were written in Eiffel and are computable (i.e., they are checkable at runtime).

Therefore, DbC in Eiffel provides at least the following:

- a notation for expressing the design that seamlessly integrates with a programming language, making it easy to learn and use
- formal specifications, expressed as assertions in Floyd-Hoare style [Hoare 1969]
- specifications checkable at runtime and whose violations may be handled by a system of exceptions
- automatic software documentation

However, DbC is not by itself an example of a formal method, as defined above. When we additionally enforce proving that the software components fit their specifications, we are using Verified Design by Contract (VDbC). This can be carried out within several environments, all of which share the characteristics mentioned above, including the following:

- Java Modeling Language (JML) implements DbC in Java. VDbC can then be carried out using tools like Extended Static Checking (ESC/Java) [Cok 2005] or TACO [Galeotti 2010].
- Perfect Developer [Crocker 2003] is a specification and modeling language and tool which, together with the Escher C Verifier allow performing VDbC for C and C++ programs.
- Spec# [Barnett 2004] allows VDbC within the C# framework.
- Modern Eiffel [Eiffel 2012] within the Eiffel framework.

# 2.4 Adaptation

In this section we describe the $PSP_{VDC,}$ which introduces new phases as well as modifying others already present in the ordinary PSP. In each case we describe in detail the corresponding activities and show the new scripts. Figure 2 shows the $PSP_{VDC}$. We assume the engineer will be using an environment similar to those listed at the end of the previous section, meaning that a computerized tool (akin to a verifying compiler) is used for

- checking the syntax of formal assertions. These are written in the language employed in the environment (e.g., as Java Boolean expressions, if JML is used) which we call the *carrier* language.
- computing proof obligations (i.e., given code with assertions, to establish the list of conditions that need to be proven in order to ascertain the correctness of the program)
- developing proofs in a semi-automatic way

The elements of Figure 2 described below summarize the most relevant novelties of $PSP_{VDC}$.

- After the Design Review phase, a new phase of Test Case Construct is added. This phase is used to determine the set of test cases to use in the validation of the program.
- After the Test Case Construct phase, a new phase called Formal Specification is added. In this phase the design is formalized, in the sense that class invariants and pre- and post-conditions to methods are made explicit and formal (in the carrier language).
- The Formal Specification Review is used to detect defects injected in the formal specification produced in the previous phase. A review script is used for this activity.
- The Formal Specification Compile phase consists of automatically checking the formal syntax of the specification.
- The Pseudo Code phase consists of writing down the pseudo code of every method.
- The Pseudo Code Review phase consists of precisely reviewing the pseudo code produced in the former phase.
- The Proof phase comes after production, review, and compilation of the code. The general idea is to supplement the design with formal specifications of the components and the code with a formal proof to show that it matches the formal specifications. This proof is to be carried out with the help of a tool akin to a verifying tool, in the sense that it is employed to statically check the logical correctness of the code besides its syntactic well-formedness.

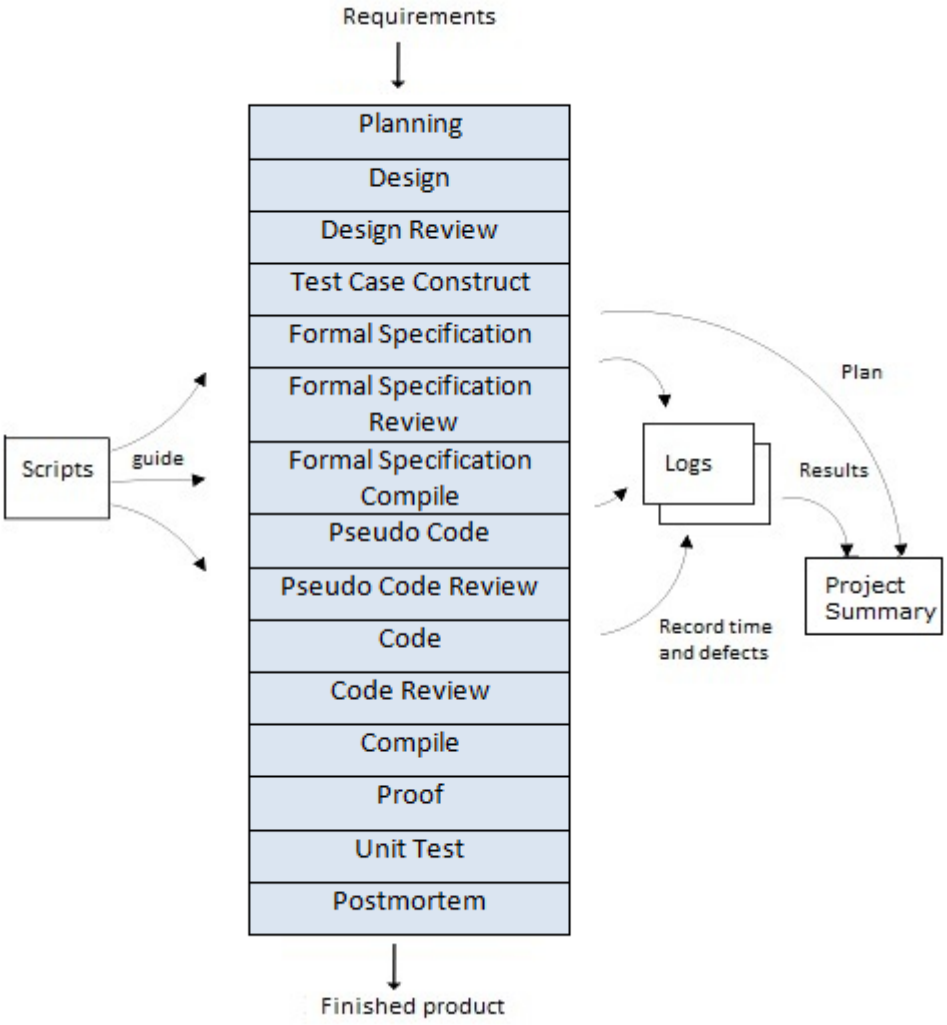



*Figure 2: Phases of the $PSP_{VDC}$*

In the following subsections we present in detail all the phases of the $PSP_{VDC}$, indicating in each case the activities to be performed and the modifications introduced in the scripts with respect to the original PSP.

# Planning

The activities of the Planning phase in $PSP_{VDC}$ are the same as in ordinary PSP: Program Requirements, Size Estimate, Resource Estimate, Task and Schedule Planning, and Defect Estimate.

*Program Requirements* is for ensuring a precise understanding of every requirement. This activity is the same as in the ordinary PSP.

*Size Estimate* involves carrying out a conceptual design (i.e., producing a module (class) structure). Each class is refined into a list of methods and the relative size of the methods of each class is estimated. This is done in the same way as in ordinary PSP: using proxies to create a categorization of the method according to its size and the functional type of the corresponding class. Categories of size of methods include very small, small, medium, large, or very large; functional kinds of classes include Calc, Logic, IO, Set-Up, and Text. Thus, using the structure of classes, the number of methods (and the size) in each class and the category of the class, we arrive at an estimation of the LOCs of the program.

PSP uses LOCs for estimating the size of the program and deriving from that an estimation of the effort required for its construction. It has been established that under certain conditions the effort is proportional to the LOCs of the program [Humphrey 05]. $PSP_{VDC}$ requires engineers to formally write down the pre- and post-conditions of each method and the invariant of each class, which is a kind of output akin to LOCs and could certainly increase the total cost of development. Nevertheless, we also continue measuring the size of the product in LOCs and postulate that the relationship between effort and size in LOCs will keep valid. It will depend on the outcome of empirical studies whether we should adjust this premise and consider also Lines of Formal Specification (LOFs) for effort estimation. Note that, for estimating LOFs, it will be necessary to specify what exactly a LOF is, which will give a criterion for counting them. It will also be necessary to use a proxy for LOF estimation. The development of the corresponding techniques is out of the scope of the present work. Because of these considerations, the activity of size estimate remains unchanged in $PSP_{VDC.}$

*Resource Estimate* estimates the amount of time needed to develop the program. For this, the PROBE method is used, which employs historical records and linear regression for producing the new estimation and for measuring and improving the precision of the estimations. In our adaptation, the activity remains conceptually the same, but will employ records associated to the new phases incorporated into $PSP_{VDC}$. Therefore, once sufficient time data has been gathered, we will be able to estimate the effort (measured in minutes) required for the formal specification and for the program proof.

*Task and Schedule Planning* is for long-term projects. These are subdivided into tasks and the time is estimated for each task. This is unchanged in $PSP_{VDC}$.

*Defect Estimate Base* is for estimating the number of defects injected and removed at each phase. Historical records and the estimated size of the program are utilized for performing this estimation. In $PSP_{VDC}$ new records are needed to estimate the defects removed and injected at each new phase.

Finally, the Planning Script in $PSP_{VDC}$ is the same as in PSP, given that the corresponding activities are unchanged.

# Design

During Design, the data structures of the program are defined, as well as its classes and methods, interfaces, components, and the interactions among all of them. In PSP, elaboration of the pseudo code is also included. In $PSP_{VDC}$ the elaboration of the pseudo code is postponed until the formal specification is available for each method. Therefore, we eliminate from the Design phase the use of the Logic Template, which corresponds to the pseudo code. The Logic Template ceases to be a member of the set of templates of the Design Template, given that in $PSP_{VDC}$ it is not a design template anymore.

Normally, although not specified in PSP, the Design phase also includes the design of the test cases. In $PSP_{VDC}$ we propose a test case design in a phase separate from the Design phase because we are interested in getting information about the time employed specifically in the construction of test cases. As explained below, such knowledge will be useful in comparing the cost of using formal methods versus that of testing and debugging.

Formal specification of methods and of invariants of classes could be carried out within the Design phase. This, however, does not allow us to keep records of the time employed specifically in Design as well as in Formal Specification. Instead, we would just record a likely significant increase in Design time. Therefore we prefer to separate the phase of Formal Specification.

The changes to process scripts appear in red text; deletions are marked with strikethrough. In sum, the activity of Design within the Development Script is modified to

| Step | Activities | Description |
|---|---|---|
| 1 | Design | - Review the requirements and produce an external specification to meet them.<br>- Complete Functional and Operational Specification templates to record this specification.<br>- Produce a design to meet this specification.<br>- Record the design in Functional, Operational, and State, ~~and Logic Specification~~ templates.<br>- Record in the Defect Recording log any requirements defects found.<br>- Record time in the Time Recording log. |

# Design Review

This is the same as in ordinary PSP and uses its Development Script.

# Test Case Construct

We want to investigate the cost effectiveness of test case construction and unit testing when formal methods are used. That is, is it practical to eliminate the Unit Test phase when using these formal methods? To answer this, we need to know

- The cost of test case construction
- The cost of unit test execution
- The defect density entering into unit test
- The yield of the unit test phase

This will also allow us to assess the economic and quality benefits of implementing VDbC using PSP. The Test Case Construct activity is incorporated into the Development Script as detailed below:

| Step | Activities | Description |
|---|---|---|
| 3 | Test Case Construct | - Design test cases and record them in the Test Report.<br>- Record time in the Time Recording log. |

# Formal Specification

This phase must be performed after Design Review. The reason for this is that reviews are very effective in detecting defects injected during design and we want to discover them as early as possible.

In this phase we start to use the computerized environment supporting VDbC. Two activities are carried out in this phase: Construction and Specification. Construction consists of preparing the computerized environment and defining within it each class with its method headers. If this is instead be done during Design as part of the functional template, omit it here. The choice is a personal one.

The second activity is Specification, in which we write down in the carrier language the pre- and post-conditions of each method as well as the class invariant. Note that, within the present approach, the use of formal methods begins once the design has been completed. It consists of the formal specification of the produced

design and the formal proof that the final code is correct with respect to this specification.

Formal Specification is incorporated into the Development Script. A standard template for the specification is used in this activity. Table 7 presents an example for the language JML.

*Table 7: Formal Specification Standard Template*

| Step | Activities | Description |
|---|---|---|
| 4 | Formal Specification | - Implement the design following the Formal Specification standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |

| | |
|---|---|
| Purpose | To guide the formal specification of programs |
| Program Headers | Begin all programs with a descriptive header. The header should use the Java documentation commenting convention ("/**") so automated documentation generation is possible. Include in the descriptive header the name of the author who writes the formal specification and a version number . |
| Header Format | /**<br>* @formal specification author Philip Johnson<br>* @formal specification version Tue Dec 26 2011<br>*/ |
| Identifiers | Use descriptive names for all variables, constants, and other identifiers. Avoid abbreviations or single letter variables. |
| Identifier Example | //@ public constraint age >= \old(age); //this is good<br>//@ public constraint i >= \old(i); //this is bad |
| Comments | Document the code so that the reader can understand its operation. Comments should explain both the purpose and behavior of the code. Comment variable declarations to indicate their purpose. |
| Good Comment | /*@ requires array != null;<br>@ ensures (* return the sum of the array elements *)<br>@ && \result == (\sum int I; 0 <= I && I < array.length; array[I]);<br>@ ensures (* without modifying the array *)<br>@ && (\forall int I; 0 <= I && I < array.length;<br>@ array[I] == \old(array[I]));<br>@*/ |
| Bad Comment | This comment is wrong:<br>/*@<br>@ ( * comment * ) assertion<br>@*/<br><br>This comment is OK:<br>/*@<br>@ ( * comment * ) && assertion<br>@*/<br>Comments are treated as assertions; therefore, they should be connected to other assertions by means of &&. |
| Indenting | Indent every level of brace from the previous one. |

| | |
|---|---|
| Indenting Example | /*@ public normal_behavior<br>@ requires divisor > 0;<br>@ ensures divisor*\result <= dividend<br>@ && divisor*(\result+1) > dividend;<br>@<br>@ also<br>@ public normal_behavior<br>@ requires divisor == 0;<br>@ ensures \result == 0;<br>@*/ |
| Capitalization | • Always use lower case in variable declarations.<br>• Use upper case for types and clases.<br>• Use upper case in invocations of a method so declared or of a JML library. |
| Capitalization Example | /*@ public model String name;<br>@ public represents name <- getName();<br>@<br>@ public invariant !"".equals(name);<br>*/ |

# Formal Specification Review

Using a formal language for specifying conditions is not a trivial task, and both syntactic and semantic defects can be injected. To avoid the propagation of these errors to further stages and the resulting increase in the cost of correction, we propose a phase called Formal Specification Review.

The script that corresponds to this phase contains these activities: Review, Correction, and Checking. The Review activity consists of inspecting the sentences of the specification using a checklist. In the Correction activity, all defects detected during Review are removed. Finally, Checking consists of looking over the corrections to verify their adequacy.

The Formal Specification Review activity is incorporated into the Development Script; the Formal Specification Review Script and Formal Specification Review Checklist are proposed for use in this activity.

*Table 8: Specification Review Script*

| Step | Activities | Description |
|---|---|---|
| 5 | Formal Specification Review | - Follow the Formal Specification Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |

| | |
|---|---|
| Purpose | To guide you in reviewing detailed designs |
| Entry Criteria | Specification Review checklist<br>Defect Type standard<br>Time and Defect Recording logs |
| General | Where the Specification was previously verified, check that the analyses covered all of the Specification, were updated for all Specification changes, and are clear and complete. |

| Step | Activities | Description |
|---|---|---|
| 1 | Preparation | Examine the specification and checklist and decide on a review strategy. |

| | | |
|---|---|---|
| 2 | Review | Follow the Specification Review checklist.<br>Review the entire specification for each checklist category; do not try to review for more than one category at a time!<br>Check off each item as you complete it.<br>Complete a separate checklist for each product or product segment reviewed. |
| 3 | Fix Check | Check each defect fix for correctness.<br>Re-review all changes.<br>Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space. |

| | |
|---|---|
| Exit Criteria | A fully reviewed detailed Specification<br>One or more Specification Review checklists for every specification reviewed<br>Documented Specification analysis results<br>All identified defects fixed and all fixes checked<br>Completed Time and Defect Recording logs |

*Table 9: Formal Specification Review Checklist Template*

| | | | |
|---|---|---|---|
| Student | | Date | |
| Program | | Program # | |
| Instructor | | Language | |
| Formal Specification Language | | | |

| | |
|---|---|
| Purpose | To guide you in conducting an effective specification review |
| General | Review the entire specification for each checklist category; do not attempt to review for more than one category at a time!<br>As you complete each review step, check off that item in the box at the right.<br>Complete the checklist for one specification or specification unit before reviewing the next. |
| General | To verify that the formal specification adequately complements the design |

| | | | | | |
|---|---|---|---|---|---|
| Assertions | Assertions are prefixed by //@ or appear between /*@ … @*/<br>Every assert clause must end in ;.<br>Verify that the variable associated to each clause \forall, \sum, \exists, etc. is appropriately initialized.<br>In each clause \forall, \sum, \exists, etc. verify balance of parentheses in IF, ELSE, FOR, WHILE.<br>In each clause \forall, \sum, \exists, etc. verify that the appropriate segment of the array is traversed.<br>Verify that every method invoked within an assertion is declared as /*@ pure @*/. | | | | |
| Preconditions | Method preconditions are declared by means of the requires clause. | | | | |
| Postconditions | Method postconditions are declared by means of the ensures clause. | | | | |
| Class Invariants | Class invariants are declared by means of the invariant clause. | | | | |

# Formal Specification Compile

Any computerized tool supporting VDbC will be able to compile the formal specification. Since this allows an early detection of errors, we consider it valuable to explicitly introduce this phase into $PSP_{VDC}$. In particular, it is worthwhile to detect all

possible errors in the formal specifications before any coding is carried out. A further reason to isolate the compilation of the formal specification is to allow the time spent in this specific activity to be recorded.

The activity Formal Specification Compile is added to the Development Script.

| Step | Activities | Description |
|---|---|---|
| 6 | Formal Specification Compile | - Compile the formal specification until there are no compile errors.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |

# Pseudo Code

The Pseudo Code phase allows us to understand and structure the solution to the specified problem just before coding. The pseudo code of each class method defined in the Logic Template is written down.

We propose that the pseudo code be produced after the compilation of the specification in order for the specification to serve as a well understood starting point for design elaboration in pseudocode. Writing down the pseudo code just before coding allows us to follow a well-defined process in which the output of each stage is taken as input to the next one.

The activity Pseudo Code is incorporated into the Development Script.

| Step | Activities | Description |
|---|---|---|
| 7 | Pseudo Code | - Produce a Pseudo Code to meet the design.<br>- Record the Design Logic Specification templates.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |

# Pseudo Code Review

A check list is used for guiding the activity in this phase. The activity Pseudo Code Review is added to the Development Script. The Pseudo Code Review script is proposed for use in this activity.

| Step | Activities | Description |
|---|---|---|
| 8 | Pseudo Code Review | - Follow the Pseudo Code Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |

# Code, Code Review, and Code Compile

Just as in ordinary PSP, these phases consist of translating the design into a specific programming language, revising the code, and compiling it. The descriptions of these activities in the $PSP_{VDC}$ Development Script are the same as in the PSP Development Script.

# Proof

This phase is added in $PSP_{VDC}$ to provide evidence of the correctness of the code with respect to the formal specification (i.e., its formal proof). A computerized

verifying tool is used which derives proof obligations and helps to carry out the proofs themselves.

The description of the activity Proof within the Development Script is as follows.

| | | |
|---|---|---|
| 12 | Proof | - Construct a formal proof of correctness of the code with respect to the formal specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |

# Unit Test

This phase is the same as in ordinary PSP. We consider it relevant for detecting mismatches with respect to the original, informal requirements of the program. These defects can arise at several points during the development, particularly as conceptual or semantic errors of the formal specifications. The test cases to be executed must therefore be designed right after the requirements are established (i.e., during the phase Test Case Construct) as already indicated.

The description of this activity in the $PSP_{VDC}$ Development Script is the same as in the PSP Development Script.

# Post-Mortem

This is the same as in ordinary PSP and its description in the $PSP_{VDC}$ Development Script is the same as in the PSP Development Script.

However, several modifications have to be made to the infrastructure supporting the new process. For instance, all new phases must be included in the support tool to keep track of the time spent at each phase, as well as to record defects injected, detected, and removed at each phase. Our intention in this paper is to present the changes in the process in order to incorporate VDbC. The adaptation of the supporting tools, scripts, and training courses is a matter for a separate work.

We have now completed the description of the modifications made to each phase of the PSP to turn it into $PSP_{VDC}$. In Table 10 we present the $PSP_{VDC}$ Process Script. This contains some modifications due to the changes made to the Development Script. In Table 11 we present the complete $PSP_{VDC}$ Development Script. In the Appendix, all scripts and templates of $PSP_{VDC}$ are shown.

*Table 10: The Process Script*

| **Purpose** | To guide the development of module-level programs |
|---|---|
| **Entry Criteria** | - Problem description<br>- PSP Project Plan Summary form<br>- Size Estimating template<br>- Historical size and time data (estimated and actual)<br>- Time and Defect Recording logs<br>- Defect Type, Coding, and Size Counting standards<br>- Stopwatch (optional) |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Planning | - Produce or obtain a requirements statement.<br>- Use the PROBE method to estimate the added and modified size and the size prediction interval of this program.<br>- Complete the Size Estimating template.<br>- Use the PROBE method to estimate the required development time and the time prediction interval.<br>- Complete a Task Planning template. |

| | | |
|---|---|---|
| | | - Complete a Schedule Planning template.<br>- Enter the plan data in the Project Plan Summary form.<br>- Complete the Time Recording log. |
| 2 | Development | - Design the program.<br>- Document the design in the design templates.<br>- Review the design, and fix and log all defects found.<br>- Design the test cases.<br>- Formally specify all methods of the classes introduced in design.<br>- Review formal specification and fix and log all defects found.<br>- Compile formal specification and fix and log all defects found.<br>- Write down pseudo code using the Logic Template.<br>- Review pseudo code and fix and log all defects found.<br>- Implement the design.<br>- Review the code and fix and log all defects found.<br>- Compile the program and fix and log all defects found.<br>- Construct formal proof of correctness of code with respect to its formal specification.<br>- Test the program and fix and log all defects found.<br>- Complete the Time Recording log. |
| 3 | Postmortem | Complete the Project Plan Summary form with actual time, defect, and size data. |

| **Exit Criteria** | - A thoroughly tested program<br>- Completed Project Plan Summary form with estimated and actual data<br>- Completed Size Estimating and Task and Schedule Planning templates<br>- Completed Design templates and Formal Specification Templates<br>- Project or other processing unit containing formal proof of code correctness. (This depends on the concrete computerized tool employed.)<br>- Completed Design Review, Formal Specification Review, Pseudo Code Review and Code Review checklists<br>- Completed Test Report template<br>- Completed PIP forms<br>- Completed Time and Defect Recording logs |
|---|---|

*Table 11: Development Script*

| **Purpose** | To guide the development of small programs |
|---|---|
| **Entry Criteria** | - Requirements statement<br>- Project Plan Summary form with estimated program size and development time<br>- For projects lasting several days or more, completed Task Planning and Schedule Planning templates<br>- Time and Defect Recording logs<br>- Defect Type standard and Coding standard |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Design | - Review the requirements and produce an external specification to meet them.<br>- Complete Functional and Operational Specification templates to record this specification.<br>- Produce a design to meet this specification.<br>- Record the design in Functional, Operational, State, ~~and Logic Specification~~ templates.<br>- Record in the Defect Recording log any requirements defects found.<br>- Record time in the Time Recording log. |
| 2 | Design | - Follow the Design Review script and checklist and review the |

| | | |
|---|---|---|
| | Review | design.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 3 | Test Case Construct | - Design test cases and record them in the Test Report.<br>- Record time in the Time Recording log. |
| 4 | Formal Specification | - Implement the design following the Formal Specification standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |
| 5 | Formal Specification Review | - Follow the Formal Specification Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 6 | Formal Specification Compile | - Compile the formal specification until there are no compile errors.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |
| 7 | Pseudo Code | - Produce a Pseudo Code to meet the design.<br>- Record the Design Logic Specification templates.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |
| 8 | Pseudo Code Review | - Follow the Pseudo Code Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 9 | Code | - Implement the design following the Coding standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |
| 10 | Code Review | - Follow the Code Review script and checklist and review the code.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 11 | Compile | - Compile the program until there are no compile errors.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 12 | Proof | - Construct formal proof of correctness of the code with respect to its formal specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 13 | Test | - Test until all tests run without error.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log.<br>- Complete a Test Report template on the tests conducted and the results obtained. |

| | |
|---|---|
| **Exit Criteria** | - A thoroughly tested program that conforms to the Coding standard<br>- A formal specification conforming to the Formal Specification standard<br>- Completed Design and Formal Specification templates<br>- Completed Design Review, Pseudo Code Review, Formal Specification Review, and Code Review checklists |

| | |
|---|---|
| | - Completed Test Report template<br>- Completed Time and Defect Recording logs |

# 2.5 Quality Planning

Quality planning in PSP includes the following:

- estimating the total number of defects injected and removed
- estimating the number of defects injected and removed at each phase
- estimating the time required at each phase

In this section we present the modifications to Quality Planning introduced in $PSP_{VDC}$.

For estimating the total number of defects injected, PSP uses the estimation of the size of the program as well as historical data about the amount of defects injected per KLOC. For estimating the number of defects injected and removed at each phase, PSP performs a distribution of the total estimate, making use of historical data.

In $PSP_{VDC}$ the new phases must be taken into account in order to perform the corresponding estimations of the number of defects and of the time required. Initially, the corresponding historical data mentioned above is not available. Therefore, the initial estimation must be done by applying expert judgment. After performing several studies, accumulated data is available for employment in the desired estimations.

In PSP some benchmarks are known that also can be used for estimating the number of defects removed. In particular, from the PSP data the following rates of defect removal are known, which usually indicate good use of the process:

- 3 to 5 defects per hour in design review
- 5 to 10 defects per hour in code review

Eventually $PSP_{VDC}$ use will produce useful benchmarks for the Formal Specification Review and Pseudo Code Review phases.

In PSP the Process Quality Indicator (PQI) suggests the following values for code and design reviews:

- the time employed in design review is not less than 50% of the time employed in design
- the time employed in the code review is not less than 50% of the time employed in coding

We are interested in obtaining, by empirical means, a relationship between the time required by the formal specification and that required by its review. Similar information is desired for the pseudo code.

## 2.6 Quality Measures

Product quality is an essential issue in PSP. Developers must remove defects, determine the causes of their injection, and learn to prevent them from occurring. PSP proposes reviews as a recommended method for defect removal because it is even more effective than testing. [Hayes1997, Vallespir 11, Vallespir 12]. To perform efficient reviews it is necessary to make measurements [Gilb 1993].

PSP defines several measurements of process quality and control, including the following:

- yield
- defect removal efficiency
- defect removal leverage
- cost of quality (COQ)

The yield of a phase is defined as the percentage of defects found at the phase in question over the total number of defects that enter the phase. It is usually employed for measuring the effectiveness of design and code reviews, as well as of compilation and testing. It can be used in $PSP_{VDC}$ for measuring the effectiveness of the new phases of formal specification review (FSR) and pseudo code review (PCR), formal specification, compile, and proof.

The yield of the process is calculated as the percentage of defects injected and removed prior to the first code compilation. In $PSP_{VDC}$, this must be adjusted by taking into account the new phases that precede the compilation phase.

$$\text{Yield (process)} = 100 \cdot \frac{\text{Defects removed before code compile}}{\text{Defects injected before code compile}}$$

Defect removal efficiency is the number of defects removed per hour at the phases of Design Review, Code Review, Compile, and Test. In $PSP_{VDC}$ it is important to also know the number of defects removed per hour in the phases of Formal Specification Review, Pseudo Code Review, Formal Specification Compile (FSC), and Proof (PRF). Defect removal efficiency for such cases is defined as follows:

$$\text{Defect removal efficiency (FSR)} = 60 \cdot \frac{\text{Defects removed in FSR}}{\text{Time in FSR (minutes)}}$$

$$\text{Defect removal efficiency (FSC)} = 60 \cdot \frac{\text{Defects removed in FSC}}{\text{Time in FSC (minutes)}}$$

$$\text{Defect removal efficiency (PCR)} = 60 \cdot \frac{\text{Defects removed in PCR}}{\text{Time in PCR (minutes)}}$$

$$\text{Defect removal efficiency (PRF)} = 60 \cdot \frac{\text{Defects removed in PRF}}{\text{Time in PRF (minutes)}}$$

Defect removal leverage is the number of defects removed per hour at one stage of the process with respect to a base phase. Normally, the base phase is Unit Test (UT). In $PSP_{VDC}$ we propose to incorporate the indicators DRL (FSR/UT), DRL (PCR/UT), DRL (FSC/UT), and DRL (PRF/UT), which correspond to the number of defects per hour removed at FSR, PCR, FSC, and PRF respectively, with respect to the UT phase.

Cost of quality (COQ) is a measure of process quality. The components of COQ are failure, appraisal, and prevention costs. Failure cost is the time dedicated to

repair and re-work, which corresponds in PSP to the phases of Compile and Test. Appraisal cost is the time spent in inspection, which in PSP is the time spent at the phases of Design and Code Review. Defect prevention is the time dedicated to the identification and resolution of the causes of the defects.

With the same idea, in $PSP_{VDC}$ failure cost corresponds to the time employed in the phases of Code Compilation, Formal Specification Compile, Proof, and Test. The appraisal cost, on the other hand, is the time spent at the phases of Design and Code Review, Formal Specification Review, and Pseudo Code Review.

The indicator Appraisal Cost of Quality (% Appraisal COQ) is defined in PSP as the percentage of the total development time employed in design and code review. High values of this indicator are associated to low number of defects in testing and high quality of the product. We modify this indicator in $PSP_{VDC}$ in order to incorporate the time employed in review of the formal specification and of the pseudo code. Therefore, the corresponding formula becomes

$$\%\,\mathrm{Appraisal\,COQ} = 100 \cdot \frac{\mathrm{Design\ Review\ Time} + \mathrm{Code\ Review\ Time} + \mathrm{FSR\ Time} + \mathrm{PCR\ Time}}{\mathrm{Total\,Development\ Time}}$$

The indicator Percent Failure COQ (% Failure Cost of Quality) is defined in PSP as the percentage of the total development time employed in compilation and testing. We modify it in $PSP_{VDC}$ in order to incorporate the time spent in compilation of the formal specification (FSC) and the time spent in making the Proof. We thus rewrite the formula as

$$\%\,\mathrm{Failure\ COQ} = 100 \cdot \frac{\mathrm{Code\,Compile\ Time} + \mathrm{Test\ Time} + \mathrm{FSC\ Time} + \mathrm{Proof\ Time}}{\mathrm{Total\,Development\ Time}}$$

A useful COQ measurement is the rate between appraisal and failure costs (A/FR). This indicator is only implicitly modified in $PSP_{VDC}$ because of the changes in A and FR.

In PSP, a value of A/FR greater than 2 is considered an indicator of high performance. This benchmark value must be adjusted in $PSP_{VDC}$ after performing empirical studies because of the possible impact of the incorporated phases.

# 2.7 Conclusions and Future Work

This paper has described $PSP_{VDC}$, a combination of PSP with Verified Design by Contract (VDbC), with the aim of developing better quality products.

In summary, we propose to supplement the design with formal specifications of the pre- and post-conditions of methods as well as class invariants. This gives rise to seven new phases which come after the Design phase, namely Test Case Construct, Formal Specification, Formal Specification Review, Formal Specification Compile, Pseudo Code, Pseudo Code Review, and Proof. We also propose to verify the logical correctness of the code by using an appropriate tool, which we call a *verifying compiler*. This motivates the new Proof phase, which provides evidence of the correctness of the code with respect to the formal specification.

The process can be carried out within any of several available environments for VDbC.

By definition, in Design by Contract (and thereby, also in VDbC) the specification language is seamlessly integrated with the programming language, either because they coincide or because the specification language is a smooth extension of the programming language. As a consequence, the conditions making up the various specifications are Boolean expressions that are simple to learn and understand. We believe that this makes the approach easier to learn and use than the ones in other proposals [Babar 2005, Suzumori 2003]. Nonetheless, the main difficulty associated with the method resides in developing a competence in carrying out the formal proofs of the written code. This is, of course, common to any approach based on formal methods. Experience shows, however, that the available tools are generally of great help in this matter. There are reports of cases in which the tools have generated the proof obligations and discharged up to 90% of the proofs automatically [Abrial 2006].

We conclude that it is possible in principle to define a new process which integrates the advantages of both PSP and formal methods, particularly VDbC. In our future work, we will evaluate the $PSP_{VDC}$ in actual practice by carrying out measurements in empirical studies. The fundamental aspect to be measured in our evaluation is the quality of the product, expressed in the amount of defects injected and removed at the various stages of development. We are also interested in measures of the total cost of the development.

# Appendix

In this section we present the Process Script, the Development Script, the Formal Specification Standard Template, the Specification Review Script, and Formal Specification Review Checklist Template.

*Table 12: Process Script*

| **Purpose** | To guide the development of module-level programs |
|---|---|
| **Entry Criteria** | - Problem description<br>- PSP Project Plan Summary form<br>- Size Estimating template<br>- Historical size and time data (estimated and actual)<br>- Time and Defect Recording logs<br>- Defect Type, Coding, and Size Counting standards<br>- Stopwatch (optional) |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Planning | - Produce or obtain a requirements statement.<br>- Use the PROBE method to estimate the added and modified size and the size prediction interval of this program.<br>- Complete the Size Estimating template.<br>- Use the PROBE method to estimate the required development time and the time prediction interval.<br>- Complete a Task Planning template.<br>- Complete a Schedule Planning template.<br>- Enter the plan data in the Project Plan Summary form.<br>- Complete the Time Recording log. |
| 2 | Development | - Design the program.<br>- Document the design in the design templates.<br>- Review the design and fix and log all defects found.<br>- Design Test cases.<br>- Formally specify the methods of every class introduced at design.<br>- Review the formal specification and fix and log all defects found.<br>- Compile the formal specification and fix and log all defects found.<br>- Write down the pseudo code, using the Logic Template.<br>- Review the pseudo code and fix and log all defects found.<br>- Implement the design.<br>- Review the code and fix and log all defects found.<br>- Compile the program and fix and log all defects found.<br>- Construct a formal proof of correctness of the code with respect to its formal specification.<br>- Test the program and fix and log all defects found.<br>- Complete the Time Recording log. |
| 3 | Postmortem | Complete the Project Plan Summary form with actual time, defect, and size data. |

| **Exit Criteria** | - A thoroughly tested program<br>- Completed Project Plan Summary form with estimated and actual data<br>- Completed Size Estimating and Task and Schedule Planning templates<br>- Completed Design templates and Formal Specification Templates<br>- Completed Design Review, Formal Specification Review, Pseudo Code Review, and Code Review checklists<br>- Completed Test Report template<br>- Completed PIP forms |
|---|---|

| | |
|---|---|
| | - Completed Time and Defect Recording logs |

*Table 13: Development Script*

| **Purpose** | To guide the development of small programs |
|---|---|
| **Entry Criteria** | - Requirements statement<br>- Project Plan Summary form with estimated program size and development time<br>- For projects lasting several days or more, completed Task Planning and Schedule Planning templates<br>- Time and Defect Recording logs<br>- Defect Type standard and Coding standard |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Design | - Review the requirements and produce an external specification to meet them.<br>- Complete Functional and Operational Specification templates to record this specification.<br>- Produce a design to meet this specification.<br>- Record the design in Functional, Operational, and State templates.<br>- Record in the Defect Recording log any requirements defects found.<br>- Record time in the Time Recording log. |
| 2 | Design Review | - Follow the Design Review script and checklist and review the design.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 3 | Test Case Construct | - Design test cases and record them in the TestReport.<br>- Record time in the Time Recording log. |
| 4 | Formal Specification | - Implement the design following the Formal Specification standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |
| 5 | Formal Specification Review | - Follow the Formal Specification Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 6 | Formal Specification Compile | - Compile the formal specification until there are no compile errors.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |
| 7 | Pseudo Code | - Produce a Pseudo Code to meet the design.<br>- Record the design Logic Specification templates.<br>- Record in the Defect Recording log any defects found.<br>- Record time in the Time Recording log. |
| 8 | Pseudo Code Review | - Follow the Pseudo Code Review script and checklist and review the specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 9 | Code | - Implement the design following the Coding standard.<br>- Record in the Defect Recording log any requirements or design defects found.<br>- Record time in the Time Recording log. |
| 10 | Code Review | - Follow the Code Review script and checklist and review the code. |

| | | |
|---|---|---|
| | | - Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 11 | Compile | - Compile the program until there are no compile errors.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 12 | Proof | - Construct a formal proof of correctness of the program with respect to the formal specification.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log. |
| 13 | Test | - Test until all tests run without error.<br>- Fix all defects found.<br>- Record defects in the Defect Recording log.<br>- Record time in the Time Recording log.<br>- Complete a Test Report template on the tests conducted and the results obtained. |

| | |
|---|---|
| **Exit Criteria** | - A thoroughly tested program that conforms to the Coding standard<br>- A formal specification conforming to the Formal Specification Standard<br>- Completed Design and Formal Specification templates<br>- Completed Design Review, Pseudo Code Review, Formal Specification Review and Code Review checklists<br>- Completed Test Report template<br>- Completed Time and Defect Recording logs |

*Table 14: Formal Specification Standard Template*

| | |
|---|---|
| Purpose | To guide the formal specification of programs |
| Program Headers | Begin all programs with a descriptive header. The header should use the Java documentation commenting convention ("/**") so automated documentation generation is possible. Include in the descriptive header the name of the author who writes the formal specification and a version number. |
| Header Format | /**<br>* @formal specification author Philip Johnson<br>* @formal specification version Tue Dec 26 2011<br>*/ |
| Identifiers | Use descriptive names for all variables, constants, and other identifiers. Avoid abbreviations or single letter variables. |
| Identifier Example | //@ public constraint age >= \old(age); //this is good<br>//@ public constraint i >= \old(i); //this is bad |
| Comments | Document the code so that the reader can understand its operation. Comments should explain both the purpose and behavior of the code. Comment variable declarations to indicate their purpose. |
| Good Comment | /*@ requires array != null;<br>@ ensures (* return the sum of the array elements *)<br>@ && \result == (\sum int I; 0 <= I && I < array.length; array[I]);<br>@ ensures (* without modifying the array *)<br>@ && (\forall int I; 0 <= I && I < array.length;<br>@ array[I] == \old(array[I]));<br>@*/ |

| | |
|---|---|
| Bad Comment | This comment is wrong:<br>/*@<br>@ ( * comment * ) assertion<br>@*/<br><br>This comment is OK:<br>/*@<br>@ ( * comment * ) && assertion<br>@*/<br>Comments are treated as assertions; therefore, they should be connected to other assertions by means of &&. |
| Indenting | Indent every level of brace from the previous one. |
| Indenting Example | /*@ public normal_behavior<br>@ requires divisor > 0;<br>@ ensures divisor*\result <= dividend<br>@ && divisor*(\result+1) > dividend;<br>@<br>@ also<br>@ public normal_behavior<br>@ requires divisor == 0;<br>@ ensures \result == 0;<br>@*/ |
| Capitalization | • Always use lower case in variable declarations.<br>• Use upper case for types and classes.<br>• Use upper case in invocations of a method so declared or of a JML library. |
| Capitalization Example | /*@ public model String name;<br>@ public represents name <- getName();<br>@<br>@ public invariant !"".equals(name);<br>*/ |

*Table 15: Specification Review Script*

| | |
|---|---|
| **Purpose** | To guide you in reviewing detailed designs |
| **Entry Criteria** | - Specification Review checklist<br>- Defect Type standard<br>- Time and Defect Recording logs |
| **General** | Where the Specification was previously verified, check that the analyses covered all of the Specification, were updated for all Specification changes, and are clear and complete. |

| **Step** | **Activities** | **Description** |
|---|---|---|
| 1 | Preparation | Examine the program and checklist and decide on a review strategy. |

| | | |
|---|---|---|
| 2 | Review | - Follow the Specification Review checklist.<br>- Review the entire program for each checklist category; do not try to review for more than one category at a time!<br>- Check off each item as you complete it.<br>- Complete a separate checklist for each product or product segment reviewed. |
| 3 | Fix Check | - Check each defect fix for correctness.<br>- Re-review all changes.<br>- Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space. |

| | |
|---|---|
| **Exit Criteria** | - A fully reviewed detailed Specification<br>- One or more Specification Review checklists for every design reviewed<br>- Documented Specification analysis results<br>- All identified defects fixed and all fixes checked<br>- Completed Time and Defect Recording logs |

*Table 16: Formal Specification Review Checklist Template*

| | | | |
|---|---|---|---|
| Student | _____ | Date | _____ |
| Program | _____ | Program # | _____ |
| Instructor | _____ | Language | _____ |
| Formal Specification Lenguage | _____ | | _____ |

| | |
|---|---|
| Purpose | To guide you in conducting an effective specification review |
| General | Review the entire Specification for each checklist category; do not attempt to review for more than one category at a time!<br>As you complete each review step, check off that item in the box at the right.<br>Complete the checklist for one specification or specification unit before reviewing the next. |
| General | To verify that the formal specification adequately complements the design. |

| | | | | | |
|---|---|---|---|---|---|
| Assertions | Assertions are prefixed by //@ or appear between /*@ … @*/<br>Every assert clause must end in ;.<br>Verify that the variable associated to each clause \forall, \sum, \exists, etc. is appropriately initialized.<br>In each clause \forall, \sum, \exists, etc. verify balance of parentheses in IF, ELSE, FOR, WHILE.<br>In each clause \forall, \sum, \exists, etc. verify that the appropriate segment of the array is traversed.<br>Verify that every method invoked within an assertion is declared as /*@ pure @*/. | | | | |
| Preconditions | Method preconditions are declared by means of the requires clause. | | | | |
| Postconditions | Method post conditions are declared by means of the ensures clause. | | | | |
| Class Invariants | Class invariants are declared by means of the invariant clause. | | | | |

# References/Bibliography

URLs are valid as of the publication date of this document.


**[Abrial 2006]**
Abrial, Jean-Raymond. "Formal Methods in Industry: Achievements, Problems, Future," 761-768. ICSE '06: Proceedings of the 28th International Conference on Software Engineering. Shanghai, China, May 2006. *www.irisa.fr/lande/lande/icse-proceedings/icse/p761.pdf.*

**[Babar 2005]**
Babar, Abdul and Potter, John. "Adapting the Personal Software Process (PSP) to Formal Methods," 192-201. Proceedings of the Australian Software Engineering Conference (ASWEC'05). Brisbane, Australia, Mar./Apr. 2005. IEEE Computer Society Press, 2005.

**[Barnett 2004]**
Barnett, Mike, Rustan, K., Leino, M., and Schulte, Wolfram. "The Spec# Programming System: An Overview," 49-69. Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Marseille, France, Mar. 2004. Springer-Verlag, 2004.

**[Cok 2005]**
Cok, David and Kiniry, Joseph. "ESC/Java2: Uniting ESC/Java and JML." Lecture Notes in Computer Science 3362 (2005): 108-128.

**[Crocker 2003]**
Crocker, David. "Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement," Lecture Notes in Computer Science 3582 (2003).

**[Eiffel 2012]**
Definition of Modern Eiffel.SourceForge, 2013.
http://tecomp.sourceforge.net/index.php?file=doc/papers/lang/modern_eiffel.txt.

**[Frege 1967]**
Frege, G. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle a. S.: Louis Nebert, 1879. Translated as Concept Script, a Formal Language of Pure Thought Modelled Upon That of Arithmetic, by S. Bauer-Mengelberg in J. vanHeijenoort (ed.), From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931. Harvard University Press, 1967.

**[Galeotti 2010]**
Galeotti, Juan, Rosner, Nicolás, Pombo, López, and Frias, Marcelo F. "Analysis of Invariants for Efficient Bounded Verification," 25-36. Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, Jul. 2010. ACM, 2010.

**[Gilb 1993]**
Gilb, Tom and Graham, Dorothy. Software Inspection. Addison-Wesley, 1994 (ISBN 978-0201-631814).

**[Hayes 1997]**
Hayes, William; & Over, James. Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers (CMU/SEI-97-TR-001). Software Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/library/abstracts/reports/97tr001.cfm

**[Hoare 1969]**
Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," Communications of the ACM 12, 10 (1969): 576-580.

**[Humphrey 2005]**
Humphrey, Watts S. PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley, 2005.

**[Humphrey 2006]**
Humphrey, Watts S. TSP: Coaching Development Teams. Addison-Wesley, 2006.

**[Kusakabe 2012]**
Kusakabe, Shigeru; Omori, Yoichi; and Araki, Keijiro. "A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report," 67-75. TSP Symposium 2012 Proceedings (CMU/SEI-2012-SR-015). Software Engineering Institute, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/library/abstracts/reports/12sr015.cfm

**[Meyer 1992]**
Meyer, Bertrand. "Applying Design by Contract," IEEE Computer 25, 10 (October 1992): 40-51.

**[Schwalbe 2007]**
Schwalbe, Kathy. Information Technology Project Management, 5th edition. Course Technology, 2007 (978-1423901457).

**[Suzumori 2003]**
Suzumori, Hisayuki, Kaiya, Haruhiko, and Kaijiri, Kenji. "VDM Over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development." Proceedings of the 27th Annual International Computer Software and Applications Conference. Dallas, Texas, Sept. 2003. IEEE Computer Society, 2003.

**[Rombach 2007]**
Rombach, D., Münch, D., Ocampo, A., Watts, H., and Burton, D. "Teaching Disciplined Software Development." Science Direct – The Journal of Systems and Software 81 (2007): 747 – 763.

**[Vallespir 2011]**
*Vallespir, Diego and Nichols, William. "Analysis of Design Defects Injection and Removal in PSP," 19-25. Proceedings of the TSP Symposium 2011: A Dedication to Excellence. Atlanta, GA, Sept. 2011*.

**[Vallespir 2012]**
*Vallespir, Diego and Nichols, William. "An Analysis of Code Defect Injection and Removal in PSP," 3-20. TSP Symposium 2012 Proceedings (CMU/SEI-2012-SR-015). Software Engineering Institute, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/library/abstracts/reports/12sr015.cfm*

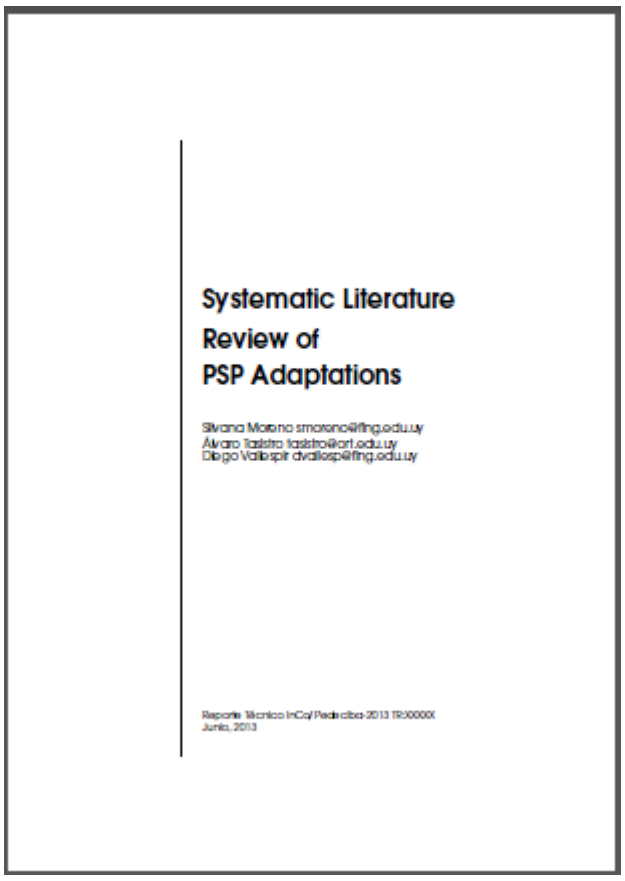
**[Wohlin 2000]**
Wohlin, C., Runeson, P, Höst, M., Ohlsson, M. C., Regenell, B. and Wesslén, A. Experimentation in Software Engineering. Kluwer Academic Publishers, 2000.

# Capítulo 3.

# Systematic Review of PSP Adaptations

In this chapter we present a systematic review of existing literature about PSP adaptations. In particular, we are interested in getting to know those adaptations that propose to incorporate the use of Formal Methods. Section 3.1 presents general concepts about systematic reviews. Section 3.2 presents the specific systematic review carried out.

Silvana Moreno, Álvaro Tasistro, Diego Vallespir



# 3.1 Systematic Reviews

A systematic review of the literature is a means of identification, evaluation and interpretation of all available information about a research question, subject area or phenomenon of interest [Kitchenham 07].

Several reasons may justify the realization of a systematic review. It may be necessary to summarize the existing investigations on a certain subject or technology in order to get to know the latter´s benefits and limitations. It could also be directed towards identifying shortcomings of current research, so as to suggest further investigations of to provide a frame wherein to position new research activities.

Generally, research projects begin with a literature review of some kind. However, this is often not carried out in an exhaustive way, which impoverishes its scientific value. This is the main reason for carrying out systematic reviews.

There are normally three phases to a systematic review: planning, conduction and report. We follow guides proposed in [Kitchenham 07].

## 3.1.1 Planning

During the planning phase the reasons for carrying out the review are identified, the research questions are specified, and the review protocol is defined and evaluated.

To state clearly the reasons for performing the review fulfills the important end of confirming its actual need.

The research questions determine the goals of the review. It is convenient that these are formulated in terms of one or several questions to be answered during the systematic review.

Finally, the protocol of the review specifies the methods to be used during its realization. It is necessary to count on a predefined protocol in order to reduce the possibility of bias on part of the researcher. For instance, in absence of a protocol, the selection of the case studies or their analysis could be influenced by the researcher's expectations. In general, the review protocols are submitted to pair evaluation.

The components of a protocol are:

- The reasons for performing the review
- The research questions
- The strategy for the search of articles
- The quality controls
- The strategy for data extraction and synthesis
- The publication strategy

We now clarify the last four points above.

The strategy for the search of articles is directed towards generating an adequate search chain and selecting the resources wherein to conduct the search. The search chain is generated by combining in several ways search terms that are derived from the research questions. One general approach consists in decomposing the questions into parts, adding synonyms, abbreviations or alternative expressions for each part, and finally connect the parts using AND, OR.

The resources wherein the search chain is normally applied are digital libraries, as well as repositories of specific journals or conference proceedings.

The selection criteria are properties required for an article to be included into the systematic review. The procedures of selection describe how the selection criteria are applied. This is necessary, for instance, to determine how each article is to be evaluated, and disagreements resolved, in the presence of several reviewers.

Quality control is normally performed using check lists. These contain the features that the researchers must observe on each article in order to ascertain its quality. In article selection, each feature is ranked, and the sum total of these ranks must overstep a predefined minimal value for the article to be included into the review.

The strategy of data extraction determines how to obtain the required information from each study. The strategy of data synthesis determines how the information obtained is to be synthesized.

Finally, the publication strategy determines the ways in which the systematic review is to be made generally available, i.e. through journal, brochure, poster, web page, report, etc.

Once the protocol is set up, it must be evaluated. The way this is done depends largely on the available budget. Students will generally submit their protocols to their supervisors' judgement. The assessment of the protocol must confirm that the search chain is adequately derived from the research questions and that the procedure of data analysis is suitable for answering such questions.

### 3.1.2 Conduction

In the phase of conduction the articles are selected, their quality is secured, and data extraction and synthesis are carried out according to the protocol.

### 3.1.3 Report

The final phase of a systematic review consists in writing down their results and making them available generally or to the stakeholders.

## 3.2 Systematic Review of Adaptations of the Personal Software Process.

In this section we present a systematic review of existing adaptations of the PSP. The division into subsections corresponds to the manner of presenting systematic reviews introduced by Kitchenham [Kitchenham 07].

### 3.2.1 Reason for the review.

The production of software has become a process focused on quality. PSP is a disciplined and individual process directed towards producing quality software. One of its principles consists in finding and correcting software defects at early stages of the development. Our proposal, $PSP_{VDC}$ is an adaptation of the PSP that incorporates the use of Formal Methods. The objective of this proposal is to improve quality by reducing the number of defects that arrive at the stage of Unit Testing.

The goal of the present review is to know whether there exist other proposals of adaptation of the PSP that aim at improving the quality of the software produced. In particular, we are interested in adaptations that incorporate the use of Formal Methods.

### 3.2.2 Research questions

The research questions are the following:

1. Do there exist proposals of adaptations to the PSP?
2. Do they incorporate use of Formal Methods?

### 3.2.3 Search strategy

The digital libraries employed in the search have been: SCOPUS, Springer, IEEE Xplore and EBSCO. These search engines comprise the main collections of journals and conference proceedings in the area of Software Engineering.

Besides, a manual search was conducted on "A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)", as well as on the collection of proceedings of the TSP Symposium carried out between 2006 and 2012. The first work cited above is a document edited by the SEI that contains references to books, chapters, sections and other types of publications concerned with the PSP and the TSP.

In order to encompass a larger number of articles, we also performed a crossed search. That is, we searched for articles that cited any of those articles found by means of our initial search strategy. In this second search we employed the same libraries as in the original search.

The search chain employed consists of three parts. The first part is related to the manners of referring to the PSP. The second part considers synonyms of "adaptation", i.e. different manners of referring to changes of the PSP. Finally, the third part refers to "use of Formal Methods". The first part is thus mandatory, whereas the second and third parts are both admissible. Therefore we arrive at the following:

(PSP **or** "personal software process") **and**

((adapting **or** extending **or** over **or** incorporating) **or** ("formal methods" **or** "design by contract"))

In applying the search chain on the search engines, we discovered that the acronym PSP is used within a large variety of subjects, which makes the search return very many irrelevant results. Therefore we decided to eliminate the option "PSP", on the basis of the consideration that "personal software process" is likely to appear either in the title or in the abstract of the articles we aimed at.

Therefore the search chain finally used is the following:

("personal software process") **and** ((adapting **or** extending **or** over **or** incorporating) **or**

("formal methods" **or** "design by contract"))

### 3.2.4 Criteria of inclusion and exclusion.

The author of the present review (and of the present thesis) evaluated each of the articles retrieved by both the automatic and manual search. In evaluating an article, we considered its title, keywords and abstract.

Those articles for which at least one of the following conditions is verified are to be excluded:

- The article does not focus on the Personal Software Process.
- It is a book or book chapter.
- It is a duplicate of one coming from a different source.
- It is not written in English.

### 3.2.5 Quality Control

We expect to find a very low number of articles satisfying the criteria of inclusion, and so we decide not to perform a quality check on them. Nevertheless, in a hypothetical quality check list we should include:

- That the adaptations proposed are presented in a clear and complete manner.
- That the article is published in a pertinent journal or conference.
- That the article has been cited.

### 3.2.6 Data Extraction and Synthesis

The data to be extracted from the selected articles are the following:

- Title
- Kind of publication (journal, magazine, conference, workshop)
- Year of publication
- Abstract
- Results/Conclusions

The data synthesis will consist in classifying the articles on the basis of our research questions, i.e. in: articles adapting PSP by incorporating Formal Methods, articles adapting PSP in any other respect, and articles using Formal Methods together with PSP without making any adaptation of the latter. The results will be studied as related work of the present thesis.

### 3.2.7 Results

During the conduct of the systematic review we identified studies. We use the search string on the selected search engines on 07/03/2013.

On the IEEE repository, after eliminating contents of types Books & eBooks, 31 results are obtained. On EBSCO results are filtered so as to avoid those contained in CAB Abstracts 1990-Present, Dentistry & Oral Sciences Source, MEDLINE and Ovid Journals. Then 34 results are obtained. Using SCOPUS the items belonging to the areas Life Sciences and Health Sciences are excluded, getting 20 results. Finally, on the Springer repository, by including only Computer Science and English articles, 20 results are obtained.

In "A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)" one article is found that proposes to adapt PSP incorporating the use of Formal Methods. This article was also found on EBSCO and IEEE.

In the TSP Symposium one article was found that did not appear in the searches conducted on the other sources, and an oral presentation.

Eliminating duplicates, the overall results of the primary search are as follows:

- EBSCO 34 articles
- Springer 19 articles
- Scopus 18 articles
- IEEE 29 articles
- TSP Symposium 1 article, 1 oral presentation

After applying the criteria of inclusion and exclusion, there remained 2 articles from the Scopus base, 1 from IEEE, and 1 article and 1 oral presentation from TSP Symposium.

Next, the data corresponding to the 4 articles found are presented. They all propose adaptations to the PSP; three of them propose adaptations that incorporate the use of formal methods. We also present the information related to the oral presentation in TSP Symposium that proposes to incorporate to the PSP integration techniques based on models. We got in touch with the authors via email in order to obtain some additional written material on the proposal, but it was not available.

**Title**: Integrating pair programming into a software development process

**Authors**: Williams, L.

**Type of article**: Conference

**Year**: 2001

**Abstract:** Anecdotal and statistical evidence indicates that pair programmers - two programmers working side-by-side at one computer, collaborating on the same design, algorithm, code or test - outperform individual programmers. One of the programmers, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects, and also thinks strategically about the direction of the work. On demand, the two programmers can brainstorm any challenging problem. Because the two programmers periodically switch roles, they work together as equals to develop software. This practice of pair programming can be integrated into any software development process. As an example, this paper describes the changes that were made to the Personal Software Process (PSP) to leverage the power of two programmers working together, thereby formulating the Collaborative Software Process (CSP). The paper also discusses the expected results of incorporating pair programming into a software development process in which traditional, individual programming is currently used [Williams 01] .

**Conclusions**: They described the changes made to the PSP to yield the CSP. These changes involved: 1) updating process scripts to document the role of the driver and the observer; 2) adapting data collection forms and analysis reports and 3) altering design and code review procedures. Making these explicit changes to the process cause several implicit, but beneficial, changes to the development environment.

**Title**: A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report

**Authors**: Kusakabe, S., Omori, Y. and Araki K.

**Type of article:** Symposium TSP

**Year:** 2012

**Abstract:** Software process is important for producing high-quality software and for its effective and efficient development. The Personal Software Process (PSP) provides a method for learning a concept of personal software process and for realizing an effective and efficient process by measuring, controlling, managing, and improving the way we develop software. PSP also serves as a vehicle to integrate advanced software engineering techniques, including formal methods, into one's own software development process. While formal methods are useful in reducing defects injected into a system, by mathematically describing and reasoning about the system, engineers may have difficulties integrating formal methods into their own software development processes. We propose an approach in which engineers use PSP to introduce formal methods into their software processes. As our initial trial, we followed one of our graduate students as he tried to improve his personal process with this approach. He measured and analyzed his own process data from PSP for Engineers-I, and proposed and experimented with an improved software process with a formal method, the Vienna Development Method (VDM). The experimental results indicate he could effectively reduce defects by using VDM [Kusakabe 12].

**Conclusions:** Kusakabe, Omori, and Araki proposed an approach in which developers use PSP as a framework of software process improvement to introduce formal methods into their software process for realizing effective and efficient development of high-quality software.

They reported one initial trial of the introduction of formal methods into personal process based on PSP. According to the process data in the trial, the developer spent more time in Design and less time in Test. He successfully reduced the number of defects he had focused on without decreasing his productivity.

**Title**: Adapting the Personal Software Process (PSP) to formal methods.

**Authors**: Babar, A., Potter, J.

**Type of article:** Conference

**Year:** 2005

**Abstract:** The goal of good software engineering practice is to deliver reliable, high-quality software on-time and on-budget. In this paper we advocate the combination of two modern approaches towards achieving this goal. On the one hand, with an eye to software quality, we consider adopting a state-based formal development method, the B-method. In terms of tool support and industry adoption, this is the most advanced such method. On the other, aimed at improving the development practices of individual developers, we consider the adoption of the Personal Software Process (PSP). To our knowledge this combination of formal methods and PSP has not been considered before; we term our special version of the combination B-PSP. We present a re-design of the PSP data collection and analysis tasks specifically geared towards the B-Method. Although we support the general framework of PSP, we also believe that developers do not enjoy having their creative and thinking process being interrupted by the need to regularly log activities. With this in mind, we present the PSP tasks in a style which should be acceptable to B developers. We view the results of this paper as a specification for some of the data logging and analysis requirements of a B-PSP-based development [Babar 05].

**Conclusions:** Babar and Potter combine Abrial's B Method with PSP into B-PSP. They add the phases of Specification, Auto Prover, Animation, and Proof. A new set of defect types is added and logs are modified so as to incorporate data extracted from the B machine's structure. The goal of this work is to provide the individual B developers with a paradigm of measurement and evaluation that promotes reflection on the practice of the B method, inculcating the habit of recognizing causes of defects injected so as to be able to avoid these in the future. We have had no notice about further results of this research. In comparison to B, our chosen formal method is significantly lighter and so, we expect, easier to incorporate into actual industrial practice.

**Title**: VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development

**Authors:** Suzumori, H., Kaiya, H., Kaijiri, K.

**Type of article:** Conference

**Year:** 2003

**Abstract:** Although formal methods seem to be useful, these is no clear way for beginners to know whether the methods are suited for them and for their problem domain, before using the methods in practice. We propose a method to confirm the suitability of a formal method. The method is realized as a pilot course based on the PSP. A course mentioned in this paper is designed for a typical formal method, VDM. Our course also helps beginners of VDM to learn VDM gradually and naturally. During the course, they can confirm its suitability as follows; First, they practice several exercises for software development, while techniques of VDM are introduced gradually. Second, process data and product data of software development

are recorded in each exercise. Third, by evaluating these data by several metrics, they can confirm the suitability of VDM for their work [Suzumori 03].

**Conclusions:** They propose the combination of VDM and PSP. The Design phase is modified incorporating the formal specification in the VDM-SL language. Besides, the phases of VDM-SL Review, Syntax Check, Type Check and Validation are added. A prototype course is proposed in which each student is to carry out nine exercises applying VDM on the PSP. Data thereby collected shows that about 90% of the defects are eliminated before the Code Review phase. A conclusion is then that the use of VDM contributes to eliminate defect injection during design. After this work was concluded, the research was discontinued for reasons internal to the organization.

**Title**: Integrating Model-Driven Engineering Techniques in the Personal Software Process

**Authors:** Pascoal, J.

**Type of article:** Symposium TSP

**Year:** 2012

**Abstract:** The authors propose an approach based on MDE (Model-Driven Engineering) for generating code from models with the objective of checking the quality of the models. The approach consists in developing structural models MDD (for instance, class skeletons) and developing models of partial behavior MBT that are sufficient for generating tests. The PSP is modified to incorporate the construction of the models and the generation of model based tests [Pascoal 12].

**Conclusions:** The authors conclude that theirs is a "PSP friendly" proposal, which promotes the realization of precise and easily revisable designs at a low cost in terms of script modifications. It is designed to bring short term productivity and quality benefits

Finally, we performed the crossed search on EBSCO, Springer, Scopus e IEEE, for the articles citing any of the 4 found in the original search, which did not add any article to the search results. Using Scopus we found that the articles incorporating the formal method B and VDM into PSP are not mentioned in any other article. Using Springer and EBSCO those two articles are not found. Finally using IEEE we found that the VDM article is quoted in the one using the B formal method in PSP, whereas the latter is not mentioned in any article.

The article "A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report" TSP Symposium, is not found by any sources Springer / EBSCO / IEEE / Scopus. Therefore is not possible to know if it was cited. Finally, using Scopus and Springer we found that the article incorporating pair programming to PSP is cited by the articles "Contemporary peer review in action: Lessons from open source development" and "A Repository of Agile Method Fragments" respectively. However these articles do not focus on PSP. Using IEEE we found that the article is cited by other two articles: "A multiple case study on the impact of pair programming on product quality" and "Heterogeneous and homogenous pairs in pair programming: an empirical analysis", which also focus on the PSP. When searching on EBSCO the article that incorporating pair programming to PSP is not found.

## 3.2.8 Discussion

We summarize the main conclusions arising from the systematic review. The goal of the review was getting to know the adaptations to the PSP that have been proposed and, in particular, finding out whether any of them proposed to incorporate the use of formal methods.

One of the works found proposes to modify the PSP to carry out pair programming in the software development process. The new process is called Collaborative Software Process (CSP). The authors explain how the scripts, templates and forms of the PSP are adjusted to incorporate pair programming. In particular, they describe the modifications that are required to distinguish the tasks corresponding to each role (developer and observer) and the times at which role switch has to be accomplished. Another modification is the use of two design check lists and two code check lists, one for each role.

The oral presentation at the TSP Symposium proposes an adaptation to integrating Model-Driven Engineering Techniques in the PSP. The proposed process modifies the design phase for including the development of a design model that describes the external structure of the system and its behavior. This model is to be subsequently refined for describing the internal structure of the system and its behavior. The phase of Design Review incorporates the checking of the model by using a static analysis tool. Finally, the phases of Code and Unit Test incorporate partial generation of code from the models as well as model based test generation.

The other articles propose adaptations to the PSP that incorporate the use of Formal methods. Babar and Potter propose a new process called B-PSP that incorporates the B formal method. The proposal by Suzumori, Kaiya y Kaijiri, as well as that by Kusakabe, Omori and Araki, propose the use of VDM within the PSP.

All the proposals found modify scripts, templates and forms in order to give support to the new processes. CSP maintains the same development phases of the PSP, incorporating to them the roles of observer and developer. The proposal that integrating model-driven engineering techniques also maintains the same phases of the PSP, modifying several of them to incorporate code and test generation as well as model checking.

The proposal VDM over PSP (VDM-PSP) both adds and modifies phases of the development process. This is the same as in our proposal, $PSP_{VDC}$. VDM-PSP modifies the Design phase to incorporate formal specification in the VDM-SL language. After formal specification, the phase of Design Review of the PSP is carried out. After that, new phases are to be performed: syntax review, syntax check, type check and validation. During syntax review, the user checks the specifications for syntax defects. Besides, a Tool box automatically carries out syntax review, type checking and validation.

In this proposal they do not apply proof techniques because VDM-PSP is intended to be carried out by inexperienced students.

As different from VDM-PSP, in $PSP_{VDC}$ we decided to add a new phase aimed at formal specification. This makes it possible to obtain information of time spent, defects injected and removed only as a consequence of formal specification.

The VDM-SL syntax review phase in VDM-PSP is similar to that of formal specification review in $PSP_{VDC}$. Both have as goal to remove the defects injected during the formal specification.

The syntax review and type checking phases, as performed automatically by a tool, could be considered similar to the formal specification compile phase of $PSP_{VDC}$. The validation phase is not clearly presented by the authors. We understand that in this phase the specification is checked with respect to the user´s requirements, using a convenient support tool. This phase can be considered similar to the review of the formal specification of $PSP_{VDC}$.

$PSP_{VDC}$ also includes Test case construct and Pseudo code phases which the VDM proposal does not specify.

Figure 3.1 shows the phases of each process. The colors and arrows indicate how $PSP_{VDC}$ phases can be mapped on those of VDM over PSP.

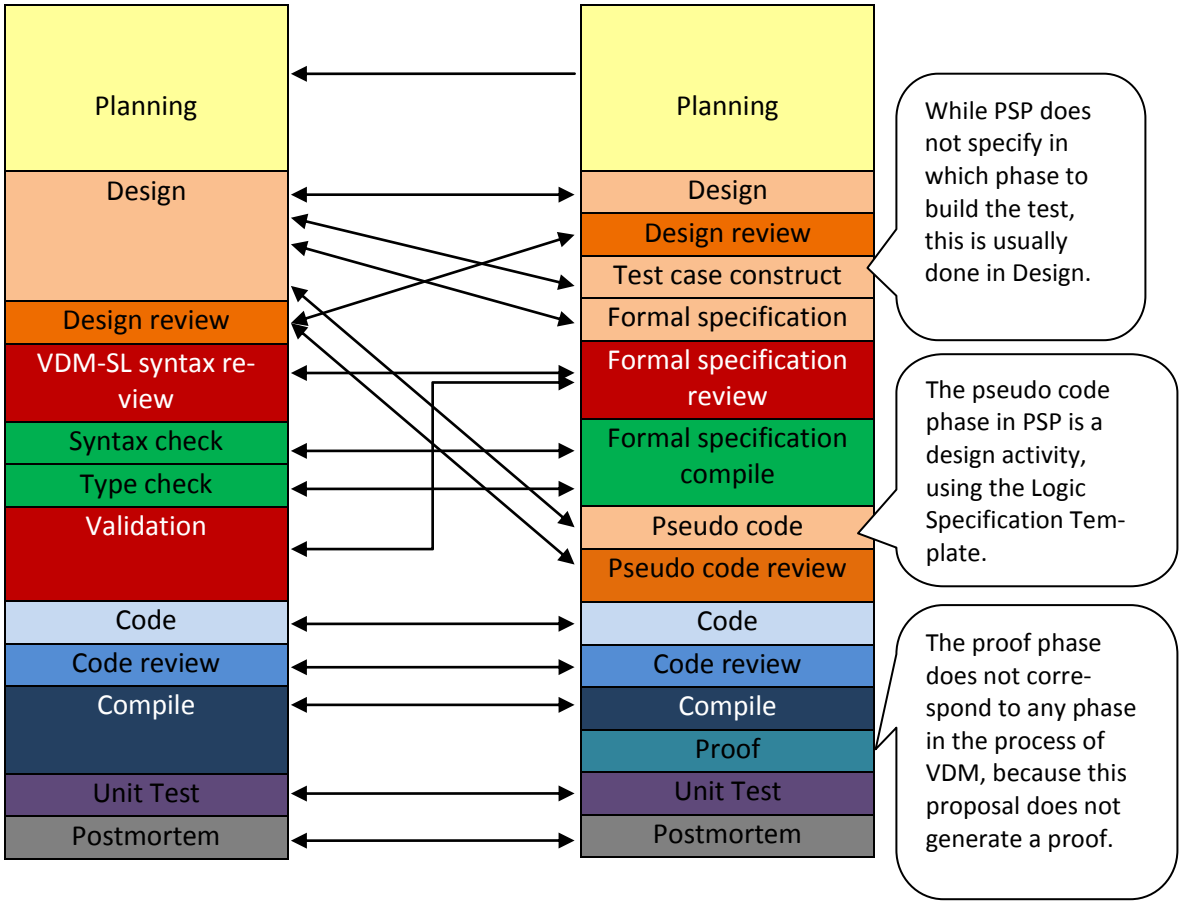



*Figure 3.1: Phases of proposals $PSP_{VDC}$ and VDM over PSP.*

The proposal by Kusakabe, Omori and Araki incorporates the formal method VDM with its various formal specification languages and a toolkit that enables syntax checking, type checking, the use of an interpreter and the generation of test obligations.

The proposal maintains the same phases that PSP, modifying:

- the design phase to incorporate formal specification using any of the specification languages (eg VDM + +) and
- design review phase, to incorporate the use of the VDM toolkit.

As mentioned earlier, this differs from $PSP_{VDC}$, where we decided to perform the specification and the specification review in new phases. It is not clear in their proposal in which phase the proof obligations are generated. In $PSP_{VDC}$ this is done during the Proof phase. We believe their proposal is incomplete, which makes comparison difficult. We contacted the authors via email for more information but have got no response.

Figure 3.2 presents the phases of each process. Colors and arrows indicate how $PSP_{VDC}$ phases can be mapped onto those of the proposal by Kusakabe, Omori and Araki.

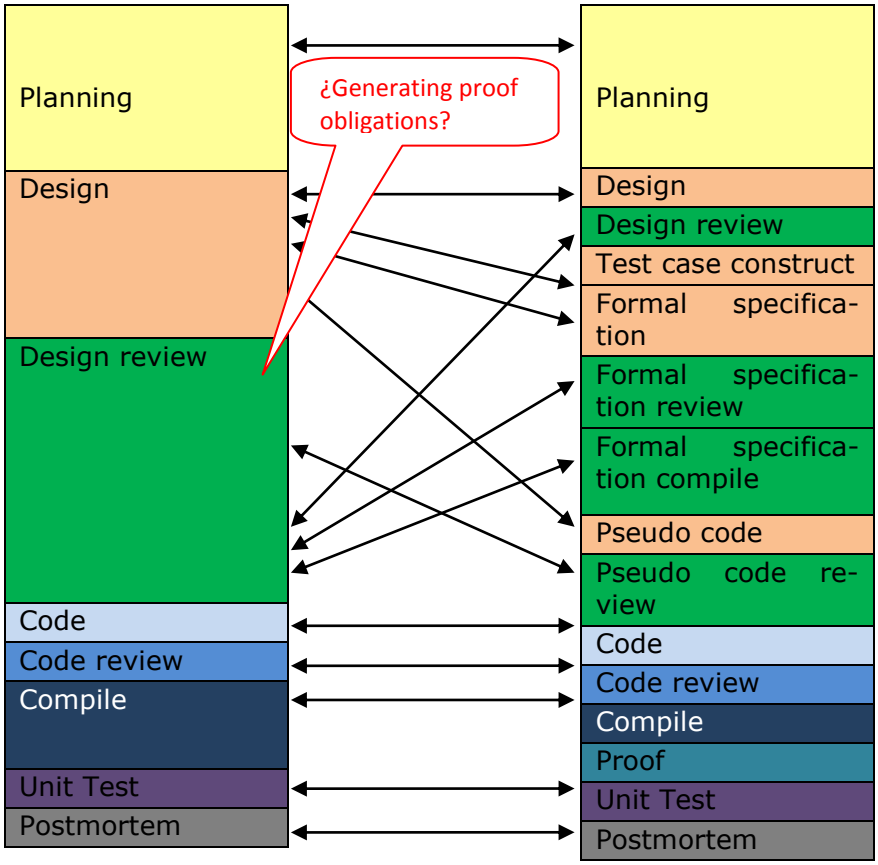



*Figure 3.2: Phases of proposals Kusakabe, Omori and Araki, and $PSP_{VDC.}$*

The B-PSP proposal incorporates Specification, Auto Prover, Animation and Auto Proof phases. During the specification the B-machines are specified using the B formal language. Later, Auto Prover and Animation phases are performed with the assistance of the B toolkit. The basic syntax checking and dependency between machines is controlled, generating proof obligations and providing animation. During the Auto Proof an interactive proof of correctness is carried out, using the B toolkit.

B-PSP also proposes generating code automatically, but does not explain how that phase is performed. The proposal eliminates the design, design review, code, code review, compile and unit test phases, but does not clarify whether the activities undertaken during these are made in some of the new phases.

When comparing B-PSP with $PSP_{VDC}$, we note that the specification activity is proposed in both cases as a new phase, allowing to collect information about cost and specific defects of that phase. We note that no specification review is performed in B-PSP. $PSP_{VDC}$, on the other hand, proposes a formal specification review that allows to detect early injected defects. The syntax control activity performed during the Auto Prover and the Animation phase in B-PSP are carried out in the formal specification review and formal specification compilation phases in $PSP_{VDC.}$

Finally, the Auto Proof phase in B-PSP is analogous to the $PSP_{VDC}$ Proof phase, seeking to generate the proof with the assistance of tools. The figure 3.3 illustrates the correspondence between the phases of both proposals.

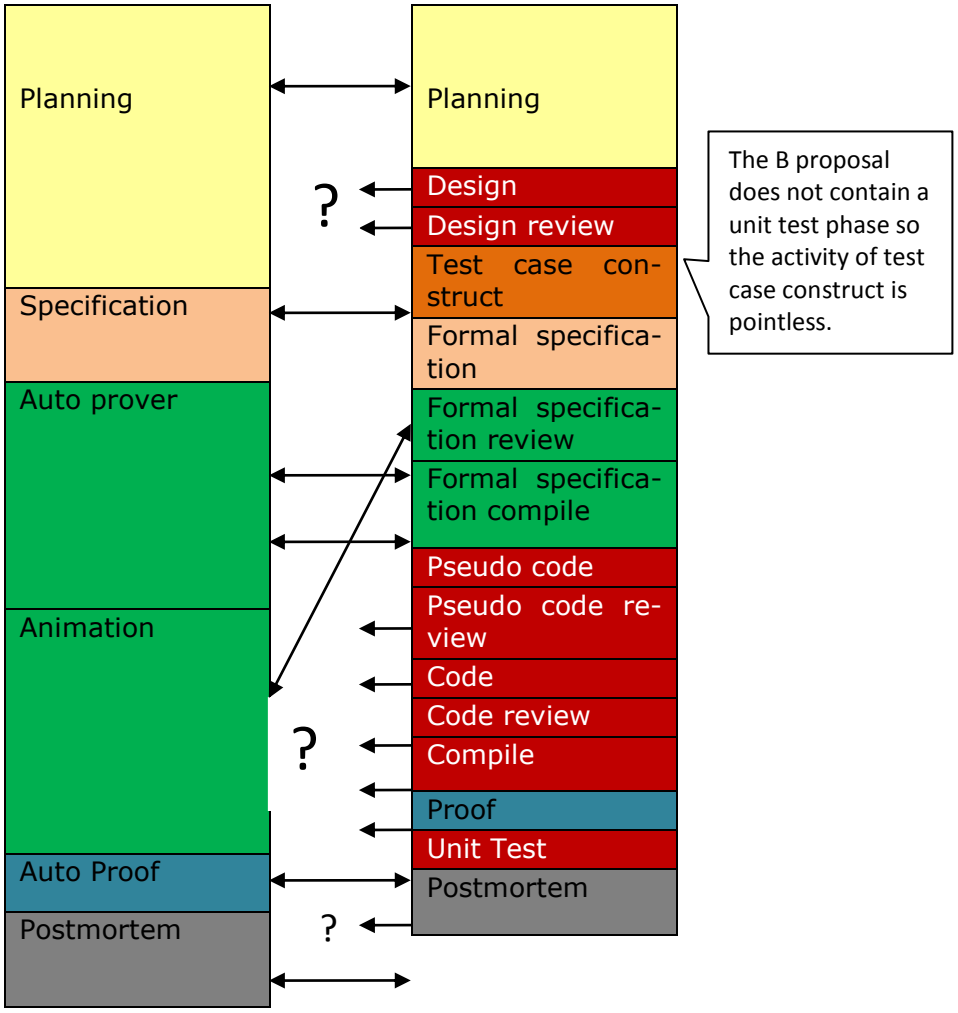



*Figure 3.3: Phases of proposals B-PSP and $PSP_{VDC}$.*

## 3.2.9 Conclusions

We present a systematic review of the literature that attempts to summarize the existing information on the adjustments made to the PSP and particularly those incorporating formal methods. Only 5 articles are found that propose adaptations of the PSP. 3 of them are adaptations of the PSP for the use of formal methods; the other two adapt the PSP to the use of pair programming techniques and integrating model-driven engineering techniques respectively.

Unfortunately, the three papers that incorporate formal methods have not presented the full proposed process and do not refer to any technical report doing so to which we can access. Because of that, it is not possible to make a thorough comparison between these processes and $PSP_{VDC}$.

Nevertheless, we have compared the various phases of the corresponding processes. As detailed in the preceding section, some of the adaptations to the PSP bear similarities to our proposal, while others do not. Different adaptations allow different degrees of granularity of the data collected.

The two proposals that incorporate the VDM to the PSP are very different in the way they perform the adaptation. One adds several phases to the process to incorporate the specification and use of tools, while the other has the same phases of the original PSP, incorporating further activity to these. Our proposal $PSP_{VDC}$ is one

possible way of adapting PSP by incorporating design by contract. There may be several other proposals for the same purpose. To determine whether $PSP_{VDC}$ produces improvements in individual development process remains now for us to investigate.

# Capítulo 4.

# Evaluación del $PSP_{VDC}$ y Comparación con el PSP

En este capítulo se presenta una propuesta para evaluar el $PSP_{VDC}$ y compararlo con el PSP. El objetivo es analizar los procesos con el propósito de conocer la calidad de los productos y la productividad que se logra, en el contexto de un experimento controlado llevado a cabo por estudiantes de la Carrera de Ingeniería en Computación de la Facultad de Ingeniería - Universidad de la República.

## 4.1 Introducción

El PSP se basa en una serie de prácticas que no son comúnmente seguidas en los desarrollos de software [Humphrey 05, Macchi 12]. Ejemplos de tales prácticas son la revisión de código y la recolección de datos del proceso. Es por esto que Humphrey, con motivo de enseñar a los ingenieros el PSP, utiliza una forma de adopción a través de la cual la familiarización con el proceso se produce gradualmente. Para esto construyó una serie de niveles, cada uno de los cuales extiende al nivel anterior introduciendo nuevas prácticas y elementos (formularios, scripts, estándares, etc) del proceso. La enseñanza del PSP se realiza mediante cursos. Los cursos propuestos por Humphrey se denominan "*PSP for engineers 1*" y "*PSP for engineers 2*". Durante estos cursos, a medida que se realizan ejercicios de programación, se van introduciendo las prácticas del PSP (utilizando los distintos niveles).

Siguiendo la misma línea que el PSP, definimos 2 niveles para el $PSP_{VDC}$. El nivel 1 denominado $PSP_{VDC}$ *light* y el nivel 2 denominado $PSP_{VDC}$ *full*. En el $PSP_{VDC}$ *light* no se realiza la demostración formal de los programas mientras que en el $PSP_{VDC}$ *full* sí. Entonces, el nivel 1 no tiene la fase *proof* mientras que el nivel 2 sí. El $PSP_{VDC}$ full es el proceso completo, que fue presentado en el capítulo 2.

Estos niveles permiten armar un curso que introduzca gradualmente las prácticas del $PSP_{VDC}$, explicando entre otras cosas en el nivel 1 lo correspondiente a especificaciones formales y en el nivel 2 los conceptos de las pruebas. Además, los datos recabados de aplicar cada uno de los niveles del $PSP_{VDC}$ permiten realizar comparaciones entre los niveles. Por ejemplo, es de nuestro interés conocer si el introducir la prueba ($PSP_{VDC}$ *full*) mejora la calidad de los productos con respecto al $PSP_{VDC}$ *light*.

En este capítulo se presenta una propuesta para evaluar el $PSP_{VDC}$ (*light* y *full*) y compararlo con el PSP mediante la realización de experimentos controlados. Los experimentos controlados son una técnica de investigación en la cual se quiere tener control del estudio y del entorno en el que éste se lleva a cabo[4]. La propuesta planteada consiste en la realización de dos experimentos controlados. El objetivo de los experimentos es comparar la productividad de los procesos $PSP_{VDC}$ *light*, $PSP_{VDC}$ *full* y PSP. En el primer experimento se comparan el $PSP_{VDC}$ *light* y el PSP y en el segundo experimento se comparan el $PSP_{VDC}$ *full* y el PSP.

En un experimento controlado a menudo aparecen dificultades imprevistas, por lo que es conveniente intentar anticiparse a estos problemas. Una forma de anticiparse es realizando un caso piloto. Un caso piloto consiste en realizar las mismas actividades del experimento controlado pero con el objetivo de detectar problemas

[4] El Anexo A "Experimentos Formales" presenta los conceptos básicos de los experimentos controlados en ingeniería de software.

y corregirlos antes de comenzar el experimento. Por este motivo, proponemos realizar dos casos pilotos con el objetivo de evaluar el funcionamiento del $PSP_{VDC}$ *light* y del $PSP_{VDC}$ *full*. Denominamos piloto *light* y piloto *full* a los casos pilotos correspondientes a $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full* respectivamente.

El piloto *light* consiste en la aplicación del $PSP_{VDC}$ *light* por parte de un sujeto a los mismos ejercicios propuestos en el curso de "*PSP for engineers 1*". Previamente el sujeto deberá ser capacitado en el lenguaje de especificación formal y las herramientas a utilizar, así como en $PSP_{VDC}$ *light*, lo cual implica también una capacitación anterior en el PSP.

Una vez finalizado el caso piloto *light* y realizadas las mejoras pertinentes se realizará el experimento controlado que compara el PSP con el $PSP_{VDC}$ *light*. El objetivo de este experimento es conocer y comparar la calidad de los productos y productividad que se obtiene al aplicar ambos procesos. Definimos calidad como la densidad de defectos encontrados en la fase de *Unit Test* (cantidad de defectos/LOCS), y la productividad como la cantidad de líneas de código producidas por hora (LOCS/horas).

El experimento se llevará a cabo como parte de una materia de facultad, donde un grupo de estudiantes aplica PSP o $PSP_{VDC}$ *light* a un conjunto de ejercicios. La mitad de los estudiantes aplicará el PSP y la otra mitad el $PSP_{VDC}$ *light* sobre los mismos ejercicios. Los estudiantes deberán ser previamente entrenados en PSP y $PSP_{VDC}$ *light*. Los entrenamientos consisten en capacitar a los estudiantes en el PSP y en el $PSP_{VDC}$ *light* antes de la ejecución del experimento, denominados entrenamientos *light*. La figura 4.1 ilustra cronológicamente la propuesta experimental para comparar el PSP y el $PSP_{VDC}$ *light*.


*Figura 4.1: Cronología Piloto-Entrenamientos-Ejecución Experimento Light*

El caso piloto *full* consiste en la aplicación por parte del mismo sujeto que aplica el piloto *light* a los ejercicios propuestos en el curso de "*PSP for engineers 2*". En este caso el sujeto deberá ser capacitado con $PSP_{VDC}$ full, las herramientas que ayudan a construir la prueba y con técnicas de pruebas de programas (*proof verification*).

Una vez finalizado el piloto *full* se realizará un experimento formal $PSP_{VDC}$ *full* que tiene como objetivo comparar la calidad y productividad de PSP y $PSP_{VDC}$ *full*. El experimento $PSP_{VDC}$ *full* también se llevará cabo como parte de una materia de facultad, donde un grupo de estudiantes aplica PSP o $PSP_{VDC}$ *full* a un conjunto de ejercicios. La mitad de los estudiantes aplica PSP y la otra mitad $PSP_{VDC}$ *full* sobre los mismos ejercicios. El experimento $PSP_{VDC}$ *full* se pretende realizar un año después que el experimento $PSP_{VDC}$ *light* y con estudiantes diferentes. Los estudiantes deberán ser entrenados en PSP y $PSP_{VDC}$ *full*. Dicho entrenamiento se realiza en los denominados entrenamientos *full*. En la figura 4.2 se ilustra la propuesta *full* descrita anteriormente.


*Figura 4.2: Cronología Piloto-Entrenamientos-Ejecución Experimento Full*

## 4.2 Experimentación con $PSP_{VDC}$ *light*

En esta sección se presentan el caso piloto *light* y el experimento controlado *light*.

### 4.2.1 Caso piloto *light*

El piloto *light* consiste en la aplicación del $PSP_{VDC}$ *light* previamente al experimento con el objetivo principal de detectar posibles problemas en el uso del proceso. Los problemas encontrados se deben corregir antes de ejecutar el experimento controlado. Algunos aspectos del $PSP_{VDC}$ *light* que nos interesa observar durante la ejecución de piloto *light* son la herramienta de soporte al proceso y los ajustes realizados a los scripts. La herramienta consiste de diversos formularios de registro de datos y los scripts son la guía para seguir el proceso; ambos son adaptados para $PSP_{VDC}$ *light* y podrían estar incompletos, ambiguos, etc. Tener conocimiento sobre estos aspectos nos permitirá ajustar y mejorar el proceso. Los registros del tiempo insumido al aplicar el proceso nos permitirán obtener datos para armar el cronograma del experimento controlado.

El piloto *light* consiste en la aplicación del $PSP_{VDC}$ *light* por parte de un sujeto a un conjunto de ejercicios. Siguiendo el proceso experimental presentado en el Apéndice A, planificamos el piloto *light* en 4 etapas: definición, planificación, operación y análisis, tal como se muestra en la figura 4.3. Durante la fase de definición se define el problema a resolver, el propósito y los objetivos. Durante la planificación se selecciona el contexto, se formulan las hipótesis, se seleccionan los sujetos, se diseña el experimento y se planifica, de ser necesario, la capacitación a los sujetos, construcción de cursos/guías, descripción de procesos, planillas y herramientas. Durante la operación se realizan los entrenamientos necesarios y se ejecuta el experimento. En el análisis se examinan los resultados y se realizan los análisis estadísticos. Si bien el piloto light no es un experimento controlado utilizamos el proceso definido para presentar la propuesta de forma ordenada.

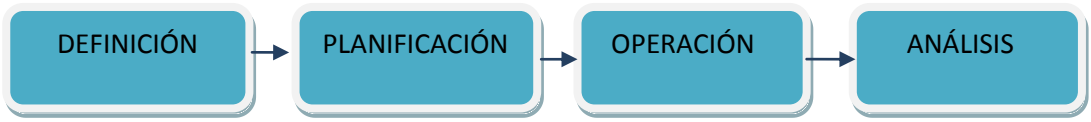



*Figura 4.3: Fases Experimento Controlado*

La operación y el análisis de piloto *light* están fuera del alcance de esta tesis de maestría. A continuación se presentan la Definición y la Planificación.

#### Definición y Planificación

El piloto *light* consiste en la aplicación de $PSP_{VDC}$ *light* por parte de un sujeto a los mismos ejercicios propuestos en el curso de "*PSP for engineers 1*". Se propone que el sujeto que participe del Piloto *light* sea el investigador principal del $PSP_{VDC}$. El motivo de esta selección es porque el investigador es quien conoce la propuesta $PSP_{VDC}$ *light* y será capaz de detectar más fácilmente posibles problemas o inconsistencias en ésta.

Java y JML [Leavens 06] serán respectivamente los lenguajes de programación y de especificación formal que se usarán durante el piloto y durante el experimento. Esto se debe a que el investigador que ejecutará el piloto conoce ambos lenguajes. Además, el lenguaje de programación Java continúa siendo muy utilizado en diversos cursos de grado en nuestra Facultad, lo que nos permitirá experimentar con estudiantes con conocimientos en Java. Para verificar las aserciones especificadas

en JML se utilizará un compilador de jml a seleccionar. Además para llevar a cabo el piloto *light* se requiere de una capacitación personal previa en el uso de lenguajes, herramientas a utilizar y el PSP. Planificamos repasar el PSP, Java y planteamos un conjunto de 4 ejercicios prácticos que pueden ser utilizados para la práctica de JML.

- Ejercicio 1: Construir una clase Java que cree un arreglo de 5 posiciones, lo cargue con los elementos [1, 2, 4, 5, 6] y despliegue su contenido. Especificar formalmente y compilar la especificación formal.

- Ejercicio 2: Construir una clase Java que cree dos arreglos de 10 posiciones, los cargue con valores de la entrada estándar y despliegue su contenido. Especificar formalmente y compilar la especificación formal.

- Ejercicio 3: Construir una clase Java que cree dos arreglos de 10 posiciones, los cargue con valores de la entrada estándar, realice el *merge* de los mismos y despliegue el resultado. Especificar formalmente y compilar la especificación formal.

- Ejercicio 4: Construir una clase Java que cree una matriz de 3x3 posiciones, la cargue con elementos de la entrada estándar y despliegue su contenido. Especificar formalmente y compilar la especificación formal.

Conocemos dos herramientas de soporte al proceso PSP que simplifican la recolección de los datos de tiempos y defectos, automatizan diversos cálculos, guardan datos históricos y realizan análisis de los datos entre otras cosas: "*PSP Student Workbook*" que fue creada por el SEI sobre Access y "*Process Dashboard*" que es *Open Source* hecha en Java. En este piloto se utilizará la herramienta "*PSP Student Workbook*"; debiendo ser adaptada al $PSP_{VDC}$ *light*. Esta herramienta fue utilizada por el investigador durante el curso de PSP, por lo que ya se conocen sus formularios y forma de utilización. La misma se deberá adaptar para considerar las nuevas fases que introduce el $PSP_{VDC}$ *light* en todos los formularios, cambios en estimación y registros de datos. Al finalizar cada ejercicio se deberá realizar una entrega a un tutor que revise el trabajo realizado para conocer si quien está ejecutando el piloto se ajusta al proceso y para obtener retroalimentación. El tutor, en lo posible, debe ser instructor autorizado del PSP por el SEI y conocer el $PSP_{VDC}$ *light*.

Durante la operación (preparación y ejecución) se realizan las actividades planificadas y posteriormente se aplicará el $PSP_{VDC}$ *light* a los 4 ejercicios del curso "*PSP for engineers 1*". Este curso fue tomado por el investigador a comienzos del 2011, durante el cual realizó los 4 ejercicios aplicando el PSP. Pasados al menos dos años y medio, consideramos que el utilizar los mismos ejercicios no es una amenaza a los objetivos del piloto *light* ya que el investigador no recuerda los problemas planteados en los ejercicios, y menos aún la solución implementada. Además el foco de este piloto *light* está puesto en detectar problemas en el uso del $PSP_{VDC}$ *light* y corregirlos y no en la programación o en los datos obtenidos de aplicar el proceso.

Durante el análisis se realiza una evaluación de la aplicación realizada y se realizan los ajustes a la herramienta y al proceso en caso necesario.

Resumiendo, las actividades a realizar durante la operación y análisis de piloto *light* son:

- Capacitación
  - Repasar Java
  - Capacitarse en JML
  - Seleccionar y capacitarse en el uso del compilador

    - Realizar utilizando JML los 4 ejercicios mencionados anteriormente para familiarizarse con el uso práctico del JML y sus herramientas
    - Repasar el PSP
- Adaptar la herramienta "*PSP Student Workbook*"
- Aplicar el $PSP_{VDC}$ light a los 4 ejercicios del curso "*PSP for engineer 1*"
- Análisis y mejoras al $PSP_{VDC}$ light y al "*PSP Student Workbook*"

La figura 4.4 ilustra las actividades a realizar durante el piloto *light*.

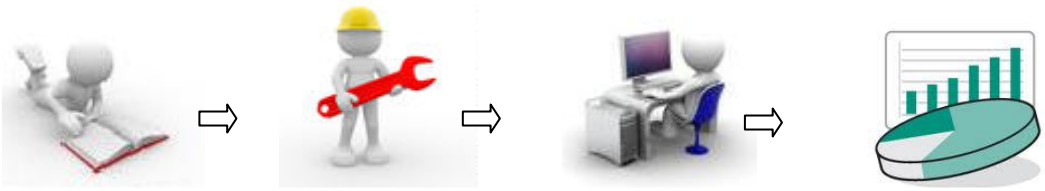



*Figura 4.4: Actividades a realizar durante el piloto light*

## 4.2.2 Un Experimento Controlado: PSP vs $PSP_{VDC}$ *light*

En esta sección presentamos la definición y planificación de un experimento controlado que busca comparar la calidad de los productos y la productividad que se obtiene al aplicar el $PSP_{VDC}$ *light* y el PSP. Definimos la calidad como la cantidad de defectos detectados en UT (*Unit Test*) y la productividad como la cantidad de líneas de código codificadas por hora (LOCS/horas).

El experimento *light* respeta el proceso experimental de la figura 4.3. Como ya mencionamos en el piloto *light*, durante la definición de objetivos se define el problema a resolver, el propósito y los objetivos. En la fase de planificación se selecciona el contexto, se formulan las hipótesis, se seleccionan los sujetos, se diseña el experimento y se planifican la capacitación a los sujetos, construcción de cursos/guías, descripción de procesos, planillas y herramientas. Durante la planificación del experimento light se planifica la preparación de los cursos de métodos formales y $PSP_{VDC}$ *light*; y se planifica el entrenamiento de los sujetos respecto a la aplicación de los procesos y al uso de las herramientas necesarias. El entrenamiento se divide en dos (denominados entrenamientos A y B *light*). Son dos entrenamientos ya que en A se entrenará a los sujetos en el PSP y en B se entrenará a la mitad de los sujetos en métodos formales y $PSP_{VDC}$ *light* mientras la otra mitad continúa la práctica en el PSP. En las secciones siguientes se presenta la definición, planificación y los entrenamientos A y B *light*.

Por último durante la operación se ejecutan los entrenamientos y demás actividades planificadas y posteriormente se ejecuta el diseño del experimento. En el análisis se examinan los resultados con el objetivo de evaluar el proceso. Queda por fuera de este trabajo de tesis la operación y análisis de resultados del experimento controlado. Sin embargo, al final del capítulo proponemos el diseño de la ejecución que se desea llevar adelante y cómo se pretenden analizar los resultados.

### Definición y Planificación

El objetivo del experimento es comparar la calidad de los productos y la productividad que se obtiene al aplicar el PSP y el $PSP_{VDC}$ *light*. El experimento se lle-

vará a cabo en el contexto de un curso en la Facultad de Ingeniería de la Universidad de la República. Los sujetos intervinientes son estudiantes de la Carrera de Ingeniería de Software.

Como es clásico en la ingeniería de software empírica, se presenta el objetivo del experimento utilizando el método Goal, Question, Metric (GQM) [Basili 94]:

**Analizar y comparar** los datos recolectados en 12 ejercicios de programación con

**el propósito** de evaluar las mejoras en la calidad de software y la productividad

**respecto** a la densidad de defectos en *Unit Test* y a la cantidad de Locs desarrolladas por hora desde el

**punto de vista** de un investigador en el

**contexto** de la aplicación de PSP y $PSP_{VDC}$ *light* en un grupo de entre 10 y 20 estudiantes.

El diseño del experimento es de un factor (proceso de desarrollo de software) con dos alternativas (PSP y $PSP_{VDC}$ *light*). La unidad experimental es el conjunto de 12 ejercicios que componen los cursos de el PSP y el $PSP_{VDC}$ *light*. Las variables de respuesta consideradas son la calidad de los productos y productividad de los procesos. La calidad se define como la densidad de defectos que llegan a UT (cantidad de defectos/LOCS) y la productividad como la cantidad de líneas de código codificadas por hora (LOCS/horas). [5]

Los sujetos participantes son estudiantes de la carrera Ingeniería en Computación de la Facultad de Ingeniería. Se buscan estudiantes de similares características, avanzados en la carrera, que se encuentren en cuarto o quinto año y que tengan aprobado el curso de Taller de Programación en el cual se aprende Java. Debido al esfuerzo de seguimiento tanto del curso como del experimento un número de entre 10 y 20 estudiantes será adecuado.

Para ejecutar el experimento es necesario capacitar a los sujetos en el PSP, el $PSP_{VDC}$ *light* y el uso de las herramientas de soporte. A la vez, para capacitar a los sujetos se requiere el armado de cursos de $PSP_{VDC}$ *light* y de métodos formales. Previo a los entrenamientos A y B *light* se debe realizar el armado de estos cursos. El curso sobre métodos formales, en particular sobre diseño por contrato deberá ser introductorio, abarcando conceptos de contratos, pre/pos condiciones, invariantes, el lenguaje JML y algunos ejemplos concretos. Se estima que el curso se dictará en clases teórico/prácticas con una duración total de 10 horas. El esfuerzo estimado para preparar este curso es de 30 horas.

El curso de $PSP_{VDC}$ *light* introducirá gradualmente las prácticas propuestas del $PSP_{VDC}$ light durante 4 días y se plantearán 4 ejercicios prácticos; uno por día. Definimos 3 sub niveles al $PSP_{VDC}$ *light* para introducir gradualmente los conceptos a lo largo del curso. El subnivel 1 introduce al PSP los conceptos de pseudo code y de construcción de casos de prueba, el subnivel 2 agrega al subnivel 1 los conceptos de especificación formal y compilación de la especificación formal, por último, el subnivel 3 incorpora al subnivel 2 la revisión del pseudo código y revisión de la especificación formal.

A continuación se presenta una descripción de los temas a tratar en cada uno de los 4 días de curso:

Día 1. Se introducen los conceptos de pseudo code y de construcción de casos de pruebas. Se explican cómo y porque estas actividades se realizan en fases distintas al PSP y se exponen los scripts correspondientes.

Día 2. Se introduce el concepto de especificación formal y compilación de la especificación formal. Se detallan estas nuevas fases junto con sus scripts.

Día 3. Se introduce el concepto de las revisiones (ya visto en PSP) y se explican las nuevas fases de revisión del pseudo código y revisión de la especifica-

[5] Los conceptos utilizados de Ingeniería de Software Empírica están descritos en el anexo A.

ción formal, así como los *scripts* y *check list*. En este día se aplica por primera vez de forma completa el $PSP_{VDC}$ light.

Día 4. Se aplica nuevamente el $PSP_{VDC}$ *light* completo al último ejercicio.

El curso de $PSP_{VDC}$ *light* se dictará en 4 clases teóricas de 3 horas. El esfuerzo estimado en preparar este curso es de 50 horas.

Resumiendo, planificamos para el experimento controlado:

- El contexto en el que se llevará a cabo
- La selección de los sujetos
- El diseño del experimento (1 factor con 2 alternativas)
- La capacitación a los sujetos
- La construcción de cursos

Si bien se planifica la construcción de los cursos de $PSP_{VDC}$ *light* y de métodos formales queda pendiente el armado propiamente dicho de estos cursos. Esta actividad que debe llevarse a cabo dentro de la planificación queda pendiente de realizar.

La figura 4.6 presenta las actividades de la fase de Operación; los Entrenamientos A y B *light* y la ejecución del experimento. La planificación de los entrenamientos se describen detalladamente en la sección Entrenamientos A y B *light*.

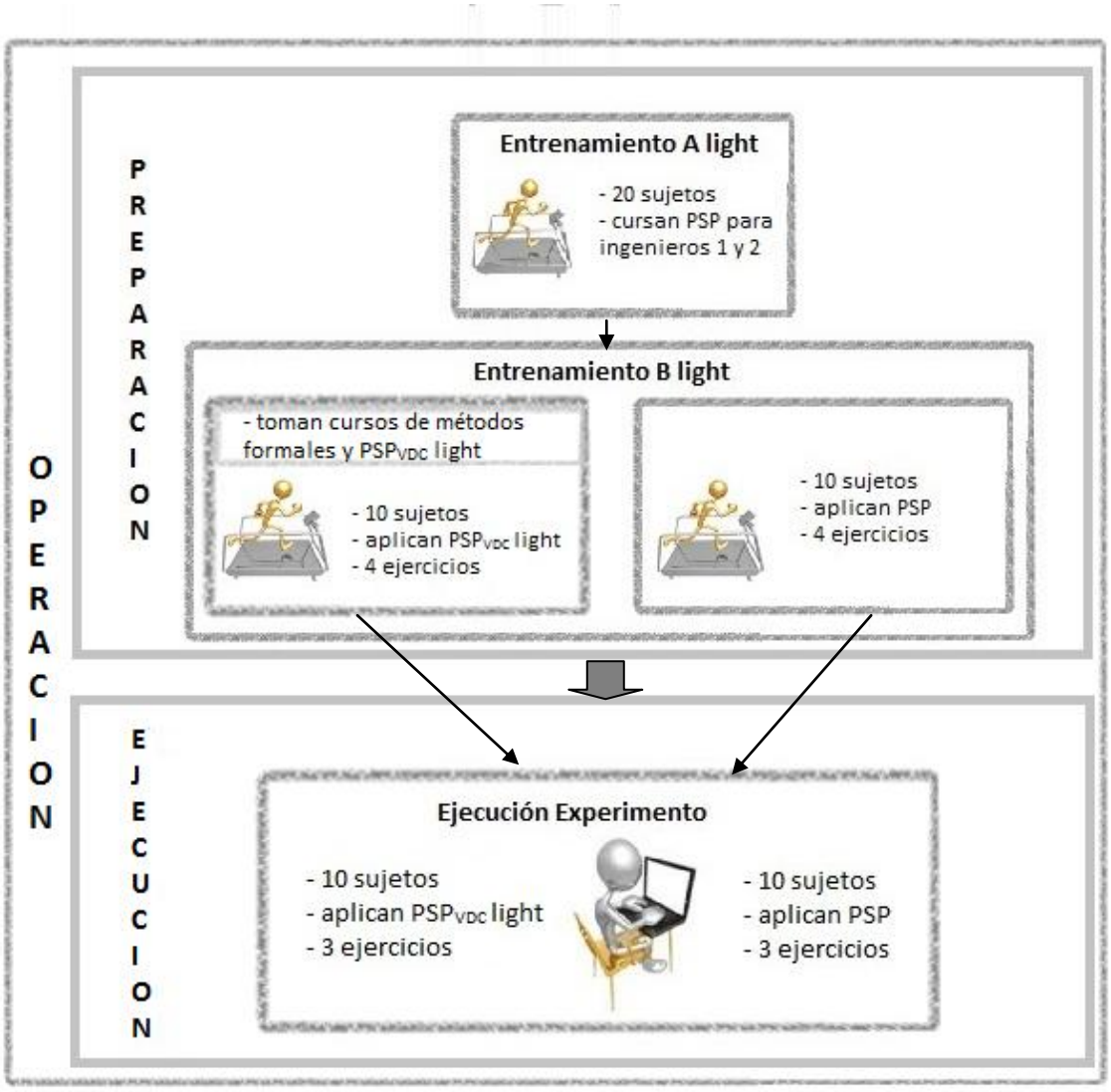



*Figura 4.6: Entrenamientos A y B light – Ejecución Experimento*

## Entrenamientos A y B *light*

En esta sección se presenta la planificación de los entrenamientos A y B *light* que tienen como objetivo capacitar a un grupo de entre 10 y 20 sujetos en PSP, $PSP_{VDC}$ *light* y métodos formales. Son entrenamientos secuenciales, primero se llevará a cabo el entrenamiento A y una vez que este finaliza se comienza con el entrenamiento B. Los sujetos son entrenados para luego formar parte de la ejecución del experimento formal.

Durante el entrenamiento A *light* se capacita a todos los sujetos con PSP a través del dictado de los cursos "*PSP for engineer 1*" y "*PSP for engineer 2*", en los cuales se introducen conceptos del PSP de forma gradual y se realizan 8 ejercicios. El dictado de estos cursos teórico/prácticos será de forma presencial/remota durante 4 semanas. Se dictarán 8 clases teóricas, dos por semana con una duración de aproximadamente 4 horas. Los ejercicios prácticos deberán serán realizados por los sujetos de forma individual en sus casas y deberán ser entregados antes de la siguiente clase teórica.

El entrenamiento B *light* se divide en dos, por un lado se entrena en métodos formales y $PSP_{VDC}$ *light* a la mitad de los sujetos participantes en el entrenamiento A, mientras la otra mitad mantiene la práctica con PSP. El motivo de esta decisión es que los estudiantes que solo van a aplicar PSP durante la ejecución del experimento no pierdan la práctica mientras los demás aprenden $PSP_{VDC}$ *light*.

La mitad de sujetos que es capacitada con métodos formales y $PSP_{VDC}$ *light* toma los cursos de métodos formales y $PSP_{VDC}$ *light*. Estos sujetos serán los que apliquen $PSP_{VDC}$ *light* durante la ejecución del experimento. En la figura 4.6 se ilustra esta decisión.

Durante los cursos de métodos formales y $PSP_{VDC}$ *light* se introducen los conceptos también de forma gradual realizando 4 ejercicios como se presentó en la sección anterior. Proponemos un cronograma similar al anterior, realizando primero el curso de métodos formales en 4 clases teóricas presenciales de 2,5 horas y luego el curso de $PSP_{VDC}$ *light*. El curso de $PSP_{VDC}$ *light* tendrá la parte teórica de forma presencial y los 4 ejercicios prácticos de forma remota. Debido a que el curso del $PSP_{VDC}$ *light* aun no fue armado debemos estimar la duración de las clases teóricas. Proponemos el dictado teórico en 4 clases, dos por semana con una duración de 3 horas. Los 4 ejercicios prácticos se realizarán de forma individual por los sujetos en sus casas.

Optamos por realizar los cursos de forma presencial/remota para lograr la cantidad estimada de sujetos participantes. Los sujetos son estudiantes que deben atender otras materias y posiblemente trabajos fuera del ambiente académico, por lo que una carga totalmente presencial puede causar una baja cantidad de participantes interesados.

## Planificación de la Ejecución y el Análisis

Si bien no se lleva a cabo la ejecución y el análisis del experimento en el contexto de esta tesis presentamos proponemos un diseño de ejecución y una propuesta para analizar los resultados obtenidos.

La operación consiste en la aplicación del PSP y del $PSP_{VDC}$ *light* por parte de los mismos sujetos participantes de los entrenamientos A y B *light* a un conjunto de 3 ejercicios. Los sujetos que durante el entrenamiento se capacitaron sólo con el PSP aplicarán el PSP, mientras que los que se capacitaron con el PSP y el $PSP_{VDC}$ *light* aplicarán el $PSP_{VDC}$ *light* como se muestra en la figura 4.6.

La ejecución del experimento se llevará a cabo en 3 sesiones de una semana de duración cada una. Se plantea la semana de lunes a viernes para que los sujetos apliquen el proceso asignado a un ejercicio durante esa semana. La aplicación del PSP y el $PSP_{VDC}$ *light* por parte de cada sujeto se realiza a distancia. El lunes de cada semana se enviará a los sujetos por correo electrónico el ejercicio correspondiente a dicha semana. Durante la semana los sujetos aplican el proceso asignado

sobre el ejercicio y a más tardar el viernes deberán hacer su entrega. Las entregas serán corregidas por el investigador durante sábado y domingo. En caso de encontrar problemas con la aplicación del proceso, se explican los problemas detectados al sujeto para que realice las correcciones y vuelva a entregar. La figura 4.7 ilustra las 3 sesiones correspondientes a la ejecución.

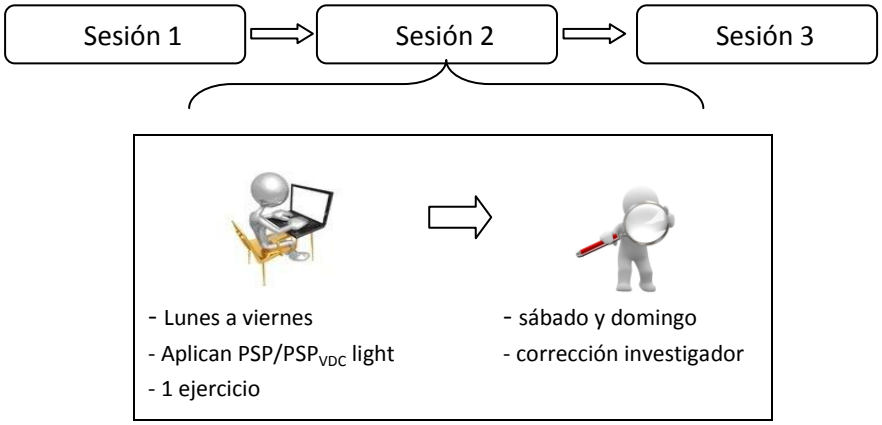



*Figura 4.7: Sesiones fase ejecución experimento controlado*

En lo que respecta a la interpretación de los datos proponemos aplicar inicialmente estadística descriptiva, es decir, la media y la mediana para medir la tendencia de los datos, la desviación estándar como medida de dispersión y el rango intercuartil. Este análisis nos va a permitir conocer el comportamiento de los datos, como se agrupan o dispersan con respecto a un valor central.

A posterior planificamos realizar pruebas de hipótesis para comparar la calidad y la productividad. Para poder aplicar test paramétricos se requiere que las muestras se aproximen a una distribución normal, y para realizar pruebas de normalidad se debe contar con una cantidad mínima de observaciones. En el caso de nuestro experimento tendremos como máximo 10 observaciones para cada alternativa (10 PSP y 10 PSP$_{VDC}$ *light*) por lo cual es preferible utilizar métodos no paramétricos.

Definimos la calidad (C) como la densidad de defectos detectados en la fase de UT y la productividad como la cantidad de líneas de código codificadas en una hora (LOCS/hora). Se utilizará el test no paramétrico de Mann-Whitney. Para el caso de la calidad la hipótesis nula indica que las medianas de calidad de ambos procesos son iguales, la hipótesis alternativa correspondiente indica que son distintas:

H0: μ C (PSP) = μ C (PSP$_{VDC}$ *light*)
H1: μ C (PSP) <> μ C (PSP$_{VDC}$ *light*)

Para la productividad (P) la hipótesis nula indica que las medianas de productividad de ambos procesos son iguales, la hipótesis alternativa correspondiente indica que son distintas:

H0: μ P (PSP) = μ P (PSP$_{VDC}$ *light*)
H1: μ P (PSP) <> μ P (PSP$_{VDC}$ *light*)

Los resultados de estos análisis nos van a permitir sacar conclusiones acerca de qué proceso resulta ser más productivo y cuál permite desarrollar productos de una mejora calidad.

# 4.3 Experimentación con $PSP_{VDC}$ *full*

En esta sección se presentan el caso piloto *full* y el experimento controlado *full*. Estos estudios son muy similares al caso piloto *light* y experimento *light*, por lo que nos centraremos en describir las actividades nuevas.

## 4.3.1 Caso piloto full

El caso piloto *full* consiste en la aplicación de $PSP_{VDC}$ *full* por parte del mismo investigador que aplica el piloto *light* a los ejercicios propuestos en el curso de "*PSP for engineers 2*". El objetivo del piloto *full* es detectar posibles problemas en el uso de $PSP_{VDC}$ *full*, que incorpora la fase de *Proof* con respecto al $PSP_{VDC}$ *light*; por lo que nos interesa evaluar dicha fase y si las herramientas propuestas son adecuadas. La intención es detectar problemas y corregirlos previamente a realizar el experimento controlado *full*.

El piloto *full* sigue de igual forma que el piloto *light* el proceso experimental que consiste de las fases de definición de objetivos, planificación, operación y análisis. La operación y el análisis están fuera del alcance de esta tesis de maestría. A continuación se describe la definición y planificación.

## Definición y Planificación

En el piloto *full* se mantienen el lenguaje de programación Java y el de especificación formal JML utilizados durante el piloto *light*.

Para el registro de los datos durante la operación de piloto *full* utilizaremos la herramienta "*PSP Student Workbook*" utilizada en piloto *light*. Dicha herramienta deberá ser adaptada para incorporar la fase de *proof* a sus formularios, estimaciones y métricas.

El piloto *full* requiere de una capacitación personal previa en técnicas de pruebas de programas (*proof verification*) y además se deberá seleccionar la/las herramientas a utilizar para ayudar a realizar la prueba. Como parte de este trabajo de tesis se realizó una búsqueda para tener un panorama de las herramientas disponibles. Se encontraron algunas herramientas obsoletas, en desuso, y otras que deberán de ser evaluadas en profundidad para su utilización. La herramienta que se seleccione, luego de un proceso de evaluación de las herramientas encontradas, será utilizada durante el piloto *full* y el experimento *full* como soporte a la construcción de la prueba de programas.

Presentamos a continuación las herramientas obtenidas de la búsqueda realizada:

- TACO (*Translation of Annotated COde*) es una herramienta desarrollada por el *Relational Formal Methods Research Group* del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires para realizar la verificación formal de programas Java respecto a su especificación en lenguaje JML [Galeotti 10].

- JmlForge es una herramienta desarrollada por el Instituto de Tecnología de Massachusetts (MIT) que es capaz de realizar la verificación estática de código Java respecto a su especificación en lenguaje JML. Realiza análisis modular (es decir, en los llamados a métodos utiliza la especificación del método llamado) y utiliza la técnica de Ejecución Simbólica para traducir un programa Java anotado con JML a una formula lógica [Dennis 09].

- Krakatoa, una herramienta de verificación estática basada en la plataforma de verificación Why y utilizando el asistente de evidencia Coq [Marche 04].

- Jack (Java Applet Correctness Kit) es una herramienta que proporciona un entorno para la verificación de programas Java con anotaciones JML, generando las obligaciones de prueba usando diferentes theorem provers [Barthe 07].

- Jahob es un verificador de programas escritos en un subconjunto de Java. Usando Jahob, los desarrolladores pueden probar estáticamente que los métodos cumplen sus contratos en todas las ejecuciones posibles [Kuncak 07].

- ESC/JAVA2 es una herramienta que intenta verificar parcialmente JML mediante el análisis estático del código del programa y sus anotaciones formales en tiempo de compilación. Examina el programa anotado y advierte de contradicciones entre las decisiones de diseño registrado en las anotaciones y el código actual, y también advierte de posibles errores de ejecución en el código. Utiliza un demostrador de teoremas automático para razonar sobre la semántica de los programas, lo que permite dar avisos estáticos de muchos errores que se detectan en tiempo de ejecución (referencia nula, límites de la matriz, errores de casteo de tipos, etc.) [Cok 04]

### 4.3.2 Un Experimento Controlado: PSP vs $PSP_{VDC}$ *full*

En esta sección presentamos la definición y planificación de un experimento controlado que busca comparar la calidad de los productos y la productividad que se obtiene al aplicar $PSP_{VDC}$ *full* y PSP.

El experimento *full* consiste en la aplicación de PSP y $PSP_{VDC}$ *full* por parte de un conjunto de sujetos en el mismo contexto que el experimento *light* (Facultad de Ingeniería). Siguiendo el proceso experimental anteriormente descrito, es necesario planificar el armado de dos cursos; uno que introduzca los conceptos de técnicas de prueba de programas y el curso de $PSP_{VDC}$ *full*.

Además se planifica el entrenamiento a los sujetos en el uso de dichas técnicas de prueba de programas, las herramientas de soporte a la prueba de programas y el $PSP_{VDC}$ *full*. El entrenamiento es análogo al presentado en el experimento *light*. Durante el Entrenamiento A *full* a los sujetos aprenden el PSP y durante el Entrenamiento B *full* se entrena a la mitad de los sujetos en técnicas de pruebas de programas y en el $PSP_{VDC}$ *full* y la otra mitad continua la práctica en el PSP. La figura 4.8 presenta las actividades de la fase de Operación; los Entrenamientos A y B *full* y la ejecución del experimento.

El diseño de la operación y como se pretenden analizar los resultados obtenidos son análogos a los presentados en el experimento *light*. Presentamos a continuación la definición y planificación del experimento *full*.

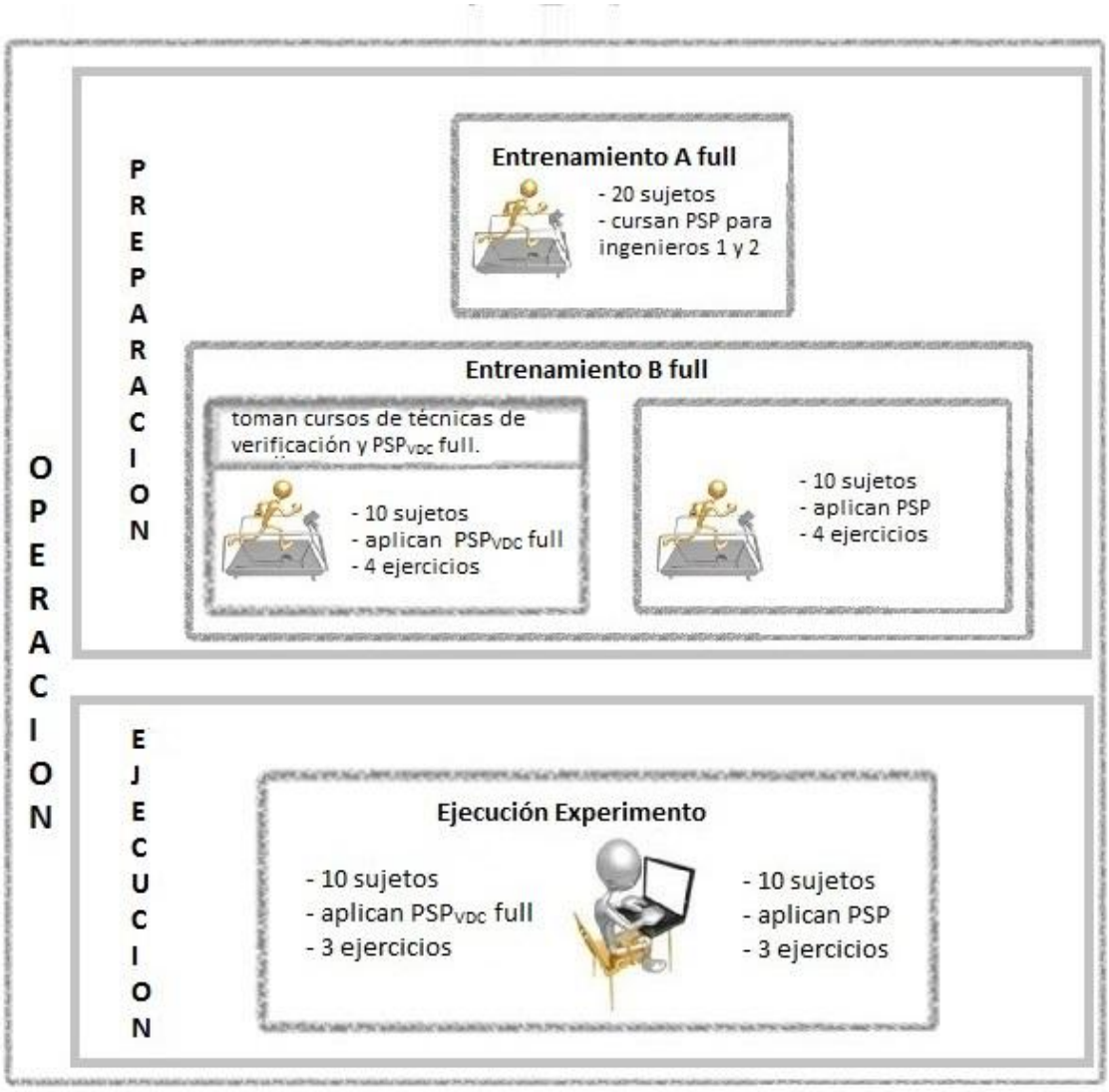



*Figura 4.8: Sesiones fase ejecución experimento controlado full*

## Definición y Planificación

El objetivo del experimento es comparar la calidad de los productos y la productividad que se obtiene al aplicar PSP y $PSP_{VDC}$ *full*. El diseño del experimento, la unidad experimental y las variables de respuesta son análogas a las del experimento *light*, cambiando el $PSP_{VDC}$ *light* por $PSP_{VDC}$ *full*. El experimento se realiza en el contexto de una materia en facultad de Ingeniería, donde los estudiantes participantes son distintos a los participantes del experimento *light*.

Se planifica el armado de dos cursos; Técnicas de pruebas de programas y $PSP_{VDC}$ *full*. El curso de técnicas de pruebas de programas deberá ser introductorio y se estima que su carga no supere las 20 horas de dictado.

El curso de $PSP_{VDC}$ *full* es una extensión del curso de $PSP_{VDC}$ *light*. Durante el curso $PSP_{VDC}$ *full* se presentan de forma gradual las fases del $PSP_{VDC}$ *light* y por último la nueva fase de *Proof* incorporada en el $PSP_{VDC}$ *full*.

Se define un subnivel más en el $PSP_{VDC}$ *full*, nivel 4 que introduce los conceptos de técnicas de prueba de programas y la nueva fase de *Proof*.

El curso se dictará en 5 días, donde en los primeros 3 días se lleva a cabo el dictado teórico y los ejercicios prácticos propuestos en los primeros 3 días del curso de $PSP_{VDC}$ *light*. El 4to día del curso $PSP_{VDC}$ *light* consiste en aplicar el proceso $PSP_{VDC}$ *light* completo, sin embargo, el 4to día del curso $PSP_{VDC}$ *full* se destinará a presentar

la nueva fase de *Proof*, la herramienta a utilizar para ayudar a generar la prueba y a realizar un ejercicio. El día 5 se aplicará el $PSP_{VDC}$ *full* completo en un quinto ejercicio de programación.

Se detalla a continuación una propuesta de temas y fases a introducir en cada uno de los 5 días de curso:

Día 1. Se introducen los conceptos de pseudo code y de construcción de casos de pruebas. Se explican cómo y por qué estas actividades se realizan en nuevas fases y se exponen los scripts correspondientes.

Día 2. Se introduce el concepto de especificación formal y compilación de la especificación formal. Se detallan estas nuevas fases junto con sus scripts.

Día 3. Se introduce el concepto de las revisiones (ya visto en PSP) y se explican las nuevas fases de revisión del pseudo código y revisión de la especificación formal, así como los *scripts* y *check list*.

Día 4. Se introducen los conceptos de técnicas de pruebas de programas, la nueva fase de *proof* y la herramienta que servirá de ayuda a generar la prueba.

Día 5. Se aplica el $PSP_{VDC}$ *full* completo al último ejercicio sin agregados.

El curso de $PSP_{VDC}$ *full* se planifica dictar durante 5 clases teóricas de 3 horas y además se deberán armar 5 ejercicios prácticos. Estimamos 60 horas para preparar dicho curso.

# 4.4 Conclusiones

La experimentación es una técnica formal, rigurosa y limitada por el costo en tiempo y en recursos. Una buena planificación evita que los experimentos puedan quedar invalidados, atrasando la investigación por meses e incluso años.

En este capítulo se presenta la planificación de dos experimentos controlados con el objetivo de comparar la calidad y productividad de PSP, $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full*. El experimento *light* compara PSP y $PSP_{VDC}$ *light* y el experimento *full* compra PSP y $PSP_{VDC}$ *full*. Ambos experimentos se llevarán a cabo en el contexto de Facultad de Ingeniería con la participación de estudiantes.

Definimos y planificamos ambos experimentos. Se establecen los objetivos, los diseños experimentales, los test de hipótesis, se planifica el armado de los cursos y los entrenamientos a los sujetos. Además, se propone el plan de la ejecución que se desea llevar adelante y cómo se pretenden analizar los resultados. También se planifican dos pilotos para detectar posibles problemas en el uso de las herramientas y los procesos. El plan también identifica tareas pendientes, como por ejemplo, adaptaciones a las herramientas, creación de cursos a ser dictados antes del experimento, etc.

Esta planificación en detalle permitirá ejecutar los experimentos de forma controlada, evitando malgastar los recursos asignados y evitando encontrarse con problemas inesperados durante la ejecución de los mismos.

# Capítulo 5.

# Conclusiones y Trabajo a Futuro

Este capítulo incluye las conclusiones (Sección 5.1) y el trabajo a futuro (Sección 5.2) de la Tesis.

## 5.1 Conclusiones

Al ser el desarrollo de software una actividad creativa e intelectual realizada por seres humanos es común que durante un proyecto el equipo de desarrollo cometa errores. Esto se debe tanto a la complejidad actual del software como a la propia naturaleza humana. Normalmente estos errores terminan en defectos en el producto de software y cuando el software se está ejecutando estos pueden causar fallos.

La búsqueda de formas de desarrollar software con bajo número de defectos ha dado lugar al desarrollo de un variado número de procesos y métodos. El cometido de dichos procesos es construir software de calidad, en el plazo estipulado y dentro de los costos previstos.

El Personal Software Process (PSP) aplica disciplina de proceso y gestión cuantitativa al trabajo individual del ingeniero de software. Promueve la utilización de prácticas específicas durante todas las etapas del desarrollo con el objetivo de mejorar la calidad del producto y aumentar la productividad del individuo [Humphrey 05, Humphrey 06].

Por otro lado, los métodos formales son un conjunto de técnicas para especificar, desarrollar y verificar sistemas software mediante el uso del lenguaje matemático formalizado. Consisten en especificar formalmente y luego demostrar matemáticamente que los programas producidos cumplen con dichas especificaciones. El diseño por contrato (DbC) es una técnica para el diseño de los componentes de un sistema de software, mediante el establecimiento de las condiciones (pre y post condiciones) en un lenguaje formal. Cuando las técnicas y herramientas incorporadas permiten realizar la demostración de la concordancia de cada pieza de software con su especificación estamos en presencia de un método formal generalmente llamado diseño por contrato verificado (VDbC).

La tesis consta de tres resultados principales: el $PSP_{VDC}$ que adapta al PSP para incorporar VDbC, la definición de dos experimentos controlados para comparar el $PSP_{VDC}$ con el PSP y una revisión sistemática de la literatura relacionada con este trabajo.

En esta tesis se investigó de qué forma se puede integrar el VDbC al PSP con el objetivo de que se puedan desarrollar productos de mejor calidad (que los desarrollados usando el PSP) manteniendo (o mejorando) la productividad. Para integrar y aprovechar las bondades de ambos enfoques construimos un proceso de desarrollo de software, llamado $PSP_{VDC.}$

El $PSP_{VDC}$ es una adaptación al PSP que mantiene la disciplina y la instrumentación del PSP y permite al desarrollador utilizar el enfoque VDbC. Las adaptaciones centrales que realizamos son las siguientes:

- Se incorporan nuevas fases y se adaptan fases ya existentes
    - *formal specification* (fase nueva)
    - *formal specification review* (fase nueva)
    - *formal specification compile* (fase nueva)
    - *test case construct* (fase nueva)
    - *pseudo code* (fase nueva)
    - *pseudo code review*(fase nueva)
    - *proof* (fase nueva)

- *design (fase adaptada)*

- Se construyen los scripts que sirven de guía para realizar las actividades de las fases mencionadas
  - Se adapta el *Process Script*,
  - Se adapta el *Development Script*
  - Se construye el *Specification Review Script*

- Se construyen los *templates* necesarios para el uso del $PSP_{VDC}$
  - Se construye el *Formal Specification Review Checklist Template*
  - Se construye el *Formal Specification Standard Template*

- Se adaptan algunas medidas de control y calidad del $PSP_{VDC}$ y se proponen nuevas.
  - *Yield* (*process*) (adaptado)
  - *Defect removal efficiency* (FSR) (adaptado)
  - *Defect removal efficiency* (FSC) (adaptado)
  - *Defect removal efficiency* (FCR) (adaptado)
  - *Defect removal efficiency* (FRF) (adaptado)
  - DRL (FSR /UT) (Nuevo indicador)
  - DRL (PCR /UT) (Nuevo indicador)
  - DRL (FSC/UT) (Nuevo indicador)
  - DRL (PRF/UT) (Nuevo indicador)
  - *Appraisal Cost of Quality* (adaptado)
  - *Percent Failure* COQ (adaptado)

Como resultado de esta tesis se especificó completamente el $PSP_{VDC}$ de forma que pueda ser utilizado por un ingeniero de software. En el $PSP_{VDC}$ se agregan varias fases para dar soporte al VDbC. El objetivo es que el uso de este nuevo proceso logre productos de mejor calidad que los desarrollados con el PSP manteniendo los costos controlados.

Con el propósito de conocer los resultados de aplicar el $PSP_{VDC}$ presentamos la definición, la planificación y el diseño de dos experimentos controlados. Se establecieron los objetivos, los diseños experimentales, los test de hipótesis, se planificó el armado de los cursos y los entrenamientos a los sujetos. Todos estos aspectos son fundamentales para poder llevar adelante una correcta experimentación.

Las hipótesis de ambos experimentos consisten en comparar la calidad y la productividad de los procesos. Los sujetos que utilizarán los procesos son estudiantes de grado de la Facultad de Ingeniería. Además, se presenta el plan de la ejecución que se desea llevar adelante y cómo se pretenden analizar los resultados. También se planifican dos estudios pilotos para detectar posibles problemas en el uso de las herramientas y los procesos previo a llevar adelante los experimentos.

Se definieron dos niveles para el $PSP_{VDC}$ denominados $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full*. El $PSP_{VDC}$ *light* no utiliza la demostración formal de programas mientras que el $PSP_{VDC}$ *full* sí.

Los experimentos son denominados experimento *light* y experimento *full*. Durante el experimento *light* un grupo de estudiantes aplican el $PSP_{VDC}$ *light* y el PSP a un conjunto de ejercicios; mientras que en el experimento *full* otro grupo de estudiantes aplican el $PSP_{VDC}$ *full* y el PSP a los mismos ejercicios del *light*. El diseño de ambos experimentos es de un factor (proceso de desarrollo de software) con dos alternativas; PSP y $PSP_{VDC}$ *light* en el experimento *light* y PSP y $PSP_{VDC}$ *full* en el experimento *full*.

Los resultados de los experimentos nos permitirán concluir acerca de la calidad y productividad del $PSP_{VDC}$ respecto al PSP. Contar con una planificación en detalle nos permitirá ejecutar los experimentos de forma controlada, evitando malgastar los recursos asignados y evitando encontrar problemas inesperados durante la ejecución de los mismos. Además, los experimentos nos permitirán conocer si el agre-

gar la fase de *proof* al experimento *full* provoca mejoras respecto al experimento *light*.

También presentamos una revisión sistemática de la literatura que resume la información existente sobre adaptaciones realizadas al PSP y particularmente aquellas que incorporan métodos formales. Se encontraron 5 trabajos que adaptan el PSP, 3 de ellos adaptan el PSP para usar métodos formales, otro adapta el PSP para usar programación por pares y el último para usar técnicas de desarrollo de software dirigido por modelos.

Las propuestas encontradas son distintas en la forma en que realizan la adaptación. Dos de las tres propuestas que incorporan métodos formales agregan nuevas fases al proceso para dar soporte a las actividades incorporadas, sin embargo, la restante opta por mantener la estructura de fases del PSP e incorporar nuevas actividades a las fases ya existentes.

El $PSP_{VDC}$ incorpora nuevas fases y modifica fases existentes al PSP. El $PSP_{VDC}$, a diferencia de las otras propuestas, se compone de una mayor cantidad de fases. Las tres propuestas que incorporan métodos formales proponen 6, 8 y 12 fases, mientras que en $PSP_{VDC}$ definimos 15 fases. Optamos por tener más fases de forma de poder obtener una mayor granularidad en los datos recolectados (esfuerzo, defectos inyectados y removidos por fase, entre otros).

En $PSP_{VDC}$ agregamos algunas fases que si bien no están directamente relacionadas con VDbC podrían ayudarnos a mejorar el proceso en base a los resultados de los experimentos. Específicamente, la fase *test case construct* se propone como una nueva fase (independiente del diseño) con el objetivo de conocer el costo empleado en la generación de los casos de prueba. Los resultados de los experimentos podrían mostrar que la inclusión de VDbC reduce significativamente los defectos que llegan a UT, con lo cual podríamos pensar en eliminar las fases de *test case construct* y *Unit Test* del $PSP_{VDC}$.

Contar con las adaptaciones realizadas al PSP que incorporan métodos formales y el $PSP_{VDC}$ totalmente definido brinda conocimiento acerca de las formas posibles de combinar el enfoque de PSP y los métodos formales. La planificación y diseño de los experimentos controlados permiten ejecutar de forma controlada los experimentos y comprobar o refutar la mejora en la calidad de los productos de software al aplicar $PSP_{VDC.}$

# 5.2 Trabajo a Futuro

Parte del trabajo de esta tesis consistió en definir y planificar los casos pilotos y experimentos controlados para comparar el $PSP_{VDC}$ con el PSP. Nuestro trabajo a futuro es llevar a cabo el armado de los cursos necesarios y adaptar la herramienta de soporte al proceso. Luego, pretendemos ejecutar los experimentos, realizar los análisis de datos correspondientes y reportar los resultados.

Además, a partir de la construcción de $PSP_{VDC}$ y sus dos niveles $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full* surgen varias líneas de trabajo sobre las cuales continuar investigando. Los resultados de los experimentos *light* y *full* permitirán conocer si la aplicación de $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full* aumentan la calidad de los productos respecto a la aplicación del PSP. En caso que así sea, es de interés comparar las propuestas $PSP_{VDC}$ *light* y $PSP_{VDC}$ *full* entre sí con el objetivo de conocer la propuesta que logra la mejor calidad de los productos y si se logra mantener o mejorar la productividad.

El trabajo de esta tesis se lleva a cabo en el contexto del PSP, por lo que una línea interesante es investigar si es posible el uso del $PSP_{VDC}$ en el contexto del Team Software Process [Humphrey 06]. Conocer si se requiere adaptar el TSP o no para poder usar el $PSP_{VDC}$ en el contexto de un equipo utilizando el TSP.

Por último es importante aclarar que la adaptación propuesta es sólo una de las posibles pero no la única. Otra línea de investigación es analizar otras posibles adaptaciones al PSP incorporando métodos formales, compararlas y analizar si se logran mejores resultados.

# References


**[Babar 05]**
Abdul Babar, John Potter. *Adapting the Personal Software Process (PSP) to Formal Methods.* Australian Software Engineering Conference (ASWEC'05), 2005, pp.192-201.

**[Barthe 07]**
G. Barthe, L. Burdy, J. Charles B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova and A. Requet. JACK: a tool for validation of security and behaviour of Java applications, Proceedings of 5th International Symposium on Formal Methods for Components and Objects, 2007.

**[Basili 94]**
Victor R. Basili and Gianluigi Caldiera and H. Dieter Rombach. The Goal Question Metric Approach, Encyclopedia of Software Engineering. Wiley 1994.

**[Cok 04]**
Cok, D. R. and Kiniry, J. ESC/Java2: Uniting ESC/Java and JML. In Proceedings of the CASSIS'05. Lecture Notes in Computer Science, vol. 3362. Springer, 108--128., 2004.

**[Dennis 09]**
Gregory D. Dennis. A Relational Framework for Bounded Program Verification. PhD thesis, Massachusetts Institute of Technology, 2009.

**[Galeotti 10]**
Juan Pablo Galeotti, Nicolas Rosner, Carlos Gustavo Lopez Pombo, Marcelo Frias. Analysis of Invariants for Efficient Bounded Verification. International Symposium on Software Testing and Analysis (ISSTA), Trento, Italy, July 2010

**[Humphrey 05]**
Watts S. Humphrey. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional; March. 2005

**[Humphrey 06]**
Watts S. Humphrey. Tsp (sm) Coach Dvlp Teams. Pearson Education, 2006

**[Kitchenham 07]**
Guidelines for performing Systematic Literature Reviews in Software Engineering, Version 2.3, EBSE Technical Report EBSE-2007-01

**[Kuncak 07]**
V. Kuncak. Modular Data Structure Verification. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

**[Kusakabe 12]**
Kusakabe, S., Omori, Y. and Araki K, A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report. TSP Symposium 2012, St. Petersburg, FL.

**[Leavens 06]**

Gary T. Leavens and Yoonsik Cheon,Design by Contract with JML, 2006

**[Macchi 12]**
Macchi, D.; Solari, M., "Software inspection adoption: A mapping study," *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En* , vol., no., pp.1,8, 1-5 Oct. 2012

**[Marche 04]**
C. Marché, C. Paulin-Mohring, and X. Urbain, "The Krakatoa tool for certification of Java/JavaCard programs annotated in JML", Journal of Logic and Algebraic Programming, 2004.

**[Pascoal 12]**
Pascoal, J. Integrating Model-Driven Engineering Techniques in the Personal Software Process. TSP Symposium 2012, St. Petersburg, FL.

**[Sommerville 05]**
Ian Sommerville. Ingeniería del Software 7a edición. University of Lancaster. ISBN 8478290745, 2005

**[Suzumori 03]**
Hisayuki Suzumori, Haruhiko Kaiya, Kenji Kaijiri, *VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development,* 27th Annual International Computer Software and Applications Conference, 2003

**[Why 02]**
The The Why certification tool is © 2002–2006 Laboratoire de Recherche en Informatique (www.lri.fr). It is open source and freely available under the terms of the GNU GENERAL PUBLIC LICENSE Version 2. See the files COPYING and GPL in the distribution.

**[Williams 01]**
Williams, L. Integrating pair programming into a software development process. Software Engineering Education Conference, Proceedings Jan 1, 2001, p27-36, 10p

# ANEXO A

En este anexo se presenta el reporte técnico del PEDECIBA 13-01. Se presentan conceptos teóricos básicos de la Ingeniería de Software Empírica, así como también técnicas y herramientas de experimentación y de estudio de casos.

**PEDECIBA Informática**
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 13-01

# Conceptos de ingeniería de Software empírica. Versión 2.0

Cecilia Apa, Stephanie de León,

Silvana. Moreno,Rosana Robaina,

Diego Vallespir

2013

Conceptos de ingeniería de software empírica v. 2.0

C. Apa, S.de Leóin, S. Moreno, R. Robaina, D. Vallespir

ISSN 0797-6410
Reporte Técnico RT 13-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2013

# Conceptos de Ingeniería de Software Empírica

*Versión 2.0*

Cecilia Apa
Stephanie de León
Silvana Moreno
Rosana Robaina
Diego Vallespir

Reporte Técnico InCo/Pedeciba-2013 TR:13-01
Febrero, 2013

**Resumen**

En este artículo se presentan conceptos teóricos básicos de la Ingeniería de Software Empírica, así como también técnicas y herramientas de experimentación y de estudio de casos. La experimentación es un método que se usa para corresponder ideas o teorías con la realidad, proporcionando evidencia que soporte las hipótesis o suposiciones que se creen válidas. La experimentación en la Ingeniería de Software no ha alcanzado aún la madurez que tiene la experimentación en otras disciplinas (por ejemplo, biología, química, sociología). Sin embargo, en los últimos años ésta área en la Ingeniería de Software ha cobrado gran importancia y su actividad ha sido creciente.

Un experimento controlado intenta validar ciertas hipótesis mediante un *ambiente* creado para tal fin. Para esto se controlan ciertas variables (independientes) y se observa el resultado que arrojan ciertas otras variables (dependientes). Luego, estadísticamente, se rechazan o aceptan las hipótesis propuestas.

Un estudio de casos es un método de aprendizaje acerca de un fenómeno; se basa en el entendimiento de dicho fenómeno, el cual se obtiene a través de la descripción y análisis del mismo dentro de su contexto. Los estudios de casos no generan resultados sobre las relaciones causales como lo hacen los experimentos controlados, sino que proporcionan una comprensión más profunda de los fenómenos bajo estudio.

Aquí se presentan dos procesos, uno para realizar experimentos controlados y otro para conducir estudios de casos. Estos procesos son utilizados por Grupo de Ingeniería de Software de esta Facultad para realizar sus estudios empíricos.

## Control de Versiones del Documento

| Versión | # Reporte | Fecha | Cambios |
|---|---|---|---|
| Versión 1 | RT 10-02, 2010 | Febrero, 2010 | Versión inicial |
| Versión 2 | RT 13-01, 2013 | Febrero 2013 | ▪ Se agrega el método de investigación Estudio de casos.<br>▪ Se agregan los trabajos de investigación del Grupo de Ingeniería de Software, UdelaR<br>▪ Se ajustan la introducción, resumen y conclusiones. |

## Índice general

## Índice de figuras

## Índice de cuadros

## 1. Introducción

Este reporte tiene como objetivo presentar los fundamentos de la Ingeniería de Software Empírica (ISE), se presentan los conceptos de experimentos formales (controlados) y estudios de casos. Se pretende que este documento sea utilizado por Proyectos de Grado de la carrera Ingeniería en Computación, Maestrías, Doctorados, etc. que se encuentran realizando trabajos de ISE con el Grupo de Ingeniería de Software (GrIS). De esta manera los estudiantes de grado y posgrado pueden usar este documento como punto de partida para comprender la ISE. Además, pueden incluir este documento como parte de su informe de proyecto o su Tesis evitando tener un enfoque distinto de la ISE en cada Proyecto de Grado y/o Tesis.

Este reporte se basa casi completamente en los libros *Experimentation in Software Engineering: An Introduction* (18), *Basics of Software Engineering Experimentation* (4), *Software Metrics - A Rigorous And Practical Approach* (3) y *Case Study Research in Software Engineering* (7).

En la sección 2 se presentan los distintos enfoques y estrategias de la ISE. Una de estas estrategias es la de experimentos formales, estos se describen en la sección 3. En la sección 4 se describe un proceso para llevar adelante un experimento formal. Otra estrategia es la de Estudios de Casos, que se describe en la sección 5. Por último en la sección 6 se describen los trabajos de investigación que ha llevado adelante el GrIS en los ultimos años.

## 2. Enfoques y Estrategias

La ISE utiliza métodos y técnicas experimentales como instrumentos para la investigación. La evidencia empírica proporciona un soporte para la evaluación y validación de atributos (p.e. costo, eficiencia, calidad) en varios tipos de elementos de Ingeniería de Software (p.e. productos, procesos, técnicas, etc.). Se basa en la experimentación como método para corresponder ideas o teorías con la realidad, la cual refiere a mostrar con hechos las especulaciones, suposiciones y creencias sobre la construcción de software.

Se pueden distinguir dos enfoques diferentes al realizar una investigación empírica: el enfoque cualitativo y el cuantitativo. El enfoque **cualitativo** se basa en estudiar la naturaleza del objeto y en interpretar un fenómeno a partir de la concepción que las personas tienen del mismo. Los datos que se obtienen de estas investigaciones están principalmente compuestos por texto, gráficas e imágenes, entre otros.

El enfoque **cuantitativo** se corresponde con encontrar una relación numérica entre dos o más grupos. Se basa en cuantificar una relación o comparar variables o alternativas bajo estudio. Los datos que se obtienen en este tipo de estudios son siempre valores numéricos, lo que permite realizar comparaciones y análisis estadístico.

Es posible utilizar los enfoques cualitativos y cuantitativos para investigar el mismo tema, pero cada enfoque responde a diferentes interro-

gantes. Se puede considerar que estos enfoques son complementarios más que competitivos, ya que el enfoque cualitativo puede ser usado como base para definir la hipótesis que luego puede ser correspondida cuantitativamente con la realidad. Cabe destacar que las investigaciones cuantitativas pueden obtener resultados más justificables y formales que los cualitativos.

Hay 3 tipos principales de técnicas o estrategias para la investigación empírica: las encuestas, los estudios de casos y los experimentos.

Las **encuestas** se utilizan o bien cuando una técnica o herramienta ya ha sido usada o antes de comenzar a hacerlo. Son estudios retrospectivos de las relaciones y los resultados de una situación. Se puede realizar este tipo de investigación cuando una técnica, o herramienta ya ha sido utilizada o antes de que ésta sea introducida. Las encuestas son realizadas sobre una muestra representativa de la población, y luego los resultados son generalizados al resto de la población. El ámbito donde son más usadas es en ciencias sociales, por ejemplo, para determinar cómo la población va a votar en la siguiente elección.

En la Ingeniería de Software Empírica las encuestas se utilizan de forma similar, se obtiene un conjunto de datos de un evento que ha ocurrido para determinar cómo reacciona la población frente a una técnica, herramienta o método particular, o para determinar relaciones o tendencias. En un estudio es fundamental seleccionar correctamente las variables a estudiar, pues de ellas dependen los resultados que se pueden obtener. Si los resultados no permiten concluir sobre los objetivos del estudio se han elegido mal las variables.

Una de las características más relevantes de las encuestas es que proveen un gran número de variables para estudiar. Esto hace posible construir una variedad de modelos y luego seleccionar el que mejor se ajusta a los propósitos de la investigación, evitando tener que especular cuáles son las variables más relevantes. Dependiendo del diseño de la investigación (cuestionario) las encuestas pueden ser clasificadas como cualitativas o cuantitativas.

Los **estudios de casos** son métodos observacionales, se basan en la observación de una actividad o proyecto durante su curso. Son utilizados para monitorear proyectos, o actividades y para investigar entidades o fenómenos en un período específico.

El nivel de control de la ejecución es menor en los estudios de casos que en los experimentos. Esto se debe principalmente a que en los estudios de casos no se controla, sólo se observa, contrario a lo que ocurre en los experimentos.

Los estudios de casos son muy útiles en el área de Ingeniería de Software, se usan en la evaluación industrial de métodos y herramientas. Además, son fáciles de planificar aunque los resultados son difíciles de generalizar y comprender. Los estudios de casos no manipulan las variables, sino que éstas son determinadas por la situación que se está investigando.

Al igual que las encuestas, los estudios de casos pueden ser clasificados como cualitativos o cuantitativos dependiendo de lo que se quiera investigar del proyecto en curso.

Los **experimentos** son generalmente ejecutados en un ambiente de laboratorio, el cual brinda un alto grado de control. El objetivo en un experimento es manipular una o más variables y controlar el resto. Un experimento es una técnica formal, rigurosa y controlada de llevar a cabo una investigación.

## 3. Experimentos Formales

Como se mencionó anteriormente, los experimentos son una técnica de investigación en la cual se quiere tener un mejor control del estudio y del entorno en el que éste se lleva a cabo.

Los experimentos son apropiados para investigar distintos aspectos de la IS, como ser: confirmar teorías, explorar relaciones, evaluar la exactitud de los modelos y validar medidas. Tienen un alto costo respecto de las otras técnicas de investigación, pero a cambio ofrecen un control total de la ejecución y son de fácil replicación.

### 3.1. Terminología

En esta sección se presentan los términos más comunmente usados en diseño experimental. Se usan dos ejemplos de experimentos a lo largo de esta sección para introducir dichos términos.

En el primer ejemplo se tiene un experimento en el campo de la medicina, mediante el cual se quiere conocer la efectividad de los analgésicos en las personas entre 20 y 40 años de edad, llamado «Efec-Analgésicos».

En el segundo ejemplo, se quiere conocer la efectividad de 5 técnicas de verificación sobre un conjunto de programas, llamado «Efec-Técnicas».

Los objetos sobre los cuales se ejecuta el experimento son llamados **Unidades Experimentales** u objetos experimentales. La unidad experimental en un experimento de Ingeniería de Software podría llegar a ser el proyecto de software como un todo o cualquier producto intermedio durante el proceso.

Para *Efec-Analgésicos* se tiene que la unidad experimental es un grupo de personas entre 20 y 40 años de edad, en ese grupo de personas es en donde se observa el efecto de los analgésicos. En el ejemplo de *Efec-Técnicas*, se tiene que la unidad experimental es el conjunto de programas sobre los cuales se aplican las técnicas de verificación.

Aquellas personas que aplican los métodos o técnicas a las unidades experimentales se les llama **Sujetos Experimentales**. A diferencia de otras disciplinas, en la IS los sujetos experimentales tienen un importante efecto en los resultados del experimento, por lo tanto es una variable que debe ser cuidadosamente considerada.

En *Efec-Analgésicos* los sujetos son aquellas personas que administran los analgésicos a ser consumidos por los pacientes (enfermeros por ejemplo). Cómo los enfermeros administran los analgésicos a los pacientes no es algo que se espere vaya a afectar el experimento. La forma en que un enfermero administra un analgésico a un paciente es

poco probable que sea diferente a la de otro, y aunque lo fuera, no se espera que afecte los resultados del experimento.

En *Efec-Técnicas* los sujetos pueden ser ingenieros que aplican la técnica en un conjunto particular de programas (unidad experimental). En este caso, los resultados del experimento podrían diferir mucho de acuerdo a la formación y experiencia de los ingenieros, así como también la forma en que las técnicas son aplicadas, incluso el estado de ánimo del verificador podría influir en los resultados.

El resultado de un experimento es llamado **Variable de Respuesta**. Este resultado debe ser cuantitativo. Una variable de respuesta puede ser cualquier característica de un proyecto, fase, producto o recurso que es medida para verificar los efectos de las variaciones que se provocan de una aplicación a otra. En ocasiones, a una variable de respuesta se le llama también variable dependiente.

En *Efec-Analgésicos* la efectividad podría ser medida en el grado de alivio del dolor en un determinado lapso de tiempo, o bien qué tan rápido el analgésico alivia el dolor. En ambos casos, la variable debe ser expresada cuantitativamente. En el primer caso se podría tener una escala, en la cual cada valor signifique un grado de alivio del dolor, en el segundo caso, el lapso de tiempo en que el analgésico es efectivo, se podría medir en minutos.

Para *Efec-Técnicas* la efectividad podría ser medida de acuerdo a la cantidad de defectos que encuentra la técnica sobre la cantidad de defectos totales del software verificado.

Un **Parámetro** es cualquier característica que permanezca invariable a lo largo del experimento. Son características que no influyen o que no se desea que influyan en el resultado del experimento o en la variable de respuesta. Los resultados del experimento serán particulares a las condiciones definidas por los parámetros. El conocimiento resultante podrá ser generalizado solamente considerando los parámetros como variables en sucesivos experimentos y estudiando su impacto en las variables de respuesta.

En el ejemplo de *Efec-Analgésicos* se tiene que el rango de edades (entre 20 y 40 años de edad) es un parámetro del experimento, los resultados serán particulares para el rango establecido.

En *Efec-Técnicas* un parámetro posible es el tamaño del software a ser verificado (por ejemplo: que tenga entre 200 y 500 LOCs). Otro parámetro para este experimento podría ser la experiencia de los verificadores, en este caso se podría fijar la experiencia en un determinado nivel.

Cada característica del desarrollo de software a ser estudiada que afecta a las variables de respuesta se denomina **Factor**. Cada factor tiene varias alternativas posibles. Lo que se estudia, es la influencia de las alternativas en los valores de las variables de respuesta. Los factores de un experimento son cualquier característica que es intencionalmente modificada durante el experimento y que afecta su resultado.

El factor en *Efec-Analgésicos* es «los analgésicos», en *Efec-Técnicas* tenemos que el factor es «las técnicas de verificación». Para ambos casos el factor se varía intencionalmente (se varía el tipo de analgé-

sico o tipo de técnica de verificación) para ver cómo afecta en la efectividad.

Los posibles valores de los factores en cada unidad experimental son llamados **Alternativas** o niveles. En algunos casos también se les llama tratamientos.

Las alternativas de *Efec-Analgésicos* son los distintos tipos de analgésicos que se estudian en el experimento (p.e. Aspirina, Zolben, etc). De igual forma, para *Efec-Técnicas* las distintas alternativas son los 5 tipos distintos de técnicas que se estudian.

El intento de ajustar determinadas características de un experimento a un valor constante no es siempre posible. Es inevitable y a veces indeseable tener variaciones de un experimento a otro. Éstas variaciones son conocidas como **Bloqueo de Variables** y dan lugar a un determinado tipo de diseño experimental, llamado *block design*.

Una variable indeseada para *Efec-Analgésicos* podría ser el «umbral del dolor». Si se aplica una alternativa de analgésico a personas con umbral del dolor alto y otra alternativa a personas con umbral del dolor bajo, se tendría una variación indeseada, ya que la efectividad que se mida de los distintos tipos de analgésico va a variar no solamente por el tipo de analgésico administrado sino por el nivel de umbral del dolor del paciente al cual se lo administra.

En el caso de *Efec-Técnicas*, podría resultar que la experiencia de los verificadores resultase una variación indeseada si no se la tiene en cuenta previamente. Una forma de bloquear la experiencia en verificación podría ser dividir a los participantes en dos grupos: uno de verificadores experientes y otro sin experiencia.

Cada ejecución del experimento que se realiza en una unidad experimental es llamada **experimento unitario** o experimento elemental. Lo que significa que cada aplicación de una combinación de alternativas de factores por un sujeto experimental en una unidad experimental es un experimento elemental.

Un experimento elemental es cada terna <$analgsico_i$, $enfermero_j$, $paciente_k$> para el ejemplo de *Efec-Analgésicos*. Para el ejemplo de *Efec-Técnicas* sería la terna <$tecnica_i$, $verificador_j$, $software_k$>.

La figura 1 ilustra la interacción entre los distintos tipos de componentes de un experimento.

### 3.2. Principios generales de diseño

Muchos aspectos deben ser tenidos en cuenta cuando se diseña un experimento. Los principios generales de diseño son: aleatoriedad, bloqueo y balance. A continuación se describe en qué consiste cada principio.

**Aleatoriedad:** el principio de aleatoriedad es uno de los principios de diseño más importantes. Todos los métodos de análisis estadístico requieren que las observaciones sean de variables independientes aleatorias. Por consiguiente, tanto las alternativas de los factores como los sujetos tienen que ser elegidos de forma aleatoria, ya que los sujetos tienen un impacto crítico en el valor de las variables de respuesta.

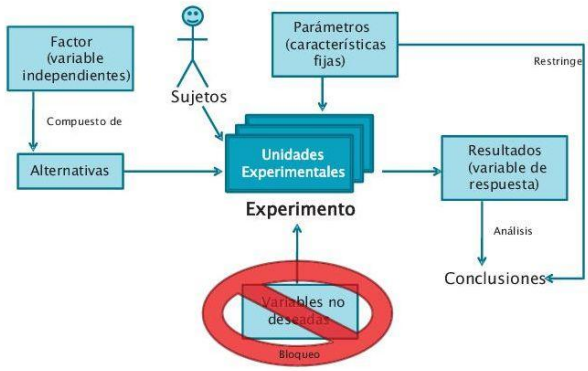



Figura 1: Componentes en un experimento de Ingeniería de Software

La aleatoriedad que se puede aplicar a un experimento también depende del tipo de diseño que se haya elegido. Por ejemplo, si se tienen dos factores A y B, cada uno con dos posibles alternativas (a1, a2, b1 y b2), las alternativas deben ser combinadas de la siguiente forma: a1b1, a1b2, a2b1, a2b2, ya que cuando se tienen dos factores se quiere observar el efecto de cada alternativa por separado y de la interacción entre ambas.

Esta combinación de alternativas es especificada por el tipo de diseño experimental que se eligió. Sin embargo, las cuatro combinaciones deben ser asignadas de forma aleatoria a los proyectos y sujetos, y es ahí en donde la aleatoriedad se aplica.

**Bloqueo:** la técnica de bloqueo se usa cuando se tienen factores que probablemente tengan efectos indeseados en las variables de respuesta y éstos efectos son conocidos y controlables.

Como se mencionaba en el ejemplo de *Efec-Técnicas* en la sección anterior, algunos verificadores podrían tener experiencia en el uso de las técnicas de verificación y otros no. Entonces, para minimizar el efecto de la experiencia, se agrupan a los participantes en dos grupos, uno con verificadores experientes y otro sin experiencia.

**Balance:** el balance es deseable ya que simplifica y fortalece el análisis estadístico de los datos, aunque no es necesario. Tomando como ejemplo el experimento de *Efec-Analgésicos* nuevamente, sería deseable que la cantidad de personas a las cuales se les administra Zolben sea igual a la cantidad de personas que se les administra Aspirina.

### 3.3. Tipos de Diseño

En el proceso del diseño experimental, primero se debe decidir (basándose en los objetivos del experimento) a qué factores y alternati-

vas estarán sujetas las unidades experimentales y qué parámetros deben ser establecidos. Luego, se debe examinar si existe la posibilidad de que algunos de los parámetros no pueda mantenerse en un valor constante, en ese caso se debe tener en cuenta cualquier variación indeseable. Finalmente, se debe elegir qué variables de respuesta serán medidas y cuáles serán los objetos y sujetos experimentales.

Teniendo establecidos los parámetros, factores, variables de bloqueo y variables de respuesta, se debe elegir el tipo de diseño experimental, en el cual se establece cuántas combinaciones de experimentos unitarios y alternativas deben haber.

Los distintos tipos de diseño experimental dependen del objetivo del experimento, del número de factores, de las alternativas de los factores y de la cantidad de variaciones indeseadas, entre otros.

Los tipos de diseño experimental se dividen en diseños de *un solo factor* y diseños de *múltiples factores*. A continuación se profundiza en los experimentos de un solo factor.

### 3.3.1. Diseño de un solo factor (*One-Factor Design*)

Para experimentos con un solo factor existen distintos tipos de diseños estándar, los principales son: los completamente aleatorios y los aleatorios con comparación por pares.

Los diseños **completamente aleatorios** son los tipos de diseño más simples, en los cuales se intenta comparar dos o más alternativas aplicadas a un determinado número de unidades experimentales, en donde cada unidad experimental se ve afectada una única vez, y por ende, por una sola alternativa. La asignación de las alternativas a los experimentos debe ser de forma aleatoria para asegurar la validez del análisis de datos.

Tomando como ejemplo *Efec-Técnicas* y suponiendo que el conjunto de programas sobre el cual se quiere conocer la efectividad de las técnicas lo componen diez programas distintos, se tendría que asignar las técnicas y los ingenieros de forma aleatoria a los programas que se vayan a verificar.

Una posible asignación aleatoria sería tener en una bolsa los nombres de todas las técnicas de verificación a aplicar, en donde la primera que se extraiga se aplique al programa $P_1$, la segunda a $P_2$ y así hasta el programa $P_{10}$. Luego de tener las duplas Programa-Técnica, efectuar la misma asignación aleatoria con los participantes: el primer participante extraído se lo asigna la dupla $(P_1, T_x)$, el segundo a la dupla $(P_2, T_y)$, y así sucesivamente.

El análisis estadístico que se puede hacer a este tipo de experimentos varía según si se aplican 2 o más alternativas para el factor.

Los diseños **aleatorios con comparación por pares** tienen como objetivo encontrar cuál es la mejor alternativa respecto de una determinada variable de respuesta. Estos tipos de diseño tienen la particularidad de que las alternativas se aplican al mismo experimento, instanciado en más de una unidad experimental.

Para el experimento de *Efec-Técnicas* no sería una buena decisión que cada ingeniero verificara 2 veces el mismo programa. En la segun-

da instancia de verificación, el ingeniero posee conocimiento tanto de los defectos del programa como de la tarea de verificar propiamente dicha (aunque sea con una técnica distinta). Por esto, para comparar las dos técnicas, ambas tienen que ser aplicadas por primera vez por ingenieros distintos, pero con similares características (ya que encontrar uno igual es imposible). La alternativa que debe aplicar cada ingeniero al programa debe ser asignada de forma aleatoria y no debe verificar un mismo programa más de una vez.

En este tipo de diseños se bloquean cierto tipo de variables que representan restricciones en la aleatoriedad que se le puede dar. Tomando como ejemplo nuevamente a *Efec-Técnicas*, si un verificador sin experiencia aplica más de una técnica durante el experimento, no sería deseable asignar al azar la técnica que cada verificador aplica en cada verificación.

Existe un efecto de aprendizaje en el cual, luego de que un verificador ejecutó una verificación, éste generó conocimiento sobre la verificación en sí, independientemente de la técnica que haya aplicado, y éste conocimiento influye significativamente en la segunda instancia de verificación que vaya a aplicar. Por tanto, la aleatoriedad en el orden de la asignación de técnicas en este ejemplo no es del todo deseable.

## 4. Proceso Experimental

Como se mencionó anteriormente, los experimentos son una técnica de investigación en la cual se quiere tener un mejor control del estudio y del entorno en el que éste se lleva a cabo.

Los experimentos son apropiados para investigar distintos aspectos de la IS, como ser: confirmar teorías, explorar relaciones, evaluar la exactitud de los modelos y validar medidas. Tienen un alto costo respecto de las otras técnicas de investigación, pero a cambio ofrecen un control total de la ejecución y son de fácil replicación.

El proceso para llevar a cabo un experimento está formado por varias fases: definición, planificación, operación, análisis e interpretación y presentación.

La primer fase es la de **definición**, en donde se define el experimento en términos del problema, objetivos y metas. La siguiente fase es la **planificación**, en la cual se determina el diseño del experimento. En la fase de **operación** se ejecuta el diseño del experimento, en donde se recolectan los datos que serán analizados posteriormente en la fase de **análisis e interpretación**. En esta última fase, conceptos estadísticos son aplicados para analizar los datos. Por último, se muestran los resultados obtenidos en la fase de **presentación**.

En la figura 2 se muestra una visión general de todo el proceso. Cada una de las fases que lo componen se detallan a continuación.

### 4.1. Definición

En la fase de Definición se determinan las bases del experimento, que se ilustra en la figura 3. Para ello se debe definir **el problema que**

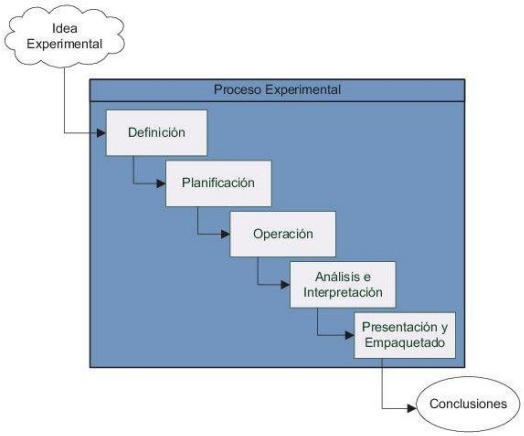



Figura 2: Visión general del Proceso Experimental

**se quiere resolver**, **propósito del experimento** y **los objetivos y metas** del mismo.


Figura 3: Fase de Definición del Experimento

Para el planteo del objetivo del experimento se debe definir *el objeto de estudio*, que es la entidad que va a ser estudiada en el experimento. Puede ser un producto, proceso, recurso u otro. También se debe establecer el *propósito*: la intención del experimento. Por ejemplo, evaluar diferentes técnicas de verificación.

Se debe definir además el *foco de calidad*, que refiere al efecto primario que esta bajo estudio, ejemplos son la efectividad y el costo de las técnicas de verificación. El propósito y el foco de calidad son las bases para las hipótesis del experimento.

Otro aspecto que debe estar presente es la *perspectiva*, que refiere al punto de vista con que los resultados obtenidos son interpretados. Por ejemplo, los resultados de la comparación de técnicas de verificación pueden verse desde la perspectiva de un experimentador, de un investigador o de un profesional. Un experimentador verá el estudio como una demostración de como una técnica de verificación puede ser

evaluada. Un investigador puede ver el estudio como una base empírica para refinar teorías sobre la verificación de software, enfocándose en los datos que apoyan o refutan estas teorías. Un profesional puede ver el estudio como una fuente de información sobre qué técnicas de verificación deberían aplicarse en la práctica.

Junto con los aspectos mencionados se debe definir el *contexto*, que es el ambiente en el que se ejecuta el experimento. En este punto se deben definir los *sujetos* que van a llevar a cabo el experimento y los *artefactos* que son utilizados en la ejecución del mismo. Se puede caracterizar el contexto de un experimento según el número de sujetos y objetos definidos en él: un solo objeto y un solo sujeto, un solo sujeto a través de muchos objetos, un solo objeto a través de un conjunto de sujetos, o un conjunto de sujetos y un conjunto de objetos.

## 4.2. Planificación

La planificación es la fase en la que se define como se va a llevar a cabo el experimento. Esta fase consta de las etapas: selección del contexto, formulación de las hipótesis, elección de las variables, selección de los sujetos, diseño del experimento, instrumentación y evaluación de la validez, que se muestran en la figura 4.

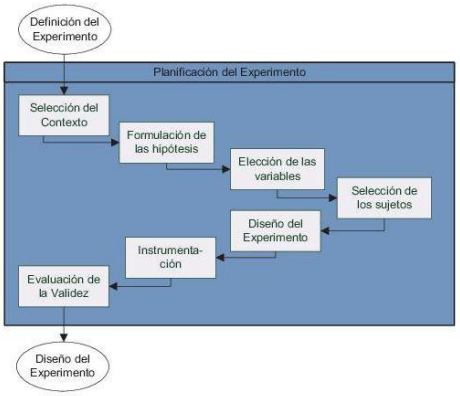



Figura 4: Fase de Planificación del Experimento

La etapa de **selección del contexto** es la etapa inicial de la planificación. En esta etapa se amplía el contexto definido en la etapa de Definición, especificando claramente las características del ambiente donde ejecuta el experimento. Se define si el experimento se va a realizar en un proyecto real (en línea, *on-line*) o en un laboratorio (fuera de línea, *off-line*), características de los sujetos y si el problema es «real» (problema existente en la industria) o «de juguete». También se debe

definir si el experimento es válido para un contexto específico o para un dominio general de la Ingeniería de Software.

Una vez que los objetivos están claramente definidos se pueden transformar en una hipótesis formal. La **formulación de las hipótesis** es una fase muy importante dentro de la etapa de planificación, ya que la verificación de la misma es la base para el análisis estadístico. En esta fase se formaliza la definición del experimento en la hipótesis.

Usualmente se definen dos hipótesis, la hipótesis nula y la hipótesis alternativa. La hipótesis nula, denotada $H_0$, asume que no hay una diferencia significativa entre las alternativas, con respecto a las variables dependientes que se están midiendo. Establece que si hay diferencias entre las observaciones realizadas, éstas son por casualidad, no producto de la alternativa aplicada. Esta hipótesis se asume verdadera hasta que los datos demuestren lo contrario, por lo que el foco del experimento está puesto en rechazarla. Un ejemplo de hipótesis nula es: «No hay diferencia en la cantidad de defectos encontrados por las técnicas de verificación».

En cambio la hipótesis alternativa, denotada $H_1$, afirma que existe una diferencia significativa entre las alternativas con respecto a las variables dependientes. Establece que las diferencias encontradas son producto de la aplicación de las alternativas. Ésta es la hipótesis a probar, para esto se debe determinar que los datos obtenidos son lo suficientemente convincentes para desechar la hipótesis nula y aceptar la hipótesis alternativa. Un ejemplo de hipótesis alternativa es, si se están comparando dos técnicas de verificación, decir que una encuentra más defectos que la otra. En caso de haber más de una hipótesis alternativa se denotan secuencialmente: $H_1$, $H_2$, $H_3$, ..., $H_n$.

Una vez definida la hipótesis, se debe identificar qué variables afectan a la/s alternativa/s. Luego de identificadas las variables se debe decidir el control a ejercer sobre las mismas.

La **selección de las variables** dependientes como la de las independientes están relacionadas, por lo que en muchos casos se realizan en simultáneo. Seleccionar estas variables es una tarea muy compleja, que en ocasiones implica conocer muy bien el dominio. Es importante definir las variables independientes y analizar sus características, para así investigar y controlar los efectos que ejercen sobre las variables dependientes. Se deben identificar las variables independientes que se pueden controlar y las que no. Además, se deben identificar las variables dependientes, mediante las cuales se mide el efecto de las alternativas. Generalmente hay sólo una variable dependiente y se deriva de la hipótesis.

Otro aspecto importante al llevar a cabo un experimento es la **selección de los sujetos**. Para poder generalizar los resultados al resto de la población, la selección debe ser una muestra representativa de la misma. Cuanto más grande es la muestra, menor es el error al generalizar los datos. Existen dos tipos de muestras que se pueden seleccionar: la probabilística, donde se conoce la probabilidad de seleccionar cada sujeto; y la no-probabilística, donde esta probabilidad es desconocida.

Luego de definir el contexto, formalizar las hipótesis, y seleccionar las variables y los sujetos, se debe **diseñar el experimento**. Es muy importante planear y diseñar cuidadosamente el experimento, para que los datos obtenidos puedan ser interpretados mediante la aplicación de métodos de análisis estadísticos.

Para comenzar a diseñar un experimento se debe elegir el diseño adecuado. Se debe planificar y diseñar el conjunto de las combinaciones de alternativas, sujetos y objetos, que conforman los experimentos unitarios. Se describe cómo estos experimentos unitarios deben ser organizados y ejecutados.

La elección del diseño del experimento afecta el análisis estadístico y viceversa, por lo que al elegir el diseño del experimento se debe tener en cuenta qué análisis estadístico es el mejor para rechazar la hipótesis nula y aceptar la alternativa.

Luego de diseñar el experimento y antes de la ejecución es necesario contar con todo lo necesario para la correcta ejecución del mismo. La **instrumentación** involucra, de ser necesario, capacitación a los sujetos, preparación de los artefactos, construcción de guías, descripción de procesos, planillas y herramientas. También implica configurar el hardware, mecanismos de consultas y experiencias piloto, entre otros. La finalidad de esta fase es proveer todo lo necesario para la realización y monitorización del experimento.

### 4.3. Evaluación de la Validez

Una pregunta fundamental antes de pasar a ejecutar el experimento es cuán válidos serían los resultados. Existen cuatro categorías de amenazas a la validez: validez de la conclusión, validez interna, validez del constructo y validez externa.

Las amenazas que afectan la **validez de la conclusión** refieren a las conclusiones estadísticas. Amenazas que afecten la capacidad de determinar si existe una relación entre la alternativa y el resultado, y si las conclusiones obtenidas al respecto son válidas. Ejemplos de estas son la elección de los métodos estadísticos, y la elección del tamaño de la muestra, entre otros.

Las amenazas que influyen en la **validez interna** son aquellas referidas a observar relaciones entre la alternativa y el resultado que sean producto de la casualidad y no del resultado de la aplicación de un factor. Esta «casualidad» es provocada por elementos desconocidos que influyen sobre los resultados sin el conocimiento de los investigadores. Es decir, la validez interna se basa en asegurar que la alternativa en cuestión produce los resultados observados.

La **validez del constructo** indica cómo una medición se relaciona con otras de acuerdo con la teoría o hipótesis que concierne a los conceptos que se están midiendo. Un ejemplo se puede observar al momento de seleccionar los sujetos en un experimento. Si se utiliza como medida de la experiencia del sujeto el número de cursos que tiene aprobados en la universidad, no se está utilizando una buena medida de la experiencia. En cambio, una buena medida puede ser utilizar la

cantidad de años de experiencia en la industria o una combinación de ambas cosas.

La **validez externa** está relacionada con la habilidad para generalizar los resultados. Se ve afectada por el diseño del experimento. Los tres riesgos principales que tiene la validez externa son: tener los participantes equivocados como sujetos, ejecutar el experimento en un ambiente erróneo y realizar el experimento en un momento que afecte los resultados.

## 4.4. Operación

Luego de diseñar y planificar el experimento, éste debe ser ejecutado para recolectar los datos que se quieren analizar. La operación del experimento consiste en tres etapas: preparación, ejecución y la validación de los datos, que se muestran en la figura 5.

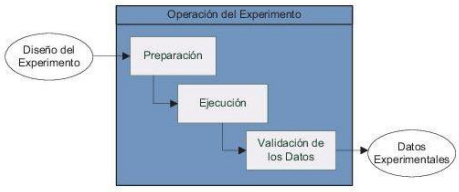



Figura 5: Fase de Operación del Experimento

En la etapa de preparación se seleccionan los sujetos y se preparan los artefactos a ser utilizados.

Es muy importante que los sujetos estén motivados y dispuestos a realizar las actividades que les sean asignadas, ya sea que tengan conocimiento o no de su participación en el experimento. Se debe intentar obtener consentimiento por parte de los participantes, que deben estar de acuerdo con los objetivos de la investigación. Los resultados obtenidos pueden volverse inválidos si los sujetos al momento que deciden participar no saben lo que tienen que hacer o tienen un concepto erróneo.

Es importante considerar la sensibilidad de los resultados que se obtienen de los sujetos, por ejemplo: es importante asegurar a los participantes que los resultados obtenidos sobre su rendimiento se mantienen en secreto y no se usarán para perjudicarlos en ningún sentido. Se debe tener en cuenta también los incentivos, ya que ayudan a motivar a los sujetos, pero se corre el riesgo de que participen sólo por el incentivo, lo que puede ser perjudicial para el experimento. En caso de no tener otra alternativa que no sea engañar a los sujetos, se debe procurar explicar y revelar el engaño lo más temprano posible.

Como se vio en la instrumentación, para que los sujetos comiencen la ejecución es necesario tener prontos todos los instrumentos, formularios, herramientas, guías y otros artefactos que sean necesarios para la ejecución del experimento. Muchas veces se debe preparar un

conjunto de instrumentos especial para cada sujeto y otras se utiliza el mismo conjunto de artefactos para todos los sujetos.

Existen muchas formas distintas de ejecutar los experimentos, la duración varía desde días hasta años.

Los datos pueden ser recolectados de las siguientes formas:

- Manualmente mediante el llenado de formularios por parte de los sujetos.
- Manualmente soportado por herramientas.
- Mediante entrevistas.
- Automáticamente por herramientas.

La primera es la forma más común y no requiere mucho esfuerzo por parte del experimentador. Tanto en los formularios como en los métodos soportados por herramientas no es posible identificar inconsistencias o defectos hasta que no se recolecte la información, o hasta que los sujetos los descubran. En las entrevistas, el contacto con los sujetos es mucho mayor permitiendo una mejor comunicación con ellos durante la recolección de datos. Éste método es el que requiere mas esfuerzo por parte del investigador.

Un aspecto muy importante a la hora de ejecutar los experimentos es el ambiente de ejecución, tanto si el experimento se realiza dentro de un proyecto de desarrollo común o si se crea un ambiente ficticio para su ejecución. En el primer caso el experimento no debería afectar el proyecto más de lo necesario, ya que la razón de realizar el experimento dentro de un proyecto es ver los efectos de las alternativas en el ambiente del proyecto. Si el experimento cambia demasiado el ambiente del proyecto, éstos efectos se perderían.

Cuando se obtienen los datos, se debe chequear que fueron recolectados correctamente y que son razonables. Algunas fuentes de error son que los sujetos llenen mal sus planillas, o no recolecten los datos seriamente, lo que hace que se descarten datos. Es importante revisar que los sujetos hagan un trabajo serio y responsable y que apliquen las alternativas en el orden correcto, en otras palabras: que el experimento sea ejecutado en la forma en que fue planificado. De lo contrario los resultados podrían ser inválidos.

### 4.5. Análisis e Interpretación

Luego de que finaliza la ejecución del experimento y se cuenta con los datos recolectados, comienza la fase de análisis de los mismos conforme a los objetivos planteados.

Un aspecto importante a considerar en el análisis de los datos es la **escala de medida**. La escala de medida utilizada para recolectar los datos restringe el tipo de cálculos estadísticos que se pueden realizar. Una medida es un mapeo de un atributo de una entidad a un valor de medida, por lo general un valor numérico. Las entidades son objetos que se observan en la realidad, por ejemplo, una técnica de verificación.

El propósito de mapear los atributos en un valor de medida es caracterizar y manipular los atributos formalmente. La medida seleccionada debe ser válida, por tanto, no debe violar ninguna propiedad necesaria del atributo que mide, y debe ser una caracterización matemática adecuada del atributo.

El mapeo de un atributo a un valor de medida puede realizarse de varias formas. Cada tipo de mapeo posible de un atributo se conoce como escala. Los tipos más comunes de escala son:

- Escala Nominal.- Es la menos poderosa de las escalas. Solo mapea el atributo de la entidad en un nombre o símbolo. El mapeo puede verse como una clasificación de las entidades acorde al atributo. Ejemplos de escala nominal son clasificaciones, etiquetados, entre otras.
- Escala Ordinal.- La escala ordinal categoriza las entidades según un criterio de ordenación. Es más poderosa que la escala nominal. Ejemplos de criterios de ordenación son «mayor que», «mejor que» y «más complejo». Ejemplos de escala nominal son grados, complejidad del software, entre otras.
- Escala de intervalo.- La escala de intervalo se utiliza cuando la diferencia entre dos medidas es significativa, pero no el valor en si mismo. Este tipo de escala ordena los valores de la misma forma que la escala ordinal, pero existe la noción de «distancia relativa» entre dos entidades. Esta escala es más poderosa que la ordinal. Ejemplos de escala de intervalo son la temperatura medida en Celsius o Fahrenheit.
- Escala ratio (cociente de dos números).- Si existe un valor cero significativo y la división entre dos medidas es significativa, se puede utilizar una escala ratio. Ejemplos de escala ratio son distancia, temperatura medida en Kelvin, etc.

Después de obtener los datos es necesario interpretarlos para llegar a conclusiones válidas. La interpretación se realiza en tres etapas: caracterizar el conjunto de datos usando estadística descriptiva, reducción del conjunto de datos y realización de las pruebas de hipótesis que se ilustran en la figura 6.


Figura 6: Fase de Análisis e Interpretación de los Datos del Experimento

### 4.5.1. Estadística Descriptiva

La **estadística descriptiva** se utiliza antes de la prueba de hipótesis, para entender mejor la naturaleza de los datos y para identificar datos falsos o anormales. Los aspectos principales que se examinan son: la tendencia central, la dispersión y la dependencia. A continuación se presentan las medidas más comunes de cada uno de estos aspectos. Para ello se asume que existen $x_1 \dots x_n$ muestras.

Las **medidas de tendencia central** indican «el medio» de un conjunto de datos. Entre las medidas más comunes se encuentran: la media aritmética, la mediana y la moda.

La *media aritmética* se conoce como el promedio, y se calcula sumando todas las muestras y dividiendo el total por el número de muestras:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

La *media*, denotada $\bar{x}$, resume en un valor las características de una variable teniendo en cuenta a todos los casos. Es significativa para las escalas de intervalo y ratio.

La *mediana*, denotada $\tilde{x}$, representa el valor medio de un conjunto de datos, tal que el número de muestras que son mayores que la mediana es el mismo que el número de muestras que son menores que la mediana. Se calcula ordenando las muestras en orden ascendente o descendente, y seleccionando la observación del medio. Este cálculo está bien definido si $n$ es impar. Si $n$ es par, la mediana se define como la media aritmética de los dos valores medios. Esta medida es significativa para las escalas ordinal, de intervalo y ratio.

La *moda* representa la muestra más común. Se calcula contando el número de muestras para cada valor único y seleccionando el valor con más cantidad. La moda esta bien definida si hay solo un valor más común que los otros. Si este no es el caso, se calcula como la mediana de las muestras más comunes. La moda es significativa para las escalas nominal, ordinal, de intervalo y ratio.

La media aritmética y la mediana son iguales si la distribución de las muestras es simétrica. Si la distribución es simétrica y tiene un único valor máximo, las tres medidas son iguales.

Las medidas de tendencia central no proveen información sobre la dispersión del conjunto de datos. Cuanto mayor es la dispersión, más variables son las muestras, cuanto menor es la dispersión, más homogéneas a la media son las muestras.

Las **medidas de dispersión** miden el nivel de desviación de la tendencia central, o sea, que tan diseminados o concentrados están los datos respecto al valor central. Entre las principales medidas de dispersión están: la varianza, la desviación estándar, el rango y el coeficiente de variación.

La *varianza* ($s^2$) que presenta una distribución respecto de su media se calcula como la media de las desviaciones de las muestras respec-

to a la media aritmética. Dado que la suma de las desviaciones es siempre cero, se toman las desviaciones al cuadrado:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2}$$

Se divide por $n-1$ y no por $n$, porque dividir por $n-1$ provee a la varianza de propiedades convenientes. La varianza es significativa para las escalas de intervalo y ratio.

La *desviación estándar*, denotada $s$, se de
ne como la raíz cuadrada de la varianza:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3}$$

A menudo esta medida se prefiere sobre la varianza porque tiene las mismas dimensiones (unidad de medida) que los valores de las muestras. En cambio, la varianza se mide en unidades cuadráticas. La desviación estándar es significativa para las escalas de intervalo y ratio.

La dispersión también se puede expresar como un porcentaje de la media. Este valor se llama *coeficiente de variación*, y se calcula como:

$$100 \cdot \frac{s}{\bar{x}} \tag{4}$$

Esta medida no tiene dimensión y es significativa para la escala ratio. Permite comparar la dispersión o variabilidad de dos o más grupos.

El **rango** de un conjunto de datos es la distancia entre el valor máximo y el mínimo:

$$range = x_{max} - x_{min} \tag{5}$$

Es una medida significativa para las escalas de intervalo y ratio. Cuando el conjunto de datos consiste en muestras relacionadas de a pares (xi; yi) de dos variables, X e Y, puede ser interesante examinar la dependencia entre estas variables. Las principales medidas de dependencia son: regresión lineal, covarianza y el coeficiente de correlación lineal.

### 4.5.2. Reducción del Conjunto de Datos

Para las pruebas de hipótesis se utilizan métodos estadísticos. El resultado de aplicar estos métodos depende de la calidad de los datos. Si los datos no representan lo que se cree, las conclusiones que se derivan de los resultados de los métodos son incorrectas. Errores en el conjunto de datos pueden ocurrir por un error sistemático, o por lo que se conoce en estadística con el nombre de outlier. Un outlier es un dato mucho más grande o mucho más chico de lo que se puede esperar observando el resto de los datos.

Las estadísticas descriptivas se ven fuertemente influenciadas por aquellas observaciones que su valor dista significativamente del resto

de los valores recolectados. Estas observaciones llevan el nombre de *outliers*.

Los *outliers* influencian las medidas de dispersión, aumentando la variabilidad de lo que se está midiendo. En algunos casos se realiza un análisis acerca de estos valores que difieren mucho de la media y se decide quitarlos de los datos a analizar porque no son representativos de la población, ya que fueron causados por algún tipo de anomalía: errores de medición, variaciones no deseadas en las características de los sujetos, entre otras.

Quitar *outliers* requiere de un análisis pormenorizado, por quitar outliers se demoró en detectar el agujero de la capa de ozono. [1]

Una vez identificado un outlier se debe identificar su origen para decidir qué hacer con él. Si se debe a un evento raro o extraño que no volverá a ocurrir, el punto puede ser excluido. Si se debe a un evento extraño que puede volver a ocurrir, no es aconsejable excluir el valor del análisis, pues tiene información relevante. Si se debe a una variable que no fue considerada, debería ser considerado para basar los cálculos y modelos también en esta variable.

### 4.5.3. Pruebas de Hipótesis

El objetivo de la **prueba de hipótesis** es ver si es posible rechazar cierta hipótesis nula $H_0$. Si la hipótesis nula no es rechazada, no se puede decir nada sobre los resultados. En cambio, si es rechazada, se puede declarar que la hipótesis es falsa con una significancia dada ($\alpha$). Este nivel de significancia también es denominado nivel de riesgo o probabilidad de error, ya que se corre el riesgo de rechazar la hipótesis nula cuando en realidad es verdadera. Este nivel está bajo el control del experimentador.

Para probar $H_0$ se define una unidad de prueba $t$ y un área crítica $C$, la cual es parte del área sobre la que varía $t$. A partir de estas definiciones se formula la prueba de significancia de la siguiente forma:

- Si $t \in C$, rechazar $H_0$
- Si $t \notin C$, no rechazar $H_0$

Por ejemplo, un experimentador observa la cantidad de defectos detectados por LOC de una técnica de verificación desconocida bajo determinadas condiciones, y quiere probar que no es la técnica B, de la cual sabe que en las mismas condiciones (programa, verificador, etc.) detecta 1 defecto cada 20 LOC. El experimentador sabe que también pueden haber otras técnicas que detecten 1 defecto cada 20 LOC. A partir de esto se define la hipótesis nula: "$H_0$: La técnica observada es la B". En este ejemplo, la unidad de prueba $t$ es cada cuantos LOC se detecta un defecto y el área crítica es

[1] En 1985 los científicos británicos anunciaron un agujero en la capa de ozono sobre el polo sur. El reporte fue descartado ya que observaciones más completas, obtenidas por instrumentos satelitales, no mostraban nada inusual. Luego, un análisis más exhaustivo reveló que las lecturas de ozono en el polo sur eran tan bajas que el programa que las analizaba las había suprimido automáticamente como outliers en forma equivocada.

$C = \{1, 2, 3, \ldots, 19, 21, 22, \ldots\}$. La prueba de significancia es: si $t \leq 19$ o $t \geq 21$, rechazar $H_0$, de lo contrario no rechazar $H_0$.

Si se observa que $t = 20$, la hipótesis no puede ser rechazada ni se pueden derivar conclusiones, pues pueden haber otras técnicas que detecten un defecto cada 20 LOC.

El área crítica, $C$, puede tener distintas formas, lo más común es que tenga forma de intervalo, por ejemplo: $t \leq a$ o $t \geq b$. Si $C$ consiste en uno de estos intervalos es unilateral. Si consiste de dos intervalos ($t \leq a$, $t \geq b$, donde $a < b$), es bilateral.

Hay varios métodos estadísticos, de aquí en adelante denotados *tests*, que pueden utilizarse para evaluar los resultados de un experimento, más específicamente para determinar si se rechaza la hipótesis nula. Cuando se lleva a cabo un *test* es posible calcular el menor valor de significancia posible (denotado $p$-valor) con el cual es posible rechazar la hipótesis nula. Se rechaza la hipótesis nula si el p-valor asociado al resultado observado es menor o igual que el nivel de significancia establecido.

Las siguientes son tres probabilidades importantes para la prueba de hipótesis:

- $\alpha$ = $P$(cometer el error tipo I) = $P$(rechazar $H_0$ | $H_0$ es verdadera). Es la probabilidad de rechazar $H_0$ cuando es verdadera.
- $\beta$ = $P$(cometer el error tipo II) = $P$(no rechazar $H_0$ | $H_0$ es falsa). Es la probabilidad de no rechazar $H_0$ cuando es falsa.
- Poder = $1 - \beta$ = $P$(rechazar $H_0$ | $H_0$ es falsa). El poder de prueba es la probabilidad de rechazar $H_0$ cuando es falsa.

El experimentador debería elegir un test con un poder de prueba tan alto como sea posible. Hay varios factores que afectan el poder de un test. Primero, el test en sí mismo puede ser más o menos efectivo. Segundo, la cantidad de muestras: mayor cantidad de muestras equivale a un poder de prueba más alto. Otro aspecto es la selección de una hipótesis alternativa unilaterial o bilateral. Una hipótesis unilateral da un poder mayor que una bilateral.

La probabilidad de cometer un error tipo I se puede controlar y reducir. Si la probabilidad es muy pequeña, sólo se rechazará la hipótesis nula si se obtiene evidencia muy contundente en contra de esta hipótesis. La probabilidad máxima de cometer un error tipo I se conoce como la significancia de la prueba ($\alpha$).

Los valores de uso más común para la significancia de una prueba son 0.01, 0.05 y 0.10. La significancia es en ocasiones presentada como un porcentaje, tal como 1%, 5% o 10%. Esto quiere decir que el experimentador está dispuesto a permitir una probabilidad de 0.01, 0.05, o 0.10 de rechazar la hipótesis nula cuando es cierta, o sea, de cometer un error tipo I.

El valor de la significancia es seleccionado antes de comenzar a hacer el experimento en una de varias formas.

El valor de $\alpha$ puede estar establecido en el área de investigación, por ejemplo: se puede obtener de artículos que se publican en revistas

científicas. Otra forma de seleccionarlo es que sencillamente sea impuesto por la persona o compañía para la cual se trabaja. Finalmente, puede ser seleccionado tomando en cuenta el costo de cometer un error tipo I. Mientras más alto el costo, más pequeña debe ser la probabilidad $\alpha$ de cometer un error tipo I. El valor usual de $\alpha$ en las ciencias naturales y sociales es de 0.05. En Ingeniería de Software, el valor de $\alpha$ aún no se encuentra establecido.

Existen dos tipos de tests: paramétricos y no paramétricos. Los **tests paramétricos** están basados en un modelo que involucra una distribución específica. En la mayoría de los casos, se asume que algunos de los parámetros involucrados en un test paramétrico están normalmente distribuidos. Los tests paramétricos también requieren que los parámetros puedan ser medidos al menos en una *escala de intervalo*. Si los parámetros no pueden medirse en al menos una escala de intervalo, generalmente no se puede utilizar un test paramétrico. En este caso hay un amplio rango de tests no paramétricos disponible.

Los **tests no paramétricos** no asumen lo mismo respecto a la distribución de los parámetros, son más generales que los paramétricos. Un test no paramétrico se puede utilizar en vez de un test paramétrico, pero el caso inverso no siempre puede darse.

En la elección entre un test paramétrico y un test no paramétrico hay dos aspectos a considerar:

- Aplicabilidad.- Es importante que las suposiciones en cuanto a las distribuciones de parámetros y las que conciernen a las escalas sean realistas.
- Poder.- El poder de los tests paramétricos es generalmente mayor que el de los tests no paramétricos. Por lo tanto, los test paramétricos requieren menos datos (experimentos más pequeños), que los tests no paramétricos, siempre que sean aplicables.

Aunque es un riesgo utilizar tests paramétricos cuando no se cuenta con las condiciones requeridas, en algunos casos vale la pena tomar el riesgo. Algunas simulaciones han mostrado que los tests paramétricos son bastante robustos a las desviaciones de las pre-condiciones (escala de intervalo), mientras las desviaciones no sean demasiado grandes.

En el caso de las pruebas paramétricas, se exige que la distribución de la muestra se aproxime a una normal. Para poder utilizar aproximación normal se requiere un tamaño mínimo de la muestra, dependiendo del $p(value)$ que se requiera (11). En el cuadro 1 se muestran los tamaños mínimos de muestra para los distintos $p(value)$.

Los test paramétricos más usados en experimentos de Ingeniería de Software son:

- ANOVA (*ANalysis Of VAriance*) (18).
- ANOM (*ANalysis Of Means*) (6).

Ambos tests (ANOVA y ANOM), pueden utilizarse para diseños de un solo factor con múltiples alternativas. En ambos test la hipótesis nula

| p(value) | Tamaño mínimo de muestra |
|---|---|
| 0.50 | n = 30 |
| 0.40 ó 0.60 | n = 50 |
| 0.30 ó 0.70 | n = 80 |
| 0.20 ó 0.80 | n = 200 |
| 0.10 ó 0.90 | n = 600 |

Cuadro 1: Estadísticas descriptivas de la Efectividad

refiere a la igualdad de las medias (como es habitual en los test paramétricos):

$H_0 : \bar{x}_1 = \bar{x}_2 = \ldots = \bar{x}_I$

En ANOVA, la variación en la respuesta se divide en la variación entre los diferentes niveles del factor (los diferentes tratamientos) y la variación entre individuos dentro de cada nivel. El objetivo principal del ANOVA es contrastar si existen diferencias entre las diferentes medias de los niveles de las variables (factores).

En el caso de ANOM, este test no solamente responde a la pregunta de si hay o no diferencias entre las alternativas, sino que cuando hay diferencias, también dice cuáles alternativas son mejores y cuáles peores.

Los test no paramétricos más usado son:

- Kruskal Wallis.
- Mann-Whitney.

En el caso de los test no paramétricos, la hipótesis nula refiere a la igualdad de las medianas:

$H_0 : \tilde{x}_1 = \tilde{x}_2 = \ldots = \tilde{x}_I$

Rechazar $H_0$ significa que existe evidencia estadística como para afirmar de que hay diferencias entre las alternativas. En el caso de que hubiera más de dos alternativas, para conocer cuál es la alternativa que difiere es necesario comparar las alternativas de a dos.

En el caso de Kruskal Wallis, a pesar de no requerir una distribución normal para las muestras, sus resultados se pueden ver afectados por lo que se le llama «heterocedasticidad» de los datos. Cuando una muestra presenta datos heterocedásticos (no presentan homocedasticidad) el test de Kruskal Wallis podría dar un resultado no significativo (no rechazando $H_0$), aunque haya una diferencia real entre las muestras (debería rechazar $H_0$).

Para probar la homocedasticidad de los datos se suele utilizar el test de Levene. Las hipótesis del test de Levene son:

- $H_0 : \sigma_1 = \sigma_2 = \ldots = \sigma_k$ donde $\sigma_a$ es la varianza de la muestra a.
- $H1 : \sigma_i \neq \sigma_j = \ldots = \sigma_k$ para al menos un par de muestras $(i, j)$, donde $\sigma_a$ es la varianza de la muestra a.

Para poder aplicar ANOVA, y en algunos casos Kruskal-Wallis, es necesario que el test de Levene no sea significativo (no se rechaza $H_0$), o sea, que las varianzas de las muestras sean similares o iguales. Esto prueba la homocedasticidad de los datos.

Una vez que se prueba que al menos dos de las $k$ muestras provienen de poblaciones distintas (datos heterocedásticos) se puede aplicar, entre otros, el test de Mann-Whitney para comparar las muestras dos a dos.

Si se presume que una alternativa puede ser mejor o peor que el resto, esto quiere decir que hay un «ordenamiento» entre ellas, lo aconsejable es realizar un test de ordenamiento. Algunos test de ordenamiento son:

- Jonckheere-Terpstra Test. (5)
- Test para alternativas ordenadas L. (5)

Para los test de ordenamiento, las hipótesis que se plantean son las siguientes:

$H_0 : \tilde{x}_1 = \tilde{x}_2 = \ldots = \tilde{x}_I$

$H_1 : \tilde{x}_1 \leq \tilde{x}_2 \leq \ldots \leq \tilde{x}_I$ (con al menos una desigualdad estricta)

### 4.6. Presentación y Empaquetado

En la presentación y el empaquetado de un experimento es esencial no olvidar aspectos e información necesaria para que otros puedan replicar o tomar ventaja del experimento y del conocimiento ganado durante su ejecución.

El esquema de reporte de un experimento generalmente cuenta con los siguientes títulos: Introducción, Definición del Problema, Planificación del Experimento, Operación del Experimento, Análisis de Datos, Interpretación de los Resultados, Discusión y Conclusiones, y Apéndice.

En la *Introducción* se realiza una introducción al área y los objetivos de la investigación. En la *Definición del Problema* se describe en mayor profundidad el trasfondo de la investigación, incluyendo las razones para realizarla. En la *Planificación del Experimento* se detalla el contexto del experimento incluyendo las hipótesis, que se derivan de la definición del problema, las variables que se deben medir (tanto independientes como dependientes), la estrategia de medida y análisis de datos, los sujetos que participaran de la investigación y las amenazas a la validez.

En la *Operación del Experimento* se describe como preparar la ejecución del mismo, incluyendo aspectos que permitan facilitar la replicación y descripciones que indiquen cómo se llevaron a cabo las actividades. Debe incluirse la preparación de los sujetos, cómo se recolectaron los datos y cómo se realizó la ejecución.

En el *Análisis de Datos* se describen los cálculos y los modelos de análisis específicos utilizados. Se debe incluir información, como por ejemplo, tamaño de la muestra, niveles de significancia y métodos estadísticos utilizados, para que el lector conozca los pre-requisitos para el análisis. En la *Interpretación de los Resultados* se rechaza la hipótesis

nula o se concluye que no puede ser rechazada. Aquí se resume cómo utilizar los datos obtenidos en el experimento. La interpretación debe realizarse haciendo referencia a la validez. También se deben describir los factores que puedan tener un impacto sobre los resultados.

Finalmente, en *Discusión y Conclusiones* se presentan las conclusiones y los hallazgos como un resumen de todo el experimento, junto con los resultados, problemas y desviaciones respecto al plan. También se incluyen ideas sobre trabajos a futuro. Los resultados deberían ser comparados con los obtenidos por trabajos anteriores, de manera de identificar similitudes y diferencias. La información que no es vital para la presentación se incluye en el Apéndice. Esto puede ser, por ejemplo, los datos recavados y más información sobre sujetos y objetos. Si la intención es generar un paquete de laboratorio, el material utilizado en el experimento puede ser proveído en el apéndice.

## 5. Estudio de casos

Un estudio de casos en ingeniería de software es una investigación empírica que se basa en múltiples fuentes de evidencia para investigar un fenómeno de ingeniera de software contemporánea dentro de su contexto real, especialmente cuando los límites entre el fenómeno y su contexto no son claramente evidentes.

La conducción de un estudio de casos consiste de las siguientes fases: diseño, preparación, recolección, análisis y reporte. En esta sección se presentan los aspectos a considerar durante el diseño de estudio de casos y diversas técnicas de recolección de datos.

### 5.1. Diseño

Al diseñar un estudio de casos se deben considerar los siguientes elementos:

**Razón fundamental del estudio:**

El investigador debe tener clara la razón de llevar a cabo el estudio. Para investigaciones académicas una razón típica es generar una contribución, por ejemplo generar una nueva teoría o hipótesis. En la industria las razones pueden ser brindar una mejora a una organización o un proyecto.

**Objetivo del estudio:**

El objetivo general del estudio es una declaración de lo que el investigador espera alcanzar como resultado de la realización de ese estudio. El objetivo se refina en un conjunto de preguntas de investigación y éstas se responden mediante el análisis de los datos.

**El caso y la unidad de Análisis:**

Los investigadores hacen una distinción entre el estudio de casos y la unidad o unidades de análisis dentro del caso. Las unidades de análisis permiten definir qué es el caso. Cuando el estudio de caso se realiza sobre un objeto concreto, por ejemplo una persona (pacientes, líderes, estudiantes...), la unidad de análisis está muy clara porque es el propio objeto investigado. En cambio, en estudio de casos sobre

fenómenos o acontecimientos más complejos de definir, es necesario considerar una o varias unidades de análisis que permitan dar un paso más en la concreción de la investigación. Las unidades de análisis permiten definir los límites del caso para diferenciarlos de su contexto.

En los estudios de casos, el caso y la unidad de análisis deben ser seleccionadas intencionalmente. Esto se contrasta con las encuestas y experimentos, en donde los sujetos son una muestra de la población.

**Tipos de diseños:**

Yin propone una tipología que establece cuatro formatos básicos de estudio de casos, que resultan de la combinación de dos características: la primera es si el estudio incluye un único caso o si contiene más de uno (múltiple) (19). La segunda característica es si su análisis tiene una sola unidad de análisis, es decir, un sentido holístico, o si se divide en diversas unidades de análisis parciales (encapsulado). Tanto el estudio de un sólo caso, como el estudio de casos múltiples, puede ser, a su vez, holístico o encapsulado, resultando los cuatro tipos planteados por el autor. En el cuadro 2 se presenta la tipología mencionada.

| | **Diseño de caso único** | **Diseño de múltiples casos** |
|---|---|---|
| Holístico | Tipo 1 | Tipo 3 |
| Encapsulado | Tipo 2 | Tipo 4 |

Cuadro 2: Tipos básicos de diseño para estudios de casos

Holístico o encapsulado: La unidad de análisis puede ser un individuo, un grupo, una compañía, un país, etc. La unidad de análisis ayuda a definir el alcance del caso. El caso es con frecuencia un proceso, una institución, o un evento no tan bien definido como un individuo. La definición de la unidad de análisis está vinculada con la forma en que se presentaron las primeras preguntas de la investigación. Si solo se busca examinar la naturaleza general de una empresa o problema, se utiliza un enfoque holístico. Se procede así cuando no se logra identificar sub-unidades o sectores o cuando la naturaleza del estudio es holística. Si se examinan una o varias sub-unidades de una organización o programa, se utiliza un enfoque encapsulado.

Diseños simples o múltiples: Los diseños simples se utilizan cuando, de modo análogo a un experimento, un caso permite probar una nueva teoría, o establece las circunstancias en que valdrían ciertas proposiciones. También un diseño simple se aplica en casos únicos o extremos, o un caso "revelatorio", en el que se presenta a los ojos del investigador un fenómeno antes no estudiado. Los diseños múltiples, por otra parte, tienen la ventaja de que su evidencia es más convincente y el estudio resulta más robusto. Sus desventajas consisten en que no permiten tratar con el caso revelatorio, o raro, o crítico, de los casos simples y, además, requiere más recursos. El tema del número de casos que conviene analizar es debatido. Algunos autores se inclinan por el estudio de un solo caso y citan para avalar su posición ejemplos de casos clásicos, como Street Corner Society, que mostrarían la

importancia de concentrarse en el estudio a fondo de un único caso. Eisenhardt (2) sostienen en cambio que es posible obtener recursos para casos múltiples; de hecho, hay ejemplos de casos múltiples ya clásicos. Smith (8) relata que, en su experiencia, a medida que cada caso progresa a través de entrevistas los datos se van adecuando a un patrón, "en otras palabras, una teoría (va) emergiendo" y los datos sucesivos se hacen predecibles a partir de la teoría. Cuando se verifica este fenómeno, al cual se suele llamar saturación, puede decirse que el número de casos considerado es suficiente.

**Pregunta de investigación:**

Las preguntas de investigación son afirmaciones sobre el conocimiento que se busca, o se espera descubrir durante el estudio de casos.

El uso de estudio de casos es adecuado cuando la pregunta de investigación es un ¿cómo? o un ¿por qué?, cuando el investigador tiene poco control sobre los eventos y cuando el énfasis de la investigación está puesto sobre eventos contemporáneos dentro de su contexto en la vida real.

Una pregunta de investigación puede estar relacionada con una hipótesis, que es una supuesta explicación de un aspecto del fenómeno de estudio.

**Proposiciones e Hipótesis:**

Las proposiciones son predicciones acerca del mundo que pueden deducirse lógicamente de la teoría, nos orientan sobre los objetos que deben ser examinados en el estudio; desmenuzan las preguntas de tipo "cómo" y "por qué" para determinar qué debemos estudiar. Las hipótesis se generan a partir de las preposiciones y deben ser empíricamente testeables.

**Control y aseguramiento de la calidad, cuestiones legales, éticas y profesionales:**

Existen diversos métodos para asegurar la calidad del diseño del estudio de casos, dentro de los cuales se encuentran, que el diseño del estudio de casos sea revisado por personas ajenas al proyecto y/o conducir un estudio pilota para evaluar el diseño contruido.

Al comenzar la preparación del estudio de casos el investigador debe comprometerse a proteger a los candidatos del estudio. Como parte de la protección debe comprometerse a realizar un estudio de casos que:

- tenga el consentimiento de todos los candidatos que formaran parte del estudio, alertando del objetivo y que estos sean voluntarios del mismo.
- proteja al candidato de cualquier engaño en el estudio
- proteja la privacidad y confidencialidad de los datos de los candidatos (nombres, edades)
- tome precauciones sobre candidatos vulnerables (niños)

## 5.2. Recolección de datos

En esta sección se detallan diversas tecnicas de recolección de datos. Las técnicas de recolección de datos se pueden dividir en tres grados según Lethbridge (12):

- Primer grado: Estos son métodos directos, donde el investigador está en contacto directo con el entrevistado y recopila los datos en tiempo real. Ejemplos de estos métodos son entrevistas, grupos focales, encuestas Delphi y observaciones.
- Segundo grado: Estos son métodos indirectos donde el investigador recoge directamente los datos sin tener que interactuar con el entrevistado. Ejemplos de este método son herramientas automáticas que monitorean y observan.
- Tercer grado: estos son los métodos en donde el investigador analiza el trabajo que ya está disponible. Este enfoque se utiliza cuando se analizan especificaciones, documentos, informes, etc.

La técnica de primer grado tiende a ser más cara de aplicar que la segunda o tercera técnica debido a que se requiere un esfuerzo significativo por parte del investigador y los sujetos. Una ventaja de las técnicas de primer y segundo grado es la facilidad de controlar los datos recolados por el investigador y el contexto de recolección. Las técnicas de tercer grado son más baratas pero no ofrecen el mismo control al investigador.

Es importante decidir cuidadosamente que datos recolectar y como recolectarlos. Mientras los objetivos y las preguntas de investigación del estudio de casos están siendo decididos es imposible definir esta selección. Sin embargo, cuando el estudio de casos ya está en progreso, el investigador tiene una mayor apreciación de los datos deseables, los disponibles y los factibles de recolectar y luego analizar.

**Entrevistas:** La recolección de datos basada en entrevistas es de las más usadas y mas importantes en los estudios de casos en ingeniería de software. Casi todos los estudios de casos implican algún tipo de entrevista, ya sea para la recolección de datos primarios o para validaciones de tipos de datos. La razón de ello es que gran parte del conocimiento que es de interés no está disponible en ningún otro lugar más que en las mentes de las personas que participan del estudio de casos. Las entrevistas pueden ser llevadas a cabo de diferentes maneras, pero en general el investigador habla y hace preguntas al interrogado, quien responde a estas. Dentro de las variantes al conducir una entrevista podemos detallar, sesiones de entrevistas más o menos estructuradas, entrevistas más o menos largas, preguntas más o menos específicas.

El dialogo entre el investigador y los entrevistados es guiado por un conjunto de preguntas. Las preguntas están basadas en los tópicos de interés del estudio de casos y pueden ser abiertas o cerradas. Una pregunta abierta permite una respuesta amplia, sin embargo una pregunta cerrada limita la respuesta a un cierto rango de opciones. La

ventaja de las preguntas abiertas es que el alcance de la respuesta no tiene límite a diferencia de las cerradas. La ventaja de las preguntas cerradas es que es más fácil analizar los resultados.

Por otra parte las entrevistas se dividen en semi estructuradas, no estructuradas y completamente estructuradas. En las entrevistas no estructuradas las preguntas son formuladas de forma abierta. En este caso la conversación se realiza en base a los intereses del entrevistado y del investigador. Este tipo de entrevistas en más factible para estudios exploratorios. En las entrevistas semi estructuradas las preguntas son planificadas, pero no necesariamente se piden en el orden en que fueron listadas. El estilo de preguntas es un mix entre abiertas y cerradas. Entrevistas semi estructuradas son comunes en estudios de casos de ingeniería de software. En las entrevistas completamente estructuradas las preguntas son planificadas en detalle y se piden en el mismo orden en que se planeo. Para este tipo de entrevistas es deseable que las preguntas sean cerradas, generalmente resultan ser cuestionarios.

Durante la fase de planificación del estudio de casos se determina a quién entrevistar intentando involucrar personas con diferentes roles y personalidades. La cantidad de entrevistados debe ser decidida durante el estudio de casos. El criterio para determinar el máximo número de entrevistados es la saturación, es decir, cuando no se obtiene más información agregando un nuevo integrante.

Una vez formuladas las preguntas de la entrevista, pueden ser utilizadas sobre un primer conjunto de entrevistados (entrevista piloto). Esto permite comprobar que las preguntas fueron comprendidas por los entrevistados y que las respuestas brindadas son útiles para el investigador. Las entrevistas pilotos son conducidas de igual forma que una entrevista tradicional, pero con un extra de atención para conocer la comprensión de las preguntas.

Una sesión de entrevista se puede dividir en varias fases. Al comienzo el investigador presenta los objetivos de la entrevista y del estudio de casos y explica como los datos recabados de la entrevista van a ser usados. Durante esta etapa el investigador puede pedir permiso de grabar la entrevista. Luego de esta introducción tanto el entrevistado como el investigador se sienten más cómodos y relajados. Las preguntas de la entrevista, que constituyen la parte más larga de la entrevista se realizan a continuación de la introducción. Si la entrevista contiene preguntas personales o sensibles, concernientes a aspectos políticos, opiniones sobre colegas, conflictos, la competencia, etc., se debe tener especial cuidado. En esta situación es importante que el entrevistado confié en el entrevistador, y que este ultimo garantice la confidencialidad de los datos brindados.

Diferentes tipos de preguntas pueden hacerse de acuerdo con los siguientes tres modelos. El modelo embudo comienza con preguntas más abiertas y continua con preguntas más especificas. El modelo pirámide comienza con preguntas especificas y luego se van realizando preguntas abiertas a lo largo de la entrevista. En el modelo reloj de arena se realiza un mix de preguntas específicas y abiertas.

**Grupos focales:** A diferencia de las entrevistas, en los grupos de enfoque el investigador realiza una serie de preguntas a un grupo de

personas al mismo tiempo. Este tipo de enfoque es muy útil de usar en estudios cualitativos. En grupo las personas tienden a soltarse más en el sentido de confirmar o rechazar hechos que han ocultado en la entrevista individual. Es rentable (costo/efectividad) debido a que varias personas son entrevistadas al mismo tiempo. Este tipo de enfoque tiene la debilidad de que el grupo puede ser desordenado, hablar uno encima del otro, etc. En estos casos de debe contar con un moderador con experiencia que pueda llevar adelante la sesión.

**Observaciones:** Las observaciones son conducidas para investigar como una cierta actividad es conducida por un grupo de personas. Existen varias alternativas de observaciones, una de ellas es monitorear al grupo grabando un video y luego analizar lo grabado, observar una reunión, o utilizar herramientas que hagan preguntas a los participantes cada ciertos intervalos de tiempo.

**Datos de archivos:** Datos de archivos se refiere a datos que se encuentran disponibles en archivos. Ejemplos de estos son: minutas de reuniones, documentos técnicos, documentos de gestión, registros financieros, reportes, etc. Datos de archivo es una técnica de recolección de datos de tercer grado en un estudio de casos.

**Métricas:** Las técnicas de recolección mencionadas anteriormente se enfocan en datos cualitativos, sin embargo los datos cuantitativos también son de importancia en un estudio de casos. Medir el software es el proceso de representar las entidades del software, los procesos y productos en valores cuantitativos. En importante definir métricas que sean realmente de interés para que luego la recolección sea exitosa. En este método se definen en primera instancia los objetivos, las preguntas son refinadas basadas en dichos objetivos y luego las métricas se derivan de las preguntas, por lo tanto las métricas que se recogen son relevantes.

Para fortalecer la validez de la investigación empírica se recomienda tomar múltiples perspectivas hacia el objeto de estudio y ofrecer así una visión amplia, lo que se denomina triangulación. Se pueden aplicar los siguientes cuatro tipos de triangulación:

- Datos de origen: utilizando más de una fuente de datos o recogiendo los mismos datos en diferentes ocasiones.
- Observadores: utilizando más de un observador en el estudio.
- Metodológica: combinando diferentes tipos de métodos de recolección de datos.
- Teoría: utilizando teorías o puntos de vistas alternativos.

## 6. Investigación

El GrIS ha llevado adelante una serie de experimentos formales para conocer el comportamiento de distintas técnicas de verificación, ha empaquetado un experimento realizado y ha evaluado el funcionamiento del paquete. Además propuso un marco de comparación de experimentos formales.

Actualmente han culminado 4 experimentos. El primero se llevó a cabo en la Facultad de ingeniería en el 2008 (15). En este experimento se entrenó a un grupo de 17 estudiantes para aplicar 5 técnicas de verificación a un pequeño programa escrito en Java. Los estudiantes registran los defectos que encuentran. Cada estudiante aplicó sólo una de las técnicas. Las técnicas son inspecciones, partición en clases de equivalencia y valores limites, tablas de decisión, trayectorias linealmente independientes y cubrimiento de condición múltiple. Las conclusiones más relevantes fueron tres. En primer lugar se detectó que los defectos de performance no son fáciles de encontrar. En segundo lugar el costo de la verificación unitaria es alto y el porcentaje de defectos que se detecta es bajo. Por último, con la técnica de inspecciones se detecta una mayor variedad de defectos que con las otras técnicas.

En el siguiente experimento formal se busca conocer la efectividad de 5 técnicas de verificación unitaria (14). Las técnicas son: inspecciones, partición en clases de equivalencia y valores limites, tablas de decisión, trayectorias linealmente independientes y cubrimiento de condición múltiple. El diseño propuesto es de un factor con múltiples alternativas. El factor es la técnica de verificación y las alternativas las 5 posibilidades. El experimento es ejecutado por un conjunto de 14 estudiantes que aplican las técnicas sobre 4 programas diferentes desarrollados especialmente para este experimento. Los 14 sujetos participaron en el experimento anterior, donde fueron entrenados con el uso de las técnicas. Cada sujeto (menos uno) verifica 3 de los 4 programas aplicando una técnica distinta. El análisis de resultados se realiza utilizando estadística descriptiva y la prueba de hipótesis no paramétrica Mann-Whitney. Los resultados de la prueba indican que las técnicas partición en clases de equivalencia y tablas de decisión son más efectivas que trayectorias linealmente independientes.

En el 2011 se lleva a cabo un experimento formal que compara el comportamiento de las técnicas de verificación Cubrimiento de sentencias (CS) y Todos los usos (TU) (17). El diseño de este experimento es también de un factor con dos alternativas. Un grupo de 21 estudiantes realizan pruebas sobre un único programa en Java. En este caso el experimento se divide en dos experiencias, una con 10 sujetos (5 ejecutan CS y 5 ejecutan TU) y otra con 11 sujetos (5 ejecutan CS y 6 ejecutan TU). Como resultado utilizando estadística descriptiva se obtiene que la técnica TU tiene una mayor efectividad que la técnica CS. La efectividad promedio de TU es de 34,3 % y 29,2 % la de CS. Sin embargo, los test estadísticos realizados no rechazan la hipótesis de igualdad en la efectividad de ambas técnicas. Por otro lado, se encuentra que la técnica TU es notoriamente más costosa que la técnica CS; 328 minutos para TU y 133 minutos para CS es lo que llevó en promedio desarrollar los casos de prueba para el software bajo prueba. Los test estadísticos aplicados muestran que en este experimento hay suficiente evidencia estadística para afirmar que TU es más costosa que CS.

Parte de los estudiantes que participaron del experimento anterior forman parte de un nuevo experimento que busca comparar el comportamiento de las técnicas de verificación Cubrimiento de Senten-

cias (CS) y Todos los Usos (TU). Se busca analizar las técnicas con el propósito de conocer su efectividad y costo a nivel unitario, en el contexto de un experimento controlado. Participan 14 sujetos del experimento anterior, probando sobre un mismo programa, 6 de ellos aplican CS y 8 aplican TU. La ejecución del experimento se realiza en una única instancia con una duración de 8 semanas. Los resultados obtenidos con los test de hipótesis indican que no existe evidencia estadística para afirmar que una técnica es más efectiva que la otra. Respecto a la comparación del costo de las técnicas, se concluye que existe evidencia estadística que indica que TU es más costosa que CS. Este experimento no se encuentra aún publicado.

Además de haber realizado diversos experimentos, es de interés del GrIS contar con un marco de comparación de experimentos que permita compararlos formalmente y avanzar en la construcción de nuevos experimentos basándose en experimentos anteriores. Se propone un marco de comparación de experimentos formales que evalúan técnicas de verificación y como ejemplo se utiliza el marco con algunos experimentos conocidos (16). Se buscan características relevantes de los experimentos para poder compararlos, siendo los objetivos de los experimentos el primer punto de interés de forma de determinar si tienen el mismo propósito general. Luego el marco propone examinar los factores y alternativas de los diseños elegidos, así como las características de los sujetos participantes, la duración del experimento, la distribución de los sujetos, los lineamientos seguidos para la asignación de las técnica y programa a los sujetos, etc. Este marco busca proveer formalidad al momento de comparar distintos experimentos. Estas comparaciones no son triviales debido a la alta variabilidad de las características de los experimentos. Como resultado se obtiene una mayor comprensión de los aspectos más relevantes de cada experimento y se los puede comparar rápidamente según distintos aspectos relevantes.

Un paquete de laboratorio es el contenedor del conocimiento y materiales necesarios para replicar un experimento. En 2012 se construye un paquete de laboratorio (PL) para un experimento controlado de ingeniería de software que se realizó en el marco de trabajo del GrIS (1); para esto utilizamos la propuesta de empaquetamiento de experimentos de Solari (9). El estudio tiene dos objetivos: obtener una instancia de paquete para un experimento concreto y validar la propuesta genérica. Para realizar la instancia del PL se siguió un proceso definido que incluyó actividades de revisión, verificación y validación de los resultados. Los resultados del trabajo confirman la viabilidad y completitud de la propuesta de PL. Se ha obtenido un documento que abarca las distintas actividades del proceso experimental y contiene el conocimiento relativo al experimento en una única estructura.

Luego de presentada la propuesta de estructura de PL para experimentos (1) se busca validar la misma (10). Se realiza una evaluación a una réplica de experimento que utiliza un PL estructurado de acuerdo a la propuesta. Se pretende validar que el uso del PL implica una mejora con respecto a otros PL no estructurados. En los resultados obtenidos se pudo comprobar que el PL utilizado es completo con respecto al

proceso de investigación experimental, cubriendo las actividades de replicación en mayor grado que los PL no estructurados. También se comprobó que el documento tiene un grado de usabilidad aceptable desde el punto de vista del replicador responsable. La evaluación de la replicación muestra que la misma se ha realizado en forma más eficaz que otras anteriores. Esto se ha valorado analizando los incidentes de replicación y dudas surgidas. En cuanto a la eficiencia, la replicación evaluada fue más compleja que otras anteriores y demandó mayor esfuerzo. Sin embargo, no se han observado efectos negativos por el uso del PL.

Además, hace varios años que se realizan pruebas de procesos de desarrollo de software en el marco de una asignatura llamada Proyecto de Ingeniería de Software (13). Si bien estas pruebas no son formales, es interesante en un futuro formalizarlas.

## 7. Conclusiones

En el reporte se presentan conceptos teóricos básicos de la Ingeniería de Software Empírica y las principales técnicas para la investigación empírica: encuestas, estudios de casos y experimentos. Se pretende que este documento pueda ser utilizado por estudiantes e investigadores que se encuentran realizando trabajos de ISE en el marco del trabajo del GrIS.

También se describen algunos de los trabajos de investigación que ha realizado el GrIS, experimentos formales que buscan conocer el comportamiento de distintas técnicas de verificación, el paquete de laboratorio para experimentos y la evaluación realizada al mismo, y se describe el marco de comparación para experimentos formales.

## Bibliografía


(1) C. Apa, M. Solari, D. Vallespir, and S. Vegas. Construcción de un paquete de laboratorio para un experimento en ingeniería de software. In *Proceedings Ibero-American Conference on Software Engineering*, 2011. 6

(2) K. M. Eisenhardt. *Building theories from case study research*. The Academy of Management Review, 1989. 5.1

(3) N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, February 1998. 1

(4) N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001. 1

(5) E. J. Martínez. Notas del curso de posgrado maestría en estadística matemática. Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2004. 4.5.3

(6) P. Nelson, M. Coffin, and K. Copeland. *Introductory statistics for engineering experimentation*. Elsevier Science, California, 2003. 4.5.3

(7) R. Runeson, Höst and Regnell. *Case Study Research in Software Engineering*. John Wiley & Sons, New Jersey, USA, 2012. 1

(8) N. C. Smith. *The Case Study: A Useful Research Method For Information Management*. Journal of Information Technology, London, 1990. 5.1

(9) M. Solari. *Propuesta de Paquete de Laboratorio para Experimentos de Ingeniería de Software*. PhD thesis, Universidad Politécnica de Madrid, 2012. 6

(10) M. Solari, C. Apa, and S. Vegas. Evaluación del uso de un paquete de laboratorio en una replicación experimental. In *VIII Experimental Software Engineering Latin American Workshop (ESELAW 2011)*, 2011. 6
(11) M. Spiegel. *Estadística - 2da Edición*. Mc.Graw-Hill, Madrid, 1991. 4.5.3
(12) S. E. S. T.C. Lethbridge and J. Singer. *Studying software engineers: data collection techniques for software field studies*. Empirical Software Engineering, 2005. 5.2
(13) J. Triñanes. Construcción de un banco de pruebas de modelos de proceso. In *Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento*, 2004. 6
(14) D. Vallespir, C. Apa, S. De León, R. Robaina, and J. Herbert. Effectiveness of five verification techniques. In IEEE-Computer-Society, editor, *Proceedings of the International Conference of the Chilean Computer Society*, 2009. 6
(15) D. Vallespir and J. Herbert. Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In IEEE-Computer-Society, editor, *Proceedings of the Mexican International Conference in Computer Science*, 2009. 6
(16) D. Vallespir, S. Moreno, C. Bogado, and J. Herbert. Towards a framework to compare formal experiments that evaluate verification techniques. In *Proceedings of the Mexican International Conference in Computer Science*, 2009. 6
(17) D. Vallespir, S. Moreno, C. Bogado, and J. Herbert. Comparando las técnicas de verificación todos los usos y cubrimiento de sentencias. In *Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento*, 2010. 6
(18) C. Wohlin, P Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 1, 4.5.3
(19) R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 2003. 5.1

# ANEXO B

En este anexo se presenta el artículo enviado a la revista Milveinticuatro.

## Pruebas Unitarias en Java
## *Diseño por Contratos*

Silvana Moreno Grupo de Ingeniería de Software, UdelaR
Diego Vallespir Grupo de Ingeniería de Software, UdelaR
Álvaro Tasistro ORT Uruguay

En artículos anteriores de esta serie hemos presentado generalidades de las pruebas unitarias para luego enfocarnos principalmente en el cubrimiento de código. Este artículo cambia su foco hacia el Diseño por Contrato y cómo este, al ser ejecutable, se puede utilizar como parte de las pruebas unitarias. Presentamos de forma muy resumida qué es el Diseño por Contratos y el JML, para luego, a modo de ejemplo, especificar el contrato del método *merge* (método que utilizamos en los artículos anteriores).

### Diseño por contratos

El Diseño por Contratos da una visión de la construcción de sistemas como un conjunto de elementos de software cooperando entre sí. Los elementos juegan en determinados momentos alguno de los dos roles principales: proveedores o clientes. El cliente es quién se beneficia utilizando el elemento y el proveedor es quién origina resultados al elemento.

La cooperación entre proveedor y cliente se puede especificar mediante un contrato que establece obligaciones y beneficios. El contrato establece que se cumplirán ciertas condiciones de entrada (precondiciones), y que se deberán garantizar ciertas condiciones de salida (poscondiciones).

La novedad del Diseño por Contrato es que hace que los contratos sean ejecutables. Los contratos son definidos en el código del programa en un determinado lenguaje, y se traducen a código ejecutable por el compilador. Por lo tanto, cualquier violación del contrato que se produce mientras se ejecuta el programa puede ser detectada inmediatamente.

Los contratos de software se especifican mediante la utilización de expresiones lógicas denominadas aserciones. Existen diferentes tipos de aserciones, entre ellas se encuentran las precondiciones y las poscondiciones. Este artículo presentará este tipo de aserción.

Una precondición establece lo que espera recibir el proveedor. Visto de otro modo, define las condiciones que debe garantizar el cliente al momento de pedirle al proveedor cierto servicio. En lo que refiere a los contratos en la orientación a objetos las precondiciones normalmente se definen a nivel de los métodos de las clases.

Una poscondición da como garantías al cliente ciertas propiedades que se cumplen después de la llamada al método. En definitiva, especifica lo que se cumple luego de que se ejecuta el método. Visto desde el lado del proveedor, la poscondición es lo que éste debe asegurar que se cumpla, siempre y cuando el cliente haya respetado la precondición al momento de invocar el servicio.

### Java Modeling Language

Existen varios lenguajes que permiten especificar formalmente contratos de software. Uno de ellos es JML, sinónimo de "Java Modeling Language". Se trata de un lenguaje desarrollado para especificar y verificar clases de Java.

Añadir especificaciones JML a un programa ayuda a comprender qué función debe realizar un método o una clase. También ayuda a encontrar defectos en los programas, puesto que se puede comprobar si un método cumple con su especificación cada vez que se ejecuta.

Existen herramientas que permiten compilar la especificación formal para ciertos lenguajes de especificación. Para el JML existe el compilador JMLC, este es una extensión de un compilador Java y compila los programas Java con especificaciones JML. En el código generado se agregan instrucciones ejecutables que chequean en tiempo de ejecución si los métodos cumplen con sus especificaciones. En caso de que una especificación no se cumpla, se interrumpe la ejecución del programa y se notifica qué especificación no se cumple. Esto permite detectar defectos y por ende se puede utilizar como una forma de probar el programa durante el desarrollo de software.

El JML dispone de dos cláusulas específicas para indicar las precondiciones y las poscondiciones de un método: **requires** y **ensures**. Estas cláusulas deben ir situadas justo antes de la declaración del método.

Además, el JML añade un conjunto de operadores que hace más cómodo en muchos casos construir aserciones complejas. Entre estos operadores se incluyen los cuantificadores más comunes (\forall, \exists, \sum, \product, \max, \min, etc) que hacen del JML un lenguaje similar al lenguaje de predicados de lógica.

También el JML define dos seudovariables que pueden ser utilizadas en las poscondiciones de los métodos; \result: valor devuelto por el método y \old(E): valor de la expresión E al comenzar la ejecución del método.

## Un ejemplo sencillo

Como ejemplo utilizamos el método *merge* que fue presentado en los artículos anteriores de esta serie. Dicho método recibe como parámetros dos arreglos de enteros ordenados de menor a mayor. El método retorna otro arreglo con esos elementos ordenados de menor a mayor. Además, los arreglos de entrada no deben de ser alterados, es decir, sus valores se mantienen incambiados al salir del método.

La firma del método es la siguiente:

```
public int[] getMerge(int[] a, int[] b)
```

Al invocar al método se deben pasar dos arreglos ordenados de menor a mayor, lo que nos indica una precondición que se debe cumplir. Para cumplir con dicha precondición es necesario que para todos los elementos del arreglo se cumpla que el elemento que está en la posición i del arreglo sea menor o igual al elemento que se encuentra en la posición i+1.

Una forma de especificar formalmente dicha precondición se muestra en las siguientes dos aserciones.

```
/*@ requires (\forall int i; i >= 0 && i < a.length-1; a[i] <= a[i+1]); @*/
/*@ requires (\forall int j; j >= 0 && j < b.length-1; b[j] <= b[j+1]); @*/
```

El cuantificador \forall sirve para recorrer el arreglo desde una posición y en determinado rango, haciendo cumplir determinada condición. En este caso se recorren los arreglos y chequea que para todos elementos el valor de a[i] sea menor o igual al de a[i+1]. La forma de leer la precondición es “Para todo i entre cero (in-

clusive) y el largo del array menos 1 (sin incluir este) se cumple que a[i] es menor o igual a a[i+1]".

La anterior precondición puede especificarse también utilizando el operador AND y un único *requires* obteniendo el mismo significado. El hecho de utilizar varios *requires* significa que todas las aserciones deben cumplirse, con lo cual la concatenación de *requires* utiliza el AND de forma implícita.

Al terminar de ejecutarse el método se debe cumplir que los arreglos de entrada no fueron alterados. Esto se especifica como una poscondición y una forma de especificarlo es utilizando la pseudovariable \old(e).

Se plantea recorrer los arreglos de entrada y verificar que para todos sus elementos el valor en cada posición después de llamar al método es igual al que tenia previo al llamado (a[i] = = \old(a[i])).

```
/*@ ensures (\forall int i; i >= 0 && i <= a.length-1; a[i] == \old(a[i])); @*/
/*@ ensures (\forall int j; j >= 0 && j <= b.length-1; b[j] == \old(b[j])); @*/
```

Hasta aquí hemos tenido en cuenta las precondiciones y poscondiciones para los arreglos de entrada al método. Veamos ahora las condiciones que se deben cumplir en lo que respecta al arreglo esperado como resultado del método.

Como se mencionó anteriormente el arreglo devuelto debe contener de forma ordenada los mismos elementos que los arreglos pasados por parámetro.

Para referenciar al arreglo resultado se utiliza la pseudovariable \result y se recorre el mismo, como ya vimos anteriormente, para verificar el orden de los elementos de menor a mayor.

```
/*@ ensures (\forall int k; k >= 0 && k < \result.length-1; \result[k] <=
    \result[k+1]); @*/
```

A continuación presentamos una especificación informal para que el arreglo devuelto contenga los mismos elementos de los arreglos pasados por parámetro y no otros:

- El largo del arreglo resultado debe ser igual a la suma del largo de los arreglos **a** y **b**.
- Además, tanto los elementos del arreglo **a** como los del **b** deben estar en el arreglo resultado.
- Además, todos los elementos del arreglo resultado se encuentran o en el arreglo **a** o en el **b**.

Especificamos formalmente que el largo del arreglo resultado es igual a la suma del largo del arreglo **a** y el **b** de la siguiente forma:

```
/*@ ensures \result.length == (a.length + b.length); @*/
```

Para que todos los elementos del arreglo **a** se encuentren en el arreglo resultado especificamos:

```
/*@ ensures (\forall int m; m >= 0 && m <= a.length-1; (\exists int n; n >= 0
&& n <= \result.length-1; a[m] == \result[n])); @*/
```

Esta especificación se lee de la siguiente manera "Para todo m entre cero (inclusive) y el largo del arreglo a menos 1 (inclusive) se cumple que existe un número n entre cero (inclusive) y el largo del arreglo resultado menos 1 (inclusive) tal que el elemento en la posición m del arreglo **a** es igual al elemento en la posición n del resultado." Se puede simplificar la lectura diciendo "Cualquier elemento del arreglo **a** se encuentra en alguna posición del arreglo resultado." La especificación de esta poscondición para el arreglo **b** es espejo a la especificación para **a**.

Nos resta especificar que todos los elementos del arreglo resultado se encuentren en el arreglo **a** o en el arreglo **b**. Es decir, el arreglo resultado no tiene elementos "extraños".

```
/*@ ensures (\forall int i; i >= 0 && i <= \result.length-1; (\exists int n; n >= 0 && n <= a.length-1; a[n] == \result[i]) or (\exists int m; m >= 0 && m <= b.length-1; b[m] == \result[i])); @*/
```

En este artículo hemos presentado el Diseño por Contrato y la especificación de un ejemplo sencillo utilizando el JML. La especificación de contratos ejecutables permite introducir pruebas en el propio código y son muy efectivas para encontrar defectos durante el desarrollo de software. Sin embargo, como se desprende de este ejemplo, las especificaciones formales no son sencillas y tienen, como cualquier actividad, un costo asociado. Como con cualquier otra técnica se debe estudiar el costo-beneficio previamente a aplicarla.

Por último, les comentamos que la especificación que presentamos en el ejemplo no es correcta. Hasta el próximo artículo de esta serie les dejamos como entretenimiento revisar la especificación para encontrar el/los problemas.