Universidad Nacional de La Plata

Facultad de Informática

TESINA DE LICENCIATURA

Título: Interacciones entre aspectos: estudio sobre un caso industrial

Autores: Alvarez, Alejandro Nicolás

Director: Gordillo, Silvia

Codirector: Zambrano, Arturo

Carrera: Licenciatura en Informática – Plan 90

Resumen

Si bien la orientación a aspectos es aceptada como un mecanismo efectivo para la separación de *crosscutting concerns,* la característica de *obliviousness* entre aspectos donde el programa base desconoce la existencia de los mismos, hace que el comportamiento del sistema sea mucho más difícil de comprender. Además, debido al hecho de que los aspectos afectan varios elementos de la aplicación, es probable que los mismos interfieran entre sí de alguna manera. Esto da lugar a que se produzcan interacciones entre los aspectos, necesarias para lograr la funcionalidad final esperada. Las interacciones entre aspectos son un tema abierto de investigación. *Sanen et al.* presentan un estudio sobre las interacciones, donde los autores clasifican las mismas en las siguientes categorías: *dependency, conflict, mutex y reinforcement*.

El presente Trabajo de Grado propone estudiar las interacciones entre aspectos sobre una implementación de complejidad considerable, que involucre varios aspectos funcionales. Para esto, se implementó un *software* complejo como es el de las máquinas tragamonedas, dominio en el cual hemos trabajado por 3 años. En este estudio se reportan casos concretos de interacciones y se presentan implementanciones para el tratamiento de las mismas. En algunos casos mediante mecanimos *ad-hoc* y en otros casos con una solución genérica, aplicable a otros dominios.

Palabras Claves

Programación Orientada a Aspectos.
Interacciones entre Aspectos: *dependency, reinforcement, conflict y mutex.*
Crosscutting concerns.
Ingeniería de Software.
Máquinas tragamonedas.
Java.
AspectJ.

Trabajos Realizados

Se implementó el software de una *SM* que incluye varios *crosscutting concerns* funcionales y permite la interacción con el usuario. Se identificó el código correspondiente a las interacciones del dominio reportadas por *Zambrano et al*. Se seleccionó una interacción para cada una de las categorías de la taxonomía propuesta por *Sanen et al.* Para cada interacción se estudió un mecanismo que permite su tratamiento, realizando una implementación concreta del mismo sobre el *software* desarrollado. Para cada uno de los mecanismos desarrollados se analizaron ventajas y desventajas sobre distintos factores de la Ingeniería de Software.

Conclusiones

Se identificaron interacciones sobre los *crosscutting concerns* funcionales y no funcionales del dominio. Las mismas se categorizaron según la taxonomía de *Sanen et al*. Se seleccionó una interacción para cada una de las categorías, para las cuales se estudió e implementó un mecanismo para su tratamiento sobre el *software* desarrollado. Las interacciones fueron implementadas de forma tal que la *SM* se comporte de la manera deseada. Para cada uno de los mecanismos desarrollados, se analizaron ventajas y desventajas con respecto a la mantenibilidad, genericidad, escalabilidad y modularización.

Trabajos Futuros

Se propone como trabajo a futuro, aplicar los mecanismos propuestos en otros dominios, analizando si los mismos permiten implementar otras interacciones pertenecientes a las categorías presentadas en este trabajo. De esta manera se puede comprobar si las generalizaciones propuestas son reusables. Por otra parte, es necesario profundizar el estudio de los mecanismos planteados e implementar formas alternativas para tratar las interacciones, con el objetivo de eliminar algunas de las limitaciones de los mecanismos propuestos.

Fecha de la presentación: Abril de 2013

Índice general

1. Introducción **1**

1.1. Motivación 2

1.2. Objetivos 2

1.3. Contribución y resultados obtenidos 3

1.4. Estructura del documento 3

2. Programación Orientada a Aspectos e Interacciones entre Aspectos **5**

2.1. Programación Orientada a Aspectos 5

2.2. Interacciones entre aspectos 7

2.2.1. Taxonomía de las interacciones 7

2.3. AspectJ 8

2.3.1. Sintaxis 8

3. Concerns en el Dominio de las Slots Machines **13**

3.1. Dominio de las Slots Machines 13

3.1.1. El juego 14

3.1.2. Meters 15

3.1.3. Hardware 15

3.1.4. Program resumption 18

3.1.5. Game recall 19

3.1.6. Communication protocols 19

3.1.7. Modo Demo 20

3.2. Concerns identificados en el dominio a nivel de requerimientos 21

3.3. Relaciones entre concerns 22

4. Implementación Orientada a Aspectos de una SM 25

4.1. Game 26

4.2. HAL 27

4.3. Meters 28

4.4. Program Resumption 30

4.5. Game Recall 31

4.6. Errors Conditions 31

4.7. G2S y PCP 33

4.8. Demo 35

5. Interacciones entre Aspectos 37

5.1. Interacciones identificadas en el dominio de las SM 37

5.2. Mutex 39

5.2.1. Implementación de la Interacción 40

5.2.2. Instanciación del mecanismo para el comando Set Time . . . 42

5.2.3. Estrategias alternativas 44

5.3. Conflict 44

5.3.1. Tratamiento de la interacción 45

5.3.2. Tratamiento de la interacción entre los concerns: Demo y Meters 46

5.3.3. Tratamiento de la interacción entre los concerns: Demo y Program Resumption 47

5.3.4. Tratamiento de la interacción entre los concerns: Demo y Communication Protocols 48

5.3.5. Generalización del mecanismo 49

5.4. Dependency 50

5.4.1. Tratamiento de la interacción 50

5.4.2. La interacción de tipo Dependency no requiere una implementación ad-hoc 51

5.5. Reinforcement 53

5.5.1. Tratamiento de la interacción 53

5.6. Análisis de los mecanismos desarrollados 56

5.6.1. Modularización 57

5.6.2. Generalización 57

5.6.3. Escalabilidad 58

5.6.4. Mantenibilidad 58

5.6.5. Weaving dinámico 59

6. Conclusiones y Trabajos Futuros **61**

A. Modelo de Objetos **63**

A.1. Paquete Game 63

A.2. Paquete HAL 65

A.3. Paquete Meters 66

A.4. Paquete Game Recall 67

A.5. Paquete Errors 68

A.6. Paquete Protocols 69

B. Software desarrollado **71**

B.1. Proyecto Egm 71

B.2. Proyecto EgmUI 72

Agradecimientos

Dedico este trabajo a Naty, por su amor,
ayuda, paciencia, por estar siempre.
A mis padres, Norma y Eloy, por todo, por
darme la vida, por sus valores y consejos.
A mis hermanos, Belén y Pablo,
por estar y apoyarme siempre.
A mis tíos, Marga y Bocha,
por su ayuda y motivación constante.
A Mirta y Chano, por ayudarme desde el inicio.
A la memoria de mi querida abuela Marunga.

Agradezco hoy y siempre a mi familia que siempre han procurado mi bienestar y que si no fuese por el esfuerzo realizado por ellos, en especial mis padres, mis estudios no hubiesen sido posibles.

Quiero expresar mi más sincero agradecimiento a mi codirector Arturo Zambrano, por la orientación, dedicación y esfuerzo, que me brindó para la realización de esta tesis.

De igual manera mis agradecimientos a mi directora Silvia Gordillo y al Laboratorio LIFIA.

Nuevamente gracias a Arturo y a Johan Fabry, por darme la posibilidad de realizar una estadía de investigación en el exterior, por enriquecer con sus conocimientos y sugerencias el desarrollo de este trabajo.

Gracias a mis compañeros de LIFIA, en especial a los que durante el desarrollo de este trabajo me incentivaron a concluirlo.

Gracias a mis amigos, por estar, por ser amigos.

Alejandro N. Alvarez

Capítulo 1

Introducción

Si bien la orientación a aspectos[1] es aceptada como un mecanismo efectivo para la separación de *crosscutting concerns* [17], la característica de *obliviousness* entre aspectos [8] donde el programa base desconoce la existencia de los mismos, hace que el comportamiento del sistema sea mucho más difícil de comprender. Además, debido al hecho de que los aspectos afectan varios elementos de la aplicación, es probable que los mismos interfieran entre sí de alguna manera. Esto da lugar a que se produzcan interacciones entre los aspectos, necesarias para lograr la funcionalidad final esperada.

Las interacciones entre aspectos son un tema abierto de investigación [20]. *Sanen et al.* [23] presentan un estudio sobre las interacciones, donde los autores clasifican las mismas en las siguientes categorías: *dependency*, *conflict*, *mutex* y *reinforcement*.

El presente *Trabajo de Grado* propone estudiar las interacciones entre aspectos sobre una implementación de complejidad considerable, que involucre varios aspectos funcionales. Para esto, se implementó un *software* complejo como es el de las máquinas tragamonedas, dominio en el cual hemos trabajado por tres (3) años.

En este estudio se reportan casos concretos de interacciones [29] y se presentan implementanciones para el tratamiento de las mismas. En algunos casos mediante mecanimos *ad-hoc*, como los presentadas en [28] y en otros casos con una solución genérica.

[1]En el presente documento al referirnos a aspectos es en el sentido de *Aspect Oriented Programming*.

1.1. Motivación

Las interacciones entre aspectos se han estudiado a niveles teóricos [4] o han sido analizadas a partir de ejemplos muy simples, por lo tanto es necesario estudiar las mismas en casos concretos del mundo real. Para esto, proponemos la implementanción orientada a aspectos de un *software* de una complejidad considerable.

El *software* de una *SM* (*Slot Machines*) presenta numerosos ejemplos de interacciones entre *concerns*. Consideremos el siguiente caso:

> Las *SM*s poseen una cantidad considerable de contadores, que reflejan las actividades que ocurren dentro de esta. Por ejemplo: cantidad de juegos realizados, cantidad de monedas insertadas, cantidad de dinero apostado. Por otro lado, existen protocolos de comunicación usados para monitorear de manera remota el comportamiento de una *SM*, que reportan la información que almacenan los contadores.

En este sistema, un aspecto mantiene la información de los contadores afectando diferentes *concerns*. A la vez, un aspecto que reporta esta información necesita de los contadores para proveer su funcionalidad correctamente. Estas restricciones y otras, deben ser respetadas para asegurar el correcto funcionamiento del sistema. Al implementar un sistema de estas características es necesario estudiar y controlar las interacciones aspectuales lo cual motiva este trabajo de tesis.

1.2. Objetivos

El objetivo principal de este trabajo es tratar las interacciones entre aspectos, implementando mecanismos de forma *ad-hoc* o genéricos, con el fin de asegurar que se cumplan las restricciones impuestas por las interacciones en un desarrollo orientado a aspectos.

Para esto, tomando como referencia los *concerns* más representativos en el dominio de las *SM*, estudiados en la etapa de análisis de requerimientos [30], se propone implementar el *software* de una *SM* mediante el uso de *Aspect Oriented Programming*.

Este desarrollo es la base para estudiar los mecanismos necesarios para tratar las interacciones aspectuales y analizar su impacto en el sistema.

1.3. Contribución y resultados obtenidos

- Se implementó el *software* de una *SM* que incluye varios *crosscutting concerns* funcionales y permite la interacción con el usuario.
- Se identificó el código correspondiente a las interacciones del dominio reportadas [29].
- Se seleccionó una interacción para cada una de las categorías de la taxonomía propuesta por *Sanen et al.* [23]. Para cada interacción se estudió un mecanismo que permite su tratamiento, realizando una implementación concreta del mismo sobre el *software* desarrollado.
- Para cada uno de los mecanismos desarrollados se analizaron ventajas y desventajas sobre distintos factores de la Ingeniería de Software.

1.4. Estructura del documento

El siguiente capítulo detalla los conceptos más importantes de la Programación Orientada a Aspectos (*AOP*) [18]. También se describe la clasificación para las interacciones entre aspectos propuesta por Sanen *et al.* [23]. Por último, se introducen las caracaterísticas más relevantes de *AspectJ* [16], extensión para *AOP* del lenguaje *Java* utilizada en la implementación del *software* de una *SM*,

El capítulo 3 introduce conceptos referidos al dominio de las máquinas tragamonedas. Se detallan requerimientos y funcionalidades sobre los cuales se identifican varios *concerns* del dominio. Entre los mismos, se identificaron instancias de las interacciones que son el objeto de estudio del presente trabajo.

El capítulo 4 presenta detalles de la implementación del *software* de una *SM* realizada utilizando *AspectJ*. Se describen los objetos y aspectos que componen cada *concern*. En el apéndice A se puede encontrar el detalle completo del modelo de objetos desarrollado. En el apéndice B se indica la manera de interactuar con el *software* presentado.

En el capítulo 5 se detallan los mecanismos desarrollados para el tratamiento las interacciones encontradas. Basados en la clasificación de *Sanen et al.* [23], se estudian interacciones del dominio para los tipos *Mutex, Conflict, Dependency* y *Reinforcement*. Sobre el final de este capítulo se presenta un análisis de los mecanismos desarrollados. En el mismo se discuten ventajas y desventajas en lo referido a generalización, modularización, escalabilidad y otros factores.

Para finalizar se presentan las conclusiones obtenidas sobre el estudio de las interacciones entre aspectos y se presentan posibles trabajos futuros.

Capítulo 2

Programación Orientada a Aspectos e Interacciones entre Aspectos

En este capítulo se detallan los conceptos más importantes de la Programación Orientada a Aspectos (*AOP*) [18]. Este enfoque para la programcación, permite la modularización de *crosscutting concerns*, para lo cual introduce al aspecto como una unidad para representarlos.

La separación de *crosscutting concerns* requiere que los aspectos interactúen, de modo de lograr la funcionalidad final esperada. Sanen *et al.* [23] proponen una clasificación para estas interacciones. Las categorías de la misma se describen, dado que serán utilizadas para clasificar las interacciones halladas en el dominio de las *SM*.

Por último, se introducen las caracaterísticas más relevantes de *AspectJ* [16], extensión para *AOP* del lenguaje *Java*. La misma fue utilizada en la implementación del *software* de una *SM*, realizada para el estudio concreto de las interacciones.

2.1. Programación Orientada a Aspectos

Una aplicación orientada a objetos, se estructura en función de objetos que colaboran. Una de las mayores ventajas de la orientación a objetos, es que un sistema de *software* puede ser construido con una colección de clases, donde cada una tiene responsabilidades que están claramente definidas.

Sin embargo, existen funcionalidades de un sistema cuya implementación no puede ser modularizada y por lo tanto su implementación se encuentra distribuida en

diferentes clases, lo que se denomina *crosscutting concern*.

Ejemplos de *crosscutting concerns* no funcionales son: la funcionalidad de *logging*, *manejo de excepciones*, persistencia o inyección de código utilizado para análisis de rendimiento (*profiling*), etc. Estos *crosscutting concerns*, entre otros, fueron reportados por *Rashid et al.* [22] en el estudio sobre el uso de aspectos en la industria.

Estos *concerns* no pueden ser encapsulados de forma adecuada [18], por la tanto traen aparejados efectos negativos en el código fuente.
Uno de ellos es tener la implementación de cierta propiedad distribuida. Este problema se conoce como *scattered code* o código disperso.
Otro problema asociado, consiste en que un mismo módulo no sólo implemente el comportamiento referido a su responsabilidad, sino también otros comportamientos extrínsecos correspondientes a los *crosscutting concerns*. Esto se conoce como *tangled code* o código enmarañado.

Con el objetivo de tratar estos problemas, nace la Programación Orientada a Aspectos, presentada por *Kiczales et al.* [18] en 1997. La misma obtuvo rápidamente la atención de la comunidad científica, siendo señalada por el *MIT* en el año 2001, como una de las tecnologías claves de la próxima década [26].
Este enfoque pretende ser una solución al problema de encapsular *crosscutting concerns*, mediante módulos denominados aspectos.

La implementación de un aspecto, incluye el comportamiento del *crosscutting concern* y la especificación de los lugares en donde será ejecutado dicho código.
Un *join point* es un punto bien definido en la ejecución de un programa, por ejemplo: la llamada a un método, la asignación de una variable o el lanzamiento de una excepción.

Los aspectos se componen de *pointcuts* y *advices*. Un *pointcut* define un predicado sobre los *join points*. Por ejemplo, todas las llamadas al método m en el objeto A.
Un *advice* es equivalente a un método en *OOP*, contiene el comportamiento que implementa el *crosscutting concern*. Un *advice* es asociado a un *pointcut*. De esta manera, cuando en tiempo de ejecución un *join point* es capturado por un *pointcut*, el código del *advice* asociado es ejecutado.

Un *advice* puede ser definido para ser ejecutado antes, después o alrededor del *join point* seleccionado. En un *advice* que es ejecutado luego de la llamada de un método, es posible obtener el valor de retorno del mismo. Por otro lado, en un *advice* definido alrededor de un *join point*, el programador tiene la posibilidad de decidir si se ejecuta o no el *join point* capturado.

En algunos casos, para implementar un *crosscutting concern* de manera efectiva, puede ser necesario alterar la estructura estática del programa. *AOP* provee de un mecanismo denominado *inter-type declarations*, que permite afectar la estructura de

las clases. De esta manera, es posible desde un aspecto añadir métodos y atributos a clases ya existentes.

Definidos los aspectos, la composición de los mismos con el código de la aplicación, se realiza en un proceso denominado *weaving*. Aunque existen similitudes entre los lenguajes de AOP, una diferencia escencial reside en el momento en que se realiza este proceso. Esto puede ocurrir en tiempo de compilación o en tiempo de ejecución, lo que da lugar a la clasificación de lenguajes estáticos o dinámicos.

2.2. Interacciones entre aspectos

La orientación a aspectos es aceptada como un mecanismo efectivo para la separación de *crosscutting concerns* [17]. Esto se debe, entre otras cosas, al grado de expresividad que este enfoque brinda al programador.
Sin embargo, la característica de *obliviousness* entre aspectos [8] donde el programa base desconoce la existencia de los aspectos, hace que el comportamiento del sistema sea mucho más difícil de comprender.

Debido al hecho de que los aspectos afectan varios elementos de la aplicación, es probable que los mismos interfieran entre sí de alguna manera. Esto da lugar a que se produzcan interacciones entre los aspectos, necesarias para lograr la funcionalidad final esperada.

En algunos casos, las interacciones son deseables, y su presencia debe ser asegurada. Este es el caso de dependencias entre aspectos, donde un aspecto requiere de otro para funcionar.
En cambio en otros casos, si existen aspectos cuyo comportamiento es incopatible, es necesario evitar que la interacción se produzca.

2.2.1. Taxonomía de las interacciones

En el reporte técnico de *AOSD-Europe* sobre *Aspects Interactions* [23], los autores clasifican las mismas en las siguientes categorías: *dependency*, *conflict*, *mutex* y *reinforcement*.

Dependency: esta forma de interacción cubre las situaciones donde un aspecto A, depende de otro aspecto, B. En una situación de dependencia, si el aspecto B no se encuentra presente, el aspecto A no puede funcionar de manera correcta.

Conflict: la interacción de tipo *Conflict* captura las situaciones donde se producen interferencias semánticas entre aspectos. Dos aspectos, A y B, pueden funcionar correctamente si no se encuentran presentes al mismo tiempo. La presencia de ambos,

puede generar errores o comportamientos no deseados en la aplicación.

Mutex: esta forma de interacción, trata la exclusión mútua entre aspectos que proveen la misma funcionalidad. Al no ser aspectos complementarios, uno puede usarse, el otro no.

Reinforcement: es un tipo de interacción positiva, se produce cuando la presencia de un aspecto A, influye positivamente sobre otro, B. El aspecto A puede funcionar sin la presencia de B, pero de estar este último presente, A brinda funcionalidad extendida.

Esta taxonomía que describe las formas en que los aspectos interactúan, es utilizada en el capítulo 5 del presente trabajo, para clasificar las relaciones entre los *concerns* del dominio de las *SM*.

2.3. AspectJ

AOP no está ligada a un único lenguaje o paradigma de programación. Hoy en día existen varias extensiones de AOP [7, 16, 24] para diferentes lenguajes de programación [3, 9, 15]. Entre las mismas, se encuentra la implementación pionera: *AspectJ* [16], extensión para *AOP* del lenguaje Java [13].

AspectJ es el lenguaje de aspectos con mayor influencia y es el punto de referencia para otros lenguajes. Fue creado por un equipo de *Xerox PARC* [27], liderado por *Gregor Kiczales*, en el año 1997. Desde entonces, ha acompañado la evolución del lenguaje *JAVA*.

AspectJ ha sido utilizado en la industria, prueba de esto es que forma parte del *framework Spring* [1].
La herramienta más popular para la programación con *AspectJ* es el *plug-in AJDT* [6] para el *IDE Eclipse* [5]. El mismo ofrece un conjunto de sólidas herramientas, que facilitan el desarrollo de aplicaciones orientadas a aspectos

2.3.1. Sintaxis

AspectJ provee al usuario programador de una notación para definir aspectos, *pointcuts*, *advices* y *inter-type declarations*. A continuación se realiza una breve descripción de la sintaxis de esta extensión. Para mayor detalle se puede ver la guía de programación en línea de *AspectJ* [25].

Pointcuts

Los *pointcuts* pueden ser declarados en un aspecto, clase o interface. Como en las variables y métodos de *Java*, en un *pointcut* se puede especificar su nivel de visibilidad: *public*, *protected*, etc.
En *AspectJ*, los *pointcuts* pueden ser anónimos o nombrados. Los anónimos son declarados en el lugar donde se usan, por ejemplo al definir un *advice*. En cambio, los *pointcuts* nombrados son elementos reusables, dado que pueden ser referenciados de diferentes lugares. La figura 2.1 detalla la sintaxis de los *pointcuts* nombrados.

Código 2.1: Sintaxis de los *pointcuts* nombrados.

```
1 [access specifier] pointcut pointcut_name([args]) :
2                             pointcut_definition
```

A la izquierda de los ":" se encuentra el nombre del *pointcut* y del lado derecho su definición. Aquí mediante un predicado, se identifican los *join points*.

En el fragmento de código 2.2, se puede ver la definición de un *pointcut*, denominado `accountOperations`. El mismo captura la ejecución de todos los métodos de una clase llamada `Account`.

Código 2.2: Ejemplo de definición de un *pointcut* en *AspectJ*.

```
1 protected pointcut accountOperations() :
2                    call(* Account.*(..))
```

Advice

Como fue explicado, cuando en tiempo de ejecución un *join point* es capturado por un *pointcut*, el código del *advice* asociado es ejecutado. A comparación de los *pointcuts*, en *AspectJ* los *advices* sólo pueden definirse de manera anónima.
Para indicar el momento en que la funcionalidad del *advice* deba ser ejecutada, *AspectJ* provee las palabras claves: *before*, *after* y *around*. Mediante las mismas, se define el tipo de un *advice*.
La estructura de un *advice* puede dividirse en tres partes: la declaración del tipo del *advice*, la especificación del *pointcut* y el cuerpo.

Utilizando el *pointcut* `accountOperations()`, es posible definir un *before advice* que imprime un mensaje antes de la ejecución de cada método de la clase `Account`. Este ejemplo se puede ver en el fragmento de código 2.3, donde se utiliza la variable `thisJoinPoint`, disponible en el cuerpo del *advice*, la cual contiene información del *jointpoint* capturado.

Código 2.3: Uso de un *pointcut* nombrado en un *before advice*.

```
1 before() : accountOperations() {
2     System.out.println("Executing: " + thisJoinPoint);
3 }
```

El ejemplo anterior, también puede definirse utilizando *pointcuts* anónimos. En la figura 2.4, dicho *pointuct*, se define dentro de la estructura del propio *advice*.

Código 2.4: Uso de un *pointcut* anónimo en un *before advice*.

```
1 before() : call(* Account.*(..)) { advice body }
```

En la declaración de un *advice* es posible especificar que información de contexto está disponible para ser utilizada en el cuerpo. Por ejemplo, es posible acceder al objeto en ejecución sobre el cual el *jointpoint* es capturado.
En un *around advice*, la manera de indicar que se proceda con la ejecución del método que ha sido capturado, es mediante el uso de la palabra clave *proceed()*.

El fragmento de código 2.5, define un *around advice* sobre el método `Account.debit()`. En la definición del *advice* se especifican como parámetros: el objeto sobre el cual se ejecuta el método y el parámetro que este recibe. De esta manera, el programador podría evaluar una condición, para decidir si proceder o no con la ejecución del método.

Código 2.5: Ejemplo de definición de un *around advice*.

```
1 void around(Account account, float amount)
2      call(* Account.debit(float))
3      && target(account)
4      && args(amount)
5 {
6     if( _debitCondition ) {
7         proceed(account, amount);
8     }
9 }
```

Aspecto

El constructor `aspect`, cuya sintaxis detalla la figura 2.6, indica que el elemento que está siendo definido es un aspecto. Esta entidad tiene un nombre y una especificación de acceso. Los aspectos pueden ser abstractos, extender otros aspectos y también implementar interfaces. El cuerpo de un aspecto contiene el código en el cual se definen, entre otros elementos: *inter-type declarations*, *pointcuts* y *advices*.

Código 2.6: Sintaxis de un aspecto.

```
1 [access specification] aspect <AspectName>
2 [extends class_or_aspect_name]
3 [implements interface_list] {
4 ... aspect_body
5 }
```

El fragmento de código 2.7 define un aspecto `AccountLogging`, que herada de `AbstractLogging`. El cuerpo de este aspecto define los elementos previamente detallados.

Código 2.7: Ejemplo de un aspecto que define un *pointcut* y un *advice*.

```
1 public aspect AccountLogging extends AbstractLogging {
2
3   protected pointcut accountOperations() :
4                      call(* Account.*(..))
5
6   before() : accountOperations() {
7       System.out.println("Executing: " + thisJoinPoint);
8   }
9 }
```

Inter-type declarations

Mediante el uso de *pointcuts* y *advices* es posible modificar el comportamiento dinámico de un sistema. Para afectar la estructura estática del mismo, *AspectJ* provee de un mecanismo denominado *Introduction*. Este mecanismo permite alterar la estructura de clases, interfaces y otros aspectos.

Un ejemplo de uso de *Introduction* se puede ver en el fragmento de código 2.8. El aspeto declarado agrega a la clase `Account`, una variable en la línea 3 y un método en las líneas 5 a 7. Ambos componentes se definen de la misma manera que en una clase *Java*, con la única diferencia que se antepone el nombre de la clase destino.

Código 2.8: Definición de una variable y un método mediante *Introduction*.

```
1 public aspect AccountOperationsAspect {
2
3   private int Account._varName;
4
5   public int Account.getVarName() {
6       return _varName;
7   }
8 }
```

Weaving

En el compilador de *AspectJ*, la fase de *weaving* puede ser ejecutada en tres momentos diferentes:

- *Compile-time weaving*: es la forma utilizada cuando se dispone del código fuente de las aplicación.
- *Post-compile weaving*: es utilizado para realizar el proceso de *weaving* sobre archivos *.class* o archivos *JAR*.
- *Load-time weaving*: el proceso de *weaving* se produce en el momento en que un archivo *.class* es cargado en la *JVM*. Para poder ser utilizado este modo, el *run-time environment* debe proveer de *weaving class loaders*.

Capítulo 3

Concerns en el Dominio de las Slots Machines

En el presente capítulo, introducimos ciertos conceptos generales referidos al dominio de las máquinas tragamonedas.
En este sentido, resulta prioritario detallar requerimientos y funcionalidades que al final del capítulo serán identificados como *concerns* del dominio. Entre estos *concerns* se identificarán instancias de las interacciones que son el objeto de estudio del presente trabajo.

3.1. Dominio de las Slots Machines

Una *máquina tragamonedas*[1] es un dispositivo que puede ser encontrado generalmente en casinos y permite interactuar con un juego de apuestas.

Figura 3.1: Ejemplos de *SM*s electromecánicas.

[1]*Slot Machine*, la abreviatura *SM* será utilizada de aquí en más.

Estas máquinas, que eran completamente electromecánicas, han ido evolucionando para convertirse en la actualidad, en computadoras que ejecutan un *software* de alta complejidad.

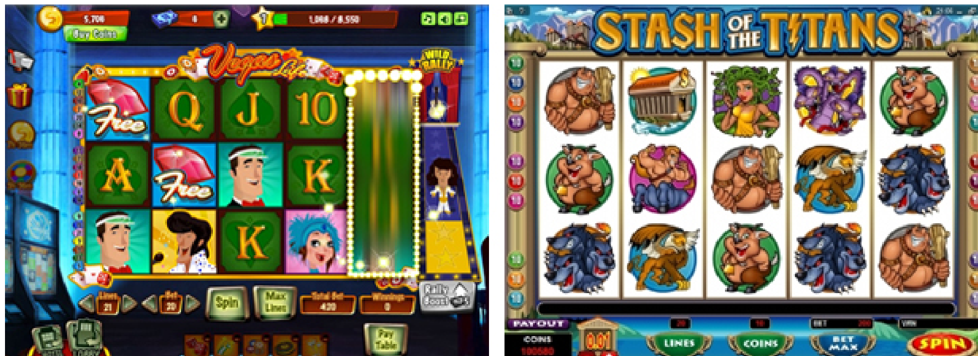

Figura 3.2: *SMs* modernas donde la interacción se realiza mediante una pantalla táctil.

En cuanto al *software* de una *SM*, es necesario y escencial explicar que se compone de un juego y un conjunto de subsistemas cuya funcionalidad varía desde la interacción con el *hardware* hasta la comunicación con servidores remotos.
A continuación se describen ciertas características del juego y de estos subsistemas, algunos de los cuales derivarán en *concerns* del dominio.

3.1.1. El juego

Los juegos se componen de 3 o 5 rodillos (*reels*), los cuales giran cuando el jugador presiona el botón de girar (*spin*). Para poder jugar se debe contar con créditos, que pueden ser ingresados de distintas formas de acuerdo al *hardware* con el que cuente la *SM*.
En las máquinas con juegos multilíneas es necesario seleccionar la cantidad de líneas sobre las que se quiere apostar (*selected paylines*) y por cada línea se apuesta una cantidad de créditos (*bet per line*). Estas acciones determinan el costo de la jugada: *bet per line* x *selected paylines*.

Al presionar el botón de *spin*, se descuenta el costo de la jugada de los créditos actuales y un sorteo determina de forma aleatoria la nueva posición de los *reels*.
Luego, se evalúa si hubo líneas ganadoras de acuerdo a la configuración de la tabla de pagos (*paytable*). Ésta indica cual es el multiplicador que se aplica a la apuesta por línea para cada combinación de símbolos que otorgue premio.
De haber créditos ganados, se suman a los actuales, los cuales pueden ser retirados (*cashout*) cuando el jugador lo determine.

3.1.2. Meters

El registro de toda la actividad que ocurre en una *SM* se almacena en un conjunto de contadores que son denominados *Meters*. Este conjunto deriva de diferentes documentos que incluyen regulaciones y recomendaciones de laboratorios de certificación [10].

Los valores de los *Meters* deben ser consistentes en todo momento y dado que almacenan información sensible, existe un procedimiento legal muy estricto para reiniciarlos.
El acceso a los mismos se realiza mediante una vista de la interfaz gráfica de la *SM*, a la cual se tiene acceso mediante el uso de llaves de seguridad.

La mayoría de los *Meters* son utilizados para actividades de contaduría y auditoría, ya sean realizadas por el propio casino o por entes reguladores externos. Todos los valores se almacenan digitalmente, pero aún en las máquinas más modernas, se cuenta con una serie de contadores electromecánicos que son exigidos por las regulaciones.

En una *SM* pueden encontrarse cientos de *Meters* que almacenan todo tipo de información, desde la cantidad de créditos apostados hasta el número de veces que se trabó una moneda.

Para dar cuenta de lo explicitado anteriormente, a continuación se detallan algunos *Meters* que se pueden encontrar en una *SM*:

- **Current Credits:** cantidad de créditos disponibles para jugar.
- **Total Drop:** total de créditos insertados.
- **Total Out:** total de créditos cobrados.
- **Game played:** cantidad de jugadas realizadas.
- **Game Won:** cantidad de juegos donde se ganó al menos un crédito.
- **Game Lost:** cantidad de juegos donde no hubo premios.

3.1.3. Hardware

En la actualidad una *SM* es una computadora con prestaciones industriales la cual cuenta con diferentes dispositivos de *hardware*.
En la figura 3.3 se indican los dispositivos físicos que se pueden encontrar en el gabinete de una *SM*. Luego, se describen las funcionalidades más relevantes del *hardware*.

Figura 3.3: Gabinete de una *Slot Machine*.

En un principio, para poder utilizar el juego se debe contar con créditos, que pueden ser ingresados de distintas formas. Cuando el jugador decide retirarse de la *SM*, debe retirar sus créditos. Los dispositivos que permiten estas operaciones son:

- **Coin Acceptor:** permite el ingreso de monedas o fichas, las cuales son convertidas a créditos según la denominación de la *SM*. Este dispositivo sólo acepta valores de monedas pre-configurados.

- **Bill Acceptor:** es un dispositivo utilizado para ingresar billetes, al igual que el *coin acceptor* sólo acepta valores pre-configurados.

- **Thermal Ticket Printer:** imprime un *ticket* cuando el usuario realiza la acción de *cashout*. Este *ticket* contiene un código de barras con información que es utiliza por servidores de los cuales se dará más detalle en la subsección 3.1.6.

- **Ticket Reader:** es otro medio para el ingreso de créditos a la *SM*. El sistema de *ticketing* posibilita el intercambio de créditos entre *SM*s. Es común encontrar el *bill acceptor* y el *ticket reader* en un mismo dispositivo físico.
- **Coin hopper:** permite cobrar los créditos en monedas o fichas.

En cuanto a los medios de pago, una *SM* puede tener configurados tanto el *coin hopper* como el sistema de *ticketing*. En el caso de tener ambos, el juego pagará con *tickets* cuando ya no cuente con monedas o fichas.

Por otro lado, dentro de los dispositivos relacionados con la seguridad y exigidos por las regulaciones se pueden encontrar **sensores** que permiten detectar la intrusión en la *SM* a diferentes niveles. Un ejemplo es el sensor de *main door*, que permite saber cuando es abierta la puerta del gabinete.
Toda la información de los sensores también es contabilizada en los *Meters*.

Con respecto al *hardware* utilizado por el personal del casino, es decir, los asistentes de juego, se pueden encontrar los siguientes dispositivos:

- **Tower Lamp:** dispositivo luminoso utilizado para llamar al asistente del casino.
  Esta lámpara puede encenderse por un evento generado por la *SM* , por ejemplo, cuando el jugador gana el *jackpot* o cuando hay un error que no permite continuar el juego.
- **Alarm Bell:** alarma sonora también utilizada para llamar a un asistente de juego en situaciones similares a las descriptas previamente.
- **Attendant Key switch:** este *switch*, que se activa mediante una llave especial, es utilizado en ocasiones donde la intervención humana es requerida.

Otra de las características de las *SM*s es que también se pueden encontrar **contadores electromecánicos**, estos son exigidos por las regulaciones para cierto subconjunto de *Meters*.
Para finalizar, podemos decir que una *SM* también cuenta con **DIP switches** los cuales están ubicados en un lugar seguro del gabinte y son utilizados por ejemplo, para cambiar el juego a modo *Demo*. Este modo es detallado en la sección 3.1.7.

Hardware Abstraction Layer
Se puede decir que la interacción entre estos dispositivos y el *core* de la *SM* se realiza mediante el intercambio de eventos. El cuadro 3.4 lista eventos de los dispositivos detallados previamente.

Dispositivo	Eventos frecuentes	Eventos no frecuentes
Bill Acceptor	Bill accepted Bill Stacked Bill Returned Bill Rejected	Bill jam Bill Stacker Full Stacker Open
Ticket Printer	Ticket printed	Printer out of paper
Attendant Key switch	Attendant arrived/leaved	
Sensors	Cabinet Open Cabinet Closed	CPU enclosure open/closed
Dip switches		Reset meters, Demo mode
Coin Hopper	Coin Paid	Hopper Empty
Coin Acceptor	Invalid Coin	Coin cheated

Figura 3.4: Eventos generados agrupados por dispositivo.

Estos eventos pueden ser condiciones de error que deben ser reportadas a otros sistemas y que además, en algunos casos, pueden generar cambios en la forma en que la *SM* se comporta.

La interacción con el hardware agrega un nivel más de complejidad que el *software* de una *SM* debe soportar.
En nuestra experiencia, es habitual que exista un componente de *software* denominado *HAL* (*Hardware Abstraction Layer*), el cual brinda un nivel de abstracción hacia los dispositivos físicos. Los eventos anteriormentente descriptos viajan desde y hacia la *HAL* para el control del *hardware*.

3.1.4. Program resumption

Una *SM* es un dispositivo electrónico, y como tal puede sufrir cortes de energía. La información sensible que se almacena dentro una *SM* requiere de un mecanismo que garantice la posibilidad de restauración de los datos y asegure la consistencia de la información.
Esta funcionalidad recibe el nombre de *Program Resumption*.

La información de una *SM* que debe guardarse incluye entre otros datos: la posición de los *reels* y los valores de los *meters*.
Este mecanismo es exigido por la mayoría de las regulaciones y agrega una alta complejidad al *software* debido a que cada cambio en ciertas variables del sistema debe guardarse antes de continuar con la ejecución del programa.

3.1.5. Game recall

Otro punto importante, es que las regulaciones también requieren una funcionalidad que recibe el nombre de *Game Recall*, cuyo objetivo es proveer el detalle de al menos los últimos 10 juegos realizados en la *SM*. Este detalle es utilizado por el personal de los casinos para resolver posibles conflictos sobre el comportamiento del juego.

Cada *Game Recall* debe contener entre otros datos: créditos apostados, símbolos que salieron sorteados, líneas ganadoras con el detalle de los créditos ganados por línea. Además, la información de entrada y salida de dinero entre cada juego debe quedar registrada en el *Game Recall*.
Otro punto importante, es que la lista de *Game Recall* también es parte de los datos que la funcionalidad de *Program Resumption* debe tener en cuenta para persistir.

3.1.6. Communication protocols

Es pertinente señalar que el *software* de las máquinas tragamonedas está conectado a sistemas de monitoreo en tiempo real y esto se debe principalmente a la necesidad de control sobre el movimiento de dinero.

Además de controlar el correcto funcionamiento de la *SM*, los sistemas de monitoreo son utilizados para las operaciones de contaduría dentro y fuera de los casinos. Para esto, existen diferentes normas sobre protocolos de comunicación, las cuales, utilizan diversas tecnologías entre las que se destacan el uso de puerto serie o conexiones *ethernet*.

Algunos protocolos funcionan con la técnica de *polling*, en la cual, un servidor de monitoreo pide los datos a la *SM* de manera regular. También existen protocolos basados en *web services*.

Es importante destacar que una *SM* debe tener la capacidad de poder trabajar con más de un *Communication Protocol* al mismo tiempo. Estos protocolos proveen a grandes rasgos la misma funcionalidad que incluye entre otras: la consulta de valores de los *Meters* ó configurar una *SM* de manera remota.

La especificación de los *Communication Protocols* define cientos de mensajes, estos pueden ser agrupados en cuatro categorías:

- **Configuración remota**
  Los mensajes en esta categoría son utilizados para configurar parte de una *SM* de forma remota. El mensaje *set current time* pertenece a esta clasificación y permite configurar la hora de la *SM*.

Otro ejemplo se da en una *SM* con múltiples juegos donde es posible seleccionar el juego actual de manera remota utilizando este tipo de mensajes.

- **Consulta de *Meters***
  Este grupo define mensajes y respuestas utilizados para consultar los *Meters* en una *SM*. Estos mensajes son los que se utilizan en los servidores para las actividades relacionadas con contaduría.

- **Sistema de *ticketing***
  En algunos casinos no se utiliza dinero para interactuar con la *SM*, sino que cuando el jugador ingresa a jugar compra cierta cantidad de créditos y se le otorga un *ticket*.

  Este *ticket* puede ser insertado en la *SM* y utilizando el código de barras que posee, se consulta con un servidor cuantos créditos deben acreditarse.

  Otra característica que posee el sistema, es que cuando el jugador realiza la operación de *cashout*, se imprime un ticket. Este a su vez, puede ser insertado en otra *SM* o puede ser cobrado por ventanilla.

  Lo que respecta al manejo de *tickets* es realizado por servidores dedicados y la comunicación de las *SM*s con dichos servidores se realiza utilizando mensajes que pertenecen a esta categoría.

- **Eventos en tiempo real y *Errors Conditions***
  Muchos de los eventos que ocurren durante el juego deben ser reportados a los sistemas de monitoreo. Un claro ejemplo de esto son los denominados *BeginGame* y *EndGame*.

  También de ser posible, deben ser reportados los eventos y *Errors Conditions* descriptos en la sección 3.1.3.
  Esto es opcional en la mayoría de las regulaciones, debido a que no todos los dispositivos físicos tienen la misma capacidad para la detección de errores y generación de *Errors Conditions*. Si fuesen detectados, deben ser enviados mediante los protocolos de comunicación, dado que es información útil para los casinos.

En el desarrollo del presente trabajo, se consideraron dos *Communication Protocols*. Uno de ellos es un protocolo propietario, del cual no se pueden dar detalles, el mismo será nombrado como PCP.
El otro protocolo elegido es abierto, basado en *web services*, denominado G2S [11].

3.1.7. Modo Demo

Es importante saber que las *SM* deben cumplir con regulaciones, las cuales varían dependiendo del país o estado donde deba ser instalada.
Existen entes llamados laboratorios de certificación, quienes luego de realizar un

amplio conjunto de pruebas, certifican que una *SM* es apta para ser instalada. Estas pruebas son muy extensas, abarcan ítems como probar completamente la *paytable* del juego o uso intensivo del *hardware*.

Para las pruebas de *software*, las *SM*s cuentan con un modo especial de funcionamiento denominado *Demo*. La principal característica del mismo es que el juego se comporta de manera particular, permitiendo simular el ingreso de créditos o forzar la salida de premios.
Esta funcionalidad es necesaria debido a que existen premios cuya probabilidad de salir es muy baja, al poder seleccionar que premio ganar en la próxima jugada se puede probar de forma completa la tabla de pagos del juego.

Una *SM* funciona en modo *Demo* dependiendo el estado de un *DIP switch* interno, pudiendo darse el caso de que un *SM* instalada en un casino, sea pasada temporalmente a este modo. Por esto, las regulaciones exigen que los eventos en modo *Demo*, no deben ser contabilizados por los *Meters* ni deben reportarse a los *Communication Protocols*.

3.2. Concerns identificados en el dominio a nivel de requerimientos

Dada las características descriptas de las *SM*s y nuestra experiencia de trabajo en el dominio, se identificaron los *concerns* que se detallan a continuación:

- **Game:** contiene la lógica de un juego de *reels*. El juego tiene créditos, los cuales se utilizan para realizar una apuesta sobre determinadas *paylines*, los *reels* giran y con el resultado del sorteo se determina la cantidad de créditos ganados.

- **Game Recall:** este *concern* mantiene la información más relevante de los últimos juegos, que en caso de ser solicitada por personal del casino o un ente regulador la misma este disponible.

- **Meters:** mantiene actualizada la colección de contadores que son obligatorios y relevantes en una *SM*. Por ejemplo, cantidad de dinero ingresado, cantidad de juegos realizados, entre otros.

- **Program resumption:** este *concern* es el que implementa el soporte para que ante un corte de energía en la *SM*, el juego pueda ser restaurado al estado previo al corte con todos los datos correspondientes.

- **Erros conditions:** ante determinadas condiciones la *SM* debe generar un error y actuar según especifican las regulaciones. Por ejemplo, deshabilitar el juego

si una puerta es abierta, o encender la *tower lamp* para llamar a personal del casino si se trabara una moneda.

- **Comunication protocols:** los protocolos son capaces de recibir comandos, que pueden requerir datos, por ejemplo, el valor de un *meter*. Estos comandos también puede cambiar el estado de la *SM*, por ejemplo el comando *set time*. Además, es opcional reportar estados y *errors conditions* de la *SM* según la regulación utilizada.
- **Demo:** el *concern* de *demo* es el que permite forzar determinados premios. Todas las acciones realizadas en este modo, no deben contabilizarse ni deben ser reportadas en los protocolos de comunicación.

Dada la complejidad del dominio, se pueden encontrar diferentes maneras de descomponer los *concerns*, la elección descripta fue realizada de manera tal que refleje las interacciones que son objeto de estudio de este trabajo.

Cada uno de estos *concerns* tiene un conjunto de requerimientos que representa una característica o propiedad del sistema. Algunos de estos *concerns* serán modelados como aspectos y otros como componentes, el detalle del diseño obtenido se presentará en el próximo capítulo.

3.3. Relaciones entre concerns

Las relaciones que se dan entre los *concerns* del dominio, se muestran mediante una notación *ad-hoc* en la figura 3.5.

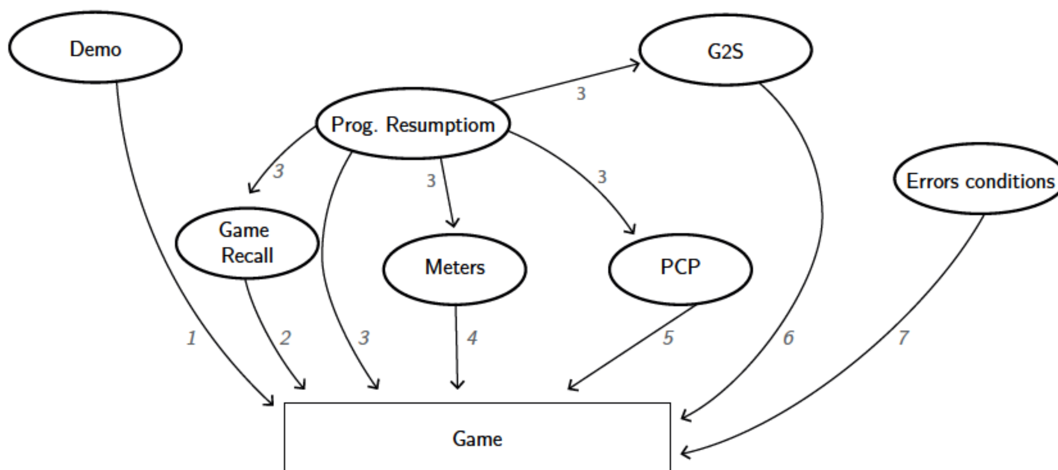

Figura 3.5: Relaciones *crosscutting* entre los *concerns* del dominio.

En esta figura el *base concern* de *Game* es representado por un rectángulo, mientras que los *crosscutting concerns* son representados por óvalos. Las relaciones de

crosscuts, representadas por flechas, son las que se detallan a continuación:

1. *Demo* a *Game*: El *concern* de *Demo* necesita alterar el comportamiento normal del juego para permitir la salida de premios.
2. *Game Recall* a *Game*: *Game Recall* requiere guardar datos a medida que el *concern* de *Game* va cambiando de estados.
3. *Program resumption* a *Game Recall*, *Game*, *Meters*, *PCP* y *G2S*: el *concern* de *Program resumption* controla la persistencia de datos y es por esto que tiene una relación con los *concerns* donde varían datos que deben ser restaurados.
4. *Meters* a *Game*: parte de los datos que contienen los *Meters* pertenecen al *concern* de *Game* y los mismos deben estar actualizados en todo momento.
5. *PCP* a *Game*: Varios eventos generados en el *concern* de *Game* requieren ser reportados mediante los protocolos.
6. *G2S* a *Game*: Al igual que *PCP*, este protocolo debe reportar cambios en el estado del juego.
7. *Error Conditions* a *Game*: dado que varias de las condiciones que pueden generar las *Errors Conditions* están dentro del comportamiento del *concern* de *Game*.

Capítulo 4

Implementación Orientada a Aspectos de una SM

Uno de los objetivos de este trabajo consiste en realizar una implementación que involucre varios *crosscutting concerns* funcionales del dominio para estudiar las interacciones entre los mismos y proponer mecanismos para su resolución.
En el presente capítulo se describen detalles de la implementación del *software* de una *SM* realizada utilizando *AspectJ*.

Los *base* y *crosscutting concerns* son los que se identificaron en el capítulo anterior. A continuación se presentan detalles de diseño, sobre los objetos y aspectos que componen cada *concern*. En el apéndice A se puede encontrar el detalle completo del modelo de objetos desarrollado.

Notación utilizada Los gráficos presentados en este capítulo están realizados en una variación del diagrama de clases UML. La figura 4.1 muestra un ejemplo de esta notación, donde:

- Un *concern* es representado por un *package*.
- Una aspecto es representado por un rectángulo que contiene el estereotipo «aspect» y el nombre del mismo en su interior.
- Una relación de *crosscuts* es representada por una línea punteada desde el aspecto hacia el elemento afectado.

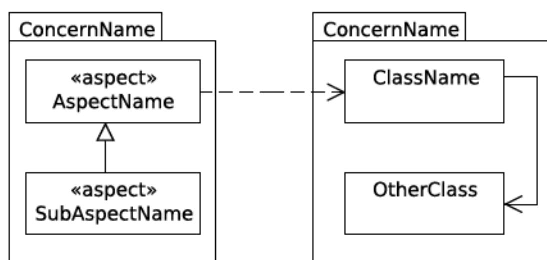

Figura 4.1: Ejemplo de la notación utilizada.

4.1. Game

Game es un *concern* funcional que no atraviesa otros *concerns* del dominio. La figura 4.2 muestra las clases de este *concern*, el cual implementa la funcionalidad principal de un juego de apuestas.

Las clases principales de este *concern* son: `Game`, `ReelsManager` y `BetManager`. Una instancia de la clase `Game` tiene créditos que se utilizan para realizar apuestas. Las mismas son administradas por un objeto `BetManager`, que puede calcular el costo de cada jugada. Este objeto además, tiene la capacidad de determinar los créditos ganados en cada juego. Este *concern* también cuenta con un objeto `ReelsManager`, que implementa por ejemplo, el sorteo de los símbolos de cada *reel* para una jugada.

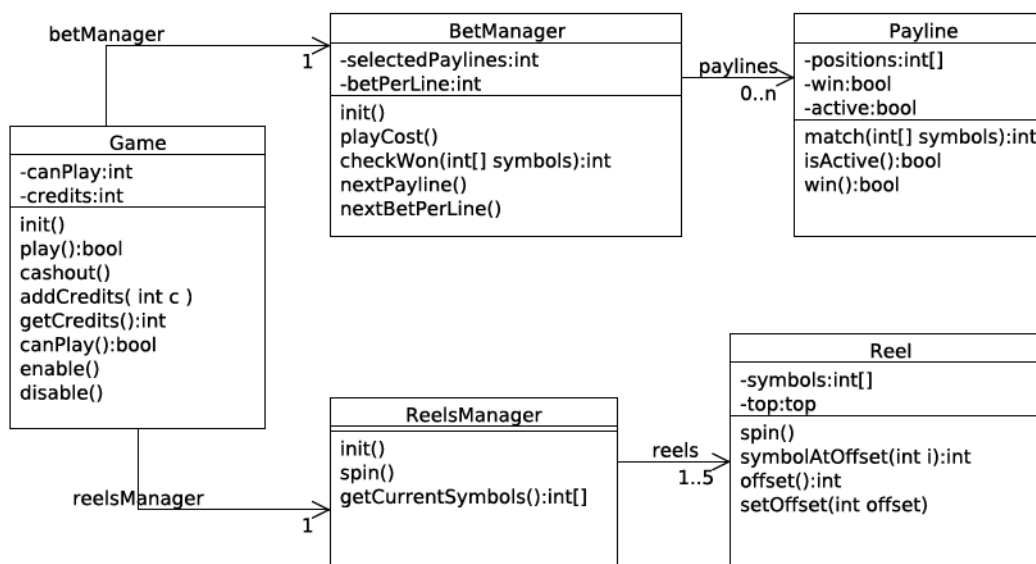

Figura 4.2: Clases del *concern* *Game*.

El siguiente fragmento de código muestra como interactúan estos objetos en el método `play()` de la clase `Game`. Previo a invocar el método `ReelsManager.spin()` para girar los *reels*, se delega en el objeto `BetManager` el cálculo del costo de la jugada (línea 5). Luego con los símbolos sorteados, obtenidos del `ReelsManager`, el manejador de apuestas controla si hubo líneas ganadoras.

Código 4.1: Método `play()` de la clase `Game`.

```
1 public class Game {
2   ...
3   public boolean play() {
4       ...
5       if ( subtractCredits( _betManager.playCost() ) )
6       {
7           _reelsManager.spin();
8           won = _betManager.checkWon(_reelsManager.
                getCurrentSymbols() );
9           ...
10      }
11      ...
12    }
13 }
```

4.2. HAL

En este *concern* se encuentran representados los dispositivos físicos con los que cuenta una *SM*. En el diseño obtenido no existen aspectos dentro de este *concern* y la interacción con los dispositivos se produce a través de una instancia de la clase `HAL`, cuya interfaz se puede ver en la figura 4.3. El detalle del resto de las clases de este *concern* se presenta en el Apéndice A.

En la implementación desarrollada un objeto de la clase `HAL` permite la interacción con pulsadores, detectar la apertura de la puerta del gabinete, cambiar el estado del *switch* de *demo*, guardar y obtener datos de una memoria no volátil.

Además de la clase `HAL` y las clases que representan los dipositivos, este *concern* define la interfaz `IHALSignalHandler`. Instancias de clases que implementen esta interfaz, se pueden registrar como *listeners* utilizando el método `registerHandler(IHALSignalHandler aHandler)`.
De esta manera, cuando un dipositivo envía a la `HAL` una *signal* determinada, se procede a notificar de la misma a todos los *listeners* registrados, como se puede ver en el fragmento de código 4.2.

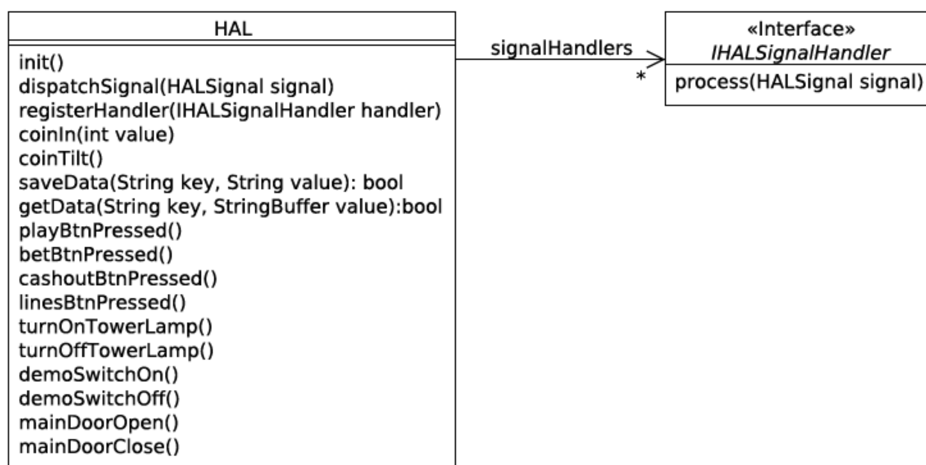

Figura 4.3: *API* de la clase HAL.

Código 4.2: Método `dispatchSignal()` de la clase `HAL`.

```
1 public class HAL {
2   ...
3   public void dispatchSignal(HALSignal signal) {
4       for(IHALSignalHandler handler : _signalsHandlers){
5           handler.process( signal );
6       }
7   }
8 }
```

4.3. Meters

El requerimiento principal de este *concern* es mantener actualizada una colección de contadores. Para esto se utiliza una jerarquía de aspectos, realizada en pro de una mejor modularización de los *advices*.
El aspecto de base `MetersAspect` tiene una referencia a un objeto `MetersManager`. Este objeto cuenta con una colección para almacenar los valores de los *meters*.
El aspecto `MetersAspect`, provee de métodos que son heredados y usados por los subaspectos. Estos métodos permiten obtener o modificar los valores almacenados del objeto `MetersManager`. Un ejemplo de esto, es el método `setMeter()`, cuya definición se muestra en el fragmento de código 4.3, el cual asocia un valor a un *meter*.

Código 4.3: Método `setMeter()` del aspecto `MetersAspect`.

```
1 public abstract aspect MetersAspect {
2   ...
3   protected void setMeter(Meter meter, int value) {
4       metersManager().setMeter(meter, value);
5   }
6 }
```

En esta subclasificación, cada subaspecto define *pointcuts* que atraviesan distintas clases. Los puntos de interés de dichos *advices*, son los métodos indicados en la figura 4.4, en donde el valor de un *meter* es leído o modificado.

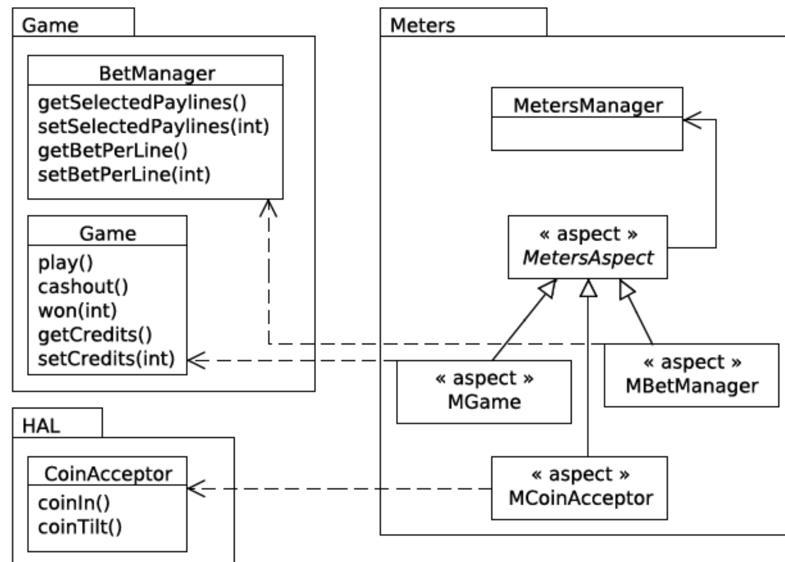

Figura 4.4: Clases y aspectos del *concern* de *Meters*.

El fragmento de código 4.4, muestra como se captura la modificación de las líneas seleccionadas durante una apuesta mediante un *after advice*. Utilizando el método heredado `setMeter()`, se actualiza el valor asociado al meter `LINES`, en el objeto `MetersManager`.

Código 4.4: *After advice* sobre el *pointcut* `BetManager.setLines()`.

```
1 public aspect MBetManager extends MetersAspect {
2   ...
3   after(int lines) : args (lines) &&
4                          BetManager.setLines(int){
5       setMeter(Meter.LINES, lines);
6   }
7 }
```

4.4. Program Resumption

La responsabilidad del *concern* de *Program Resumption* consiste en persistir ciertos datos de la *SM*, por ejemplo los valores de los *meters*. Para esto se utiliza una jerarquía de aspectos, como muestra la figura 4.5. Estos aspectos también restauran el estado de la *SM* durante el inicio del sistema.

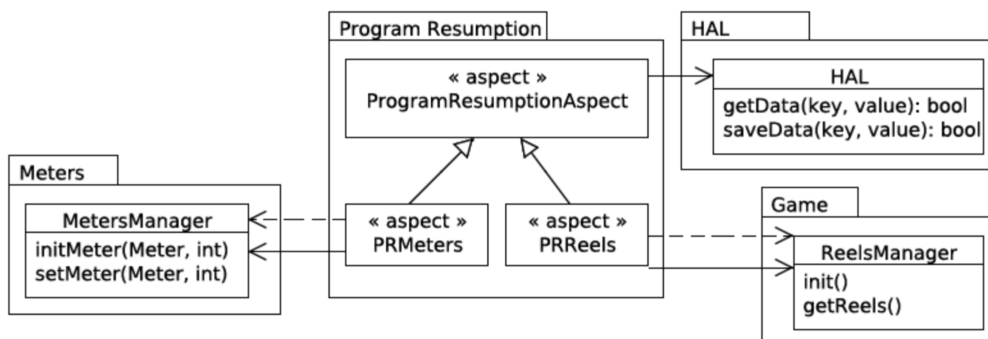

Figura 4.5: Clases y aspectos del *concern* de *Program Resumption*.

Los subaspectos utilizan métodos que el aspecto `ProgramResumptionAspect` provee, para obtener y guardar datos. La implementación de estos métodos, usa los servicios provistos por la *HAL*, para almacenar y recuperar datos de la memoria no volátil.

El listado 4.5 muestra parte del código del aspecto `Meters`. En las líneas 3 a 5 se define el *pointcut* que captura la actualización de un *meter*. Luego en las líneas 7 a 11, se define el *after advice*, que en la línea 10 persiste el cambio utilizando el método heredado.

Código 4.5: *After advice* utilizado para persistir los *meters*.

```
1 public aspect PRMeters extends ProgramResumption {
2   ...
3   pointcut setMeter(Meter meter, int value) :
4       args(meter, value) &&
5       execution(void MetersManager.setMeter(Meter, int));
6
7   after(Meter meter, int value) returning() :
8       args(meter, value) && setMeter(Meter, int)
9   {
10      save(meter.toString(), String.valueOf(value));
11  }
12 }
```

4.5. Game Recall

La responsabilidad de este *concern* consiste en guardar los datos relevantes de, al menos, los últimos 10 juegos, tal como indican las regulaciones. Los módulos principales, como se puede ver en la figura 4.6, son: un aspecto que afecta la clase `Game` y un objeto `GameRecallManager` que mantiene una colección de objetos `GameRecall`.

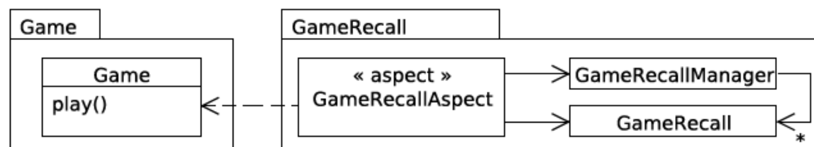

Figura 4.6: Clase y aspectos del *concern Game Recall*.

El *join point* de interés del aspecto `GameRecallAspect`, es el método `play` en el objeto `Game`. Como se puede ver en el fragmento de código 4.6, antes del sorteo que determina la nueva posición de los *reels*, el *advice* asociado guarda parte del estado de la *SM* en un objeto `GameRecall`.

Código 4.6: Implementación del aspecto `GameRecallAspect`.

```
1 before() : execution(boolean Game.play(..))
2 {
3   concerns.game.Game game = concerns.game.Game.get();
4   _gc = new GameRecall();
5   _gc.setBet(game.getBetManager().getBetPerLine());
6   _gc.setLines(game.getBetManager().getPaylines());
7   _gc.setPlayCost(game.getBetManager().playCost());
8   _gc.setCredits(game.getCredits());
9   _gc.setDate(game.getTime());
10 }
```

De manera similar al finalizar la jugada, mediante un *after advice*, se completan los datos en el objeto `GameRecall`. Este *advice* guarda las posiciones de los *reels* y los créditos ganados. Luego el objeto `GameRecall` es agregado a la colección del objeto `GameRecallManager`.

4.6. Errors Conditions

El *concern* de *Errors Conditions* implementa el tratamiento de diferentes eventos de acuerdo a las regulaciones. Por ejemplo, cuando se detecta la condición de *door*

open, se debe bloquear la *UI* del juego y los dispositivos de entrada, además de encender la *tower lamp* para indicar que se requiere la presencia de un asistente.

La figura 4.7 muestra las clases involucradas en este *concern*. Una *error condition* cuenta con un conjunto de acciones, las cuales deben aplicarse cuando la misma ocurre. A su vez, dichas acciones deben deshacerse cuando la *error condition* desaparece.

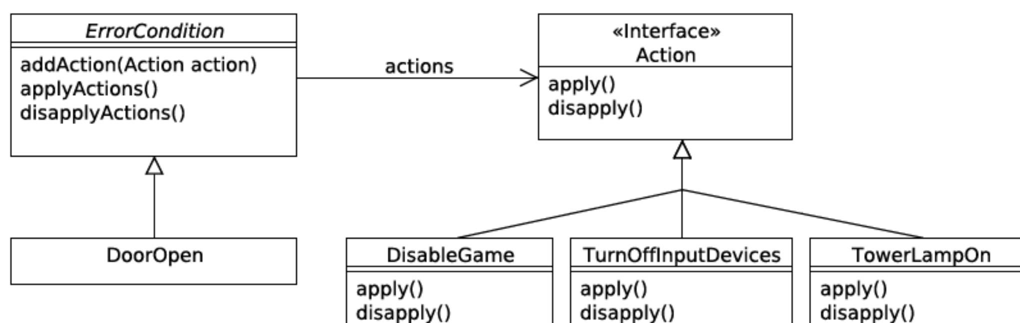

Figura 4.7: Clases del *concern* de *Error Condition*.

Este *concern* además, cuenta con un aspecto cuya implementación se muestra en el fragmento de código 4.7. Dicho aspecto, utiliza un *before advice* que está asociado al inicio del *software* de la *SM*. En el mismo, una instancia de la clase `HALSignalHandler`, se agrega como *listener* en la `HAL`.

Código 4.7: Registro de un *listener* en la `HAL`.

```
1 public aspect ErrorConditionsAspect {
2   before() : execution(* Game.init()) {
3       HAL.get().addHandler(new HALSignalHandler());
4   }
5 }
```

El *handler* creado por el aspecto es notificado cuando una *signal* ocurre. Este *handler* crea en caso de ser necesario, una `ErrorCondition` para su tratamiento. El fragmento de código 4.8 muestra parte de la implementación de la clase `HALSignalHandler`. El método `process(HALSignal)`, evalúa en un *switch* la *error condition* recibida. Esto es posible dado que la misma pertenece a un tipo enumerativo. En cada rama del *switch*, líneas 5 a 10, se definen las señales a las cuales debe asociarse una *error condition*. La misma es creada y sus acciones son aplicadas según el caso.

Código 4.8: Creación de la *error condition* DoorOpen.

```
1 public class HALSignalHandler implements IHALSignalHandler
2 {
3   @Override
4   public void process(HALSignal signal) {
5       switch (signal) {
6           case MAIN_DOOR_OPEN:
7               new DoorOpen().applyActions();
8               break;
9           case MAIN_DOOR_CLOSE:
10          ...
11  }
12 }
```

4.7. G2S y PCP

Estos *concerns* encapsulan el comportamiento de los protocolos de comunicación, además de su interacción con la *SM*. Un protocolo permite que la *SM* reporte información y reciba comandos. La ocurrencia de determinados eventos, debe ser reportada a los sistemas de monitoreo. Un ejemplo de esto, es el comienzo de un juego, indicado por el evento *Begin Game*. Por otro lado, una *SM* puede recibir comandos, los cuales pueden consultar un valor como es el caso del comando *GetMeter* o alterar el estado de la *SM*, como lo hace el comando *SetTime*. En la figura 4.8 se pueden ver las clases involucradas en este *concern*.

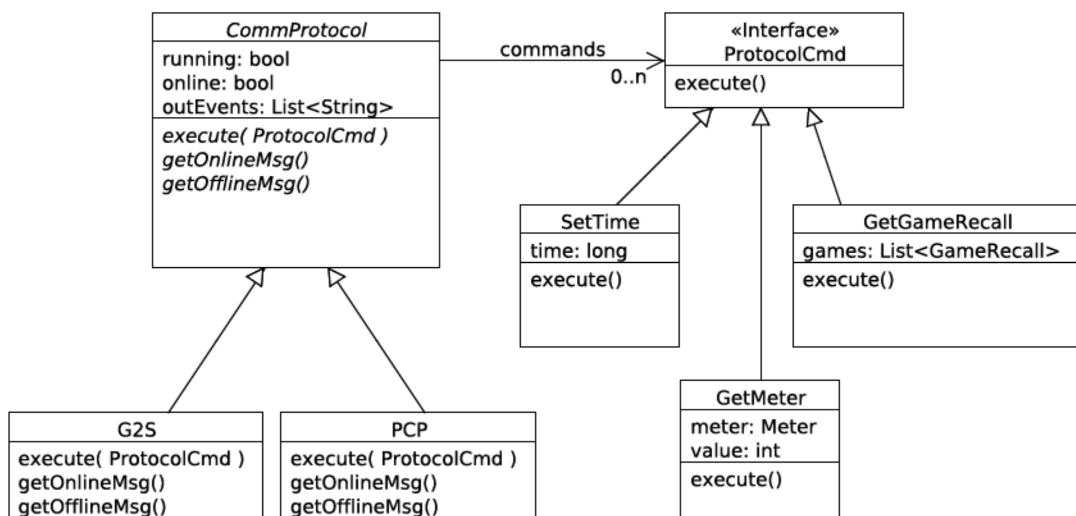

Figura 4.8: Clases de los *concerns* *G2S* y *PCP*.

Cada comando que puede ser recibido por una *SM* implementa la interfaz `ProtocolCmd`. La misma define el método `execute()`, el cual contiene el comportamiento asociado al comando. Un ejemplo de esto es la clase `SetTime`, que tiene como acción asociada modificar la hora de la *SM*.

En esta implementación, la jerarquía de comandos es compartida por ambos protocolos. Esto se debe a que los protocolos brindan la misma funcionalidad. Los protocolos solo difieren en la forma en que los mensajes son serealizados o los intervalos de tiempo utilizados para el intercambio de mensajes.

En este *concern* existe una jerarquía de aspectos que se ve en la figura 4.9. Estos aspectos son utilizados para capturar eventos ocurridos en otros *concerns*, por ejemplo el inicio de un juego en la clase `Game`.
El aspecto de base, define los *pointcuts* de interés para los protocolos. Cada subaspecto implementa los *advices* correspondientes, con el objetivo de notificar a los sistemas de monitoreo en tiempo real, de los cambios en la *SM*.

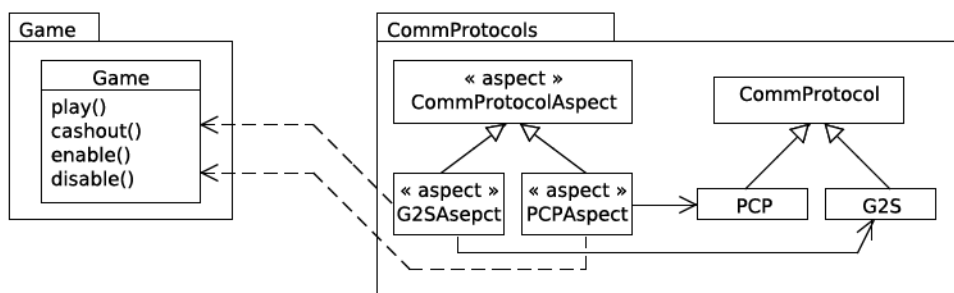

Figura 4.9: Jerarquía de aspectos utilizada para notificar cambios en la *SM*.

El fragmento de código 4.9, muestra el *before advice* del aspecto `PCPAspect`, que notifica del comienzo de una jugada a los sistemas de monitoreo. Para esto utiliza el método heredado `send()`.

Código 4.9: Notificación del evento *Begin Game* en el protocolo *PCP*.

```
1 public aspect PCPAspect extends CommProtocolAspect
2 {
3   ...
4   before() : Game.play()
5   {
6       send( new BeginGameEvent() );
7   }
8 }
```

4.8. Demo

La funcionalidad principal de este *concern* es poder seleccionar que premio ganar en la próxima jugada. Esto requiere alterar el comportamiento del *core* de la *SM* para permitir seleccionar el premio otorgado por el juego, para esto se debe afectar la forma en la que se determina la posición de los *reels*. Por esto, el *concern* de *Demo* afecta al *concern* de *Game*, como se puede ver en la figura 4.10.

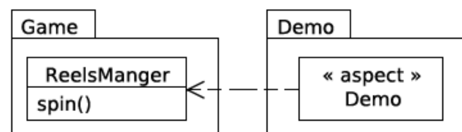

Figura 4.10: El aspecto `Demo` afecta al objeto `ReelsManager`.

En el aspecto `Demo`, la ejecución del método *ReelsManager.spin()* es capturada por un *around advice*, para alterar el sorteo de las posiciones de los *reels*. Cuando el modo *demo* se encuentra activo, se fuerza la salida de un determinado *outcome* y en caso contrario se procede con la ejecución normal del método.

Dado que el proceso de *weaving* se realiza en tiempo de compilación y el aspecto `Demo` está siempre presente, el mismo mantiene en una variable su estado de activación. Esta variable cambia de valor según el estado del *switch* de *demo* en la `HAL`. Para la actualización de la misma se cuenta con dos *advices* sobre la clase `HAL`, uno sobre el *pointcut* `demoSwitchOn()` y otro sobre `demoSwitchOff()`.

Código 4.10: *Advices* utilizados para modificar el estado del modo *demo*.

```
1 public aspect Demo {
2   private boolean _on = false;
3   ...
4   after() returning(): execution(void HAL.demoSwitchOn())
5   {
6       _on = true;
7   }
8   after() returning(): execution(void HAL.demoSwitchOff())
9   {
10      _on = false;
11  }
12 }
```

La implementación de esta funcionalidad se puede ver en el fragmento de código 4.11. La variable `_outcome` contiene las posiciones de los *reels* que permiten ganar el premio seleccionado por el usuario mediante la *UI* de la *SM*. Mediante el *around advice* se evita la ejecución del método `ReelsManager.spin()` y se altera la posición de cada uno de los *reels*.

Código 4.11: *Advice* utilizado para forzar un *outcome* en modo *demo*.

```
1 void around() : execution(void ReelsManager.spin())
2 {
3   if (_on) {
4       ArrayList<Reel> reels = Game.get().getReelsManager()
5                                         .getReels();
6       for (int i = 0; i < reels.size(); i++) {
7           reels.get(i).setOffset( _outcome[i] );
8       }
9   } else {
10      proceed();
11  }
12 }
```

Capítulo 5

Interacciones entre Aspectos

En el capítulo anterior se describieron detalles de diseño e implementación *AO* del *software* para una *SM*. Como fue descripto en el capítulo 2, entre los aspectos generan interacciones, las cuales son el objeto de estudio de este trabajo. Ejemplos de las mismas se pueden encontrar entre los *concerns* descriptos anteriormente.

Existe un conflicto entre *Demo* y *Meters*, dado que los *meters* no deben ser alterados en modo *demo*.
Una interacción de tipo *Mutex* se da cuando dos o más *Communication Protocols*, que proveen la misma funcionalidad, están activos al mismo tiempo.
Existe una interacción de tipo *dependency* entre los *Communication Protocols* y el *concern* de *Meters*, por el hecho de que estos deben ser reportados por los protocolos.
Por último una interacción de tipo *reinforcement* se da cuando una *SM* tiene la capacidad de detectar *Errors Conditions*. Estás son utilizadas por los *Communication Protocols* para ser reportadas en tiempo real, ampliando la funcionalidad de los protocolos.

En este capítulo se detallan los mecanismos desarrollados para el tratamiento estas interacciones. Los mismos permiten lograr el comportamiento deseado para el *software* de una *SM*.
Sobre el final, se presenta un análisis de los mecanismos desarrollados. En el mismo se discuten ventajas y desventajas en lo referido a generalización, modularización, escalabilidad y otros factores de la Ingeniería de *Software*.

5.1. Interacciones identificadas en el dominio de las SM

A partir del estudio de los requerimientos y considerando la taxonomía de interacciones de *Sanen et al.* [23], las relaciones *crosscutting* e interacciones que se dan entre los *concerns* del dominio, se muestran mediante una notación *ad-hoc* en la figura 5.1.

Este gráfico completa el presentado en 3.3, donde se indicaron las relaciones entre los *concerns*.
Esta es una selección de las interacciones más representativas, hay otras que no fueron incluídas en la figura para evitar sobrecargar el gráfico.

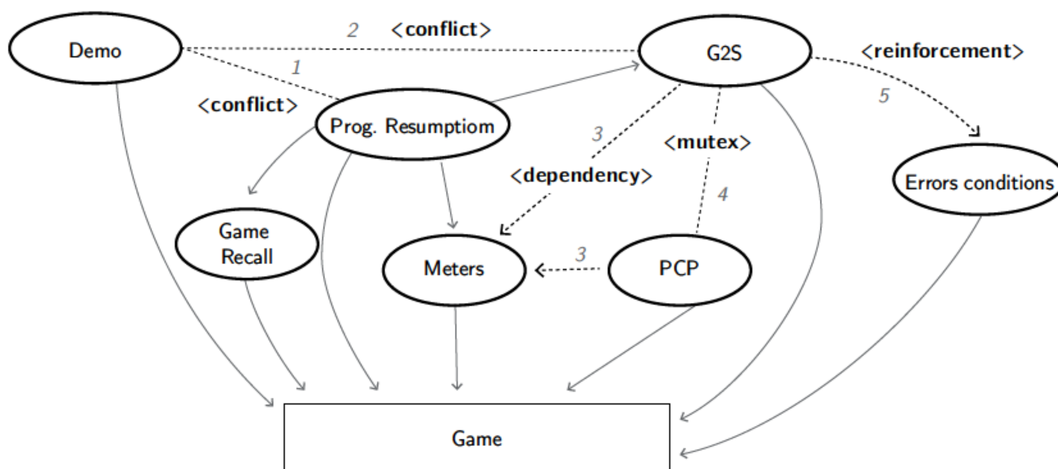

Figura 5.1: Relaciones e interacciones entre los *concerns* del dominio.

En esta figura el *base concern* de *Game* es representado por un rectángulo, mientras que los *crosscutting concerns* son representados por óvalos. Las relaciones de *crosscuts* se indican con líneas sólidas, mientras que las interacciones están representas por líneas punteadas. Las interacciones de la figura 5.1 son las siguientes:

1. **Conflict** entre *Demo* y *Program Resumption*: dado que los datos alterados en modo *Demo* no deben ser persistidos.
2. **Conflict** entre *Demo* y *G2S*: se genera por el hecho que ambos *concerns* no pueden estar activos al mismo tiempo, dado que uno de los requerimientos del modo *demo* es que los eventos no sean reportados a los protocolos de comunicación.
3. **Dependency** de *G2S* y *PCP* sobre *Meters*: una de las características de los protocolos de comunicación es que deben reportar el estado de los *meters*. Para esto dependen del correcto funcionamiento del *concern* de *Meters* para enviar datos válidos.
4. **Mutex** entre *G2S* y *PCP*: esta interacción se genera porque ambos protocolos implementan una misma funcionalidad. Por ejemplo, con ambos protocolos activos, de recibirse un comando que cambia la hora de la *SM* de manera remota, podrían producirse inconsistencias en los datos guardados por la *SM*.
5. **Reinforcement** de *G2S* con *Error Conditions*: dado que el requerimiento de reportar las *Errors Conditions* no es obligatorio, se da una interacción de tipo *reinforcement* cuando estos datos están disponibles y pueden ser utilizados por los protocolos para el reporte de los mismos en tiempo real.

En las próximas secciones se explican los mecanismos desarrollados para el tratamiento de las siguientes interacciones:

- Tipo **Mutex**: entre los *Communication Protocol G2S* y *PCP*.
- Tipo **Conflict**: entre *Demo* y: *Communication Protocols*, *Program Resumption*, *Game Recall* y *Meters*.
- Tipo **Dependency**: *Communication Protocols* sobre *Meters*.
- Tipo **Reinforcement**: *Error Condition* a *Communication Protocols*.

5.2. Mutex

Esta forma de interacción trata la exclusión mútua entre aspectos, la cual se presenta si dos aspectos brindan una misma funcionalidad. Ante este caso, es necesario asegurar que sólo se utilice uno de los aspectos a la vez. De no tratar esta interacción, la ejecución de ambos aspectos, puede generar un comportamiento no deseado en el sistema.

El *software* de las *SM* puede tener activo más de un protocolo a la vez. Como se explicó en la sección 3.1.6, los diferentes protocolos implementan funcionalidad similar. Estas operaciones se pueden dividir en dos grupos: por un lado las que obtienen datos de la *SM* y por otro lado, las que configuran su estado. La ejecución de estas últimas, en protocolos diferentes, puede ser la causa de errores e inconsistencias. Ejemplos concretos de operaciones que pueden generar estos problemas son:

Set time: este comando permite configurar de manera remota la hora de una *SM* y es enviado periódicamente por los sistemas de monitoreo, utilizando diferentes protocolos. En ciertas ocasiones, puede darse el caso de que los servidores que envían los comandos no estén sincronizados. Si la *SM* está utilizando más de un protocolo a la vez, puede generarse información inconsistente. Por ejemplo, el listado de *game recall*, podría tener eventos desordenados en el tiempo.

Set progressive: muchos juegos cuentan con una característica llamada "pozo acumulado" (progressive). Esta funcionalidad permite que un grupo de *SMs*, aporten un porcentaje de las apuestas a un pozo común, pudiendo cualquiera de ellas pagar el pozo acumulado como premio mayor.
Para la implementación de esta característica, también son utilizados los protocolos de comunicación. Si se utiliza más de un protocolo a la vez, se pueden generar varios problemas. Un ejemplo es el valor del pozo por el que una *SM* participa, el cual se muestra en un *display* de *leds* sobre el gabinete. Dicho valor es recibido frecuentemente via los protocolos, dado que varía con el aporte de

cada juego. De recibir el valor del pozo de diferentes protocolos, la *SM* estaría mostrando al usuario información incorrecta.

Como se detalló en el capítulo anterior, la funcionalidad de los protocolos *G2S* y *PCP* se implementó con aspectos. La operaciones como *set time* y *set progressive* generan una interacción de tipo *Mutex*.

5.2.1. Implementación de la Interacción

Como fue explicado, ejecutar operaciones que alteren el estado de la *SM* desde protocolos diferentes puede ser la causa de errores o inconsistencias. Una opción para tratar este problema, es asignarle permiso a un protocolo para realizar este tipo de operaciones. Desafortunadamente, esto resulta ser una solución demasiado global para los requerimientos de nuestro sistema, dado que es necesario que distintos ítems de configuración sean modificados por diferentes protocolos. Por ejemplo, si el protocolo *PCP* obtiene el permiso para ejecutar el comando *set time*, esto no debe implicar que la obtenga para el resto de las operaciones.

Por lo tanto, es necesario que la solución brinde un nivel de granularidad más fino, que permita controlar la exclusión mútua **por operación**. La solución que se propone para lograr esto consiste en:

1. Identificar el *joint point* (operación), que al ser ejecutado desde protocolos diferentes, produce el conflicto.
2. Capturar dicha ejecución mediante un *around advice*.
3. Evaluar una regla en dicho *advice*, para decidir si proceder o no, con la ejecución del método.

La figura 5.2 muestra los módulos que componen el mecanismo propuesto. Se definió una jerarquía de aspectos `Mutex`, que tiene como colaborador un objeto `LockType`, que implementa el patrón de diseño *Strategy* [12]. Este último, especifica una regla particular, la cual permite determinar si un objeto puede obtener el *lock*.

El aspecto abstracto `Mutex` define un *around advice*, el cual es asociado al *abstract pointcut* `tryLock()`. En dicho *advice* se evalúa la regla definida en el objeto `LockType` asociado. Por cada caso donde sea necesario un *mutex*, se subclasificará el aspecto definiendo un *pointcut* concreto.

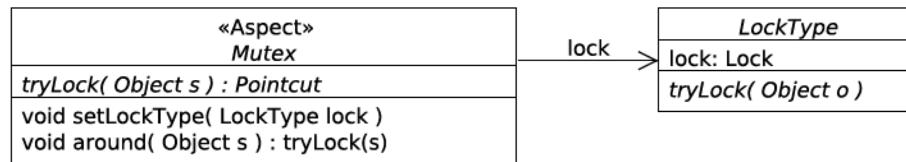

Figura 5.2: Clases abstractas del mecanismo.

La implementación del aspecto abstracto `Mutex` se puede ver en el fragmento de código 5.1. En la línea 3 se define la variable `_lock` que contendrá la instancia de `LockType`. Luego, en la línea 4 se declara un *abstract pointcut* al cual se asocia el *around advice* definido entre las líneas 6 y 10. En el cuerpo del mismo se delega en el objeto `LockType` la decisión respecto de si proceder o no con la ejecución del método.

Código 5.1: Código genérico de la clase `Mutex`.

```
1 public abstract aspect Mutex {
2
3   private LockType _lock;
4   protected abstract pointcut tryLock( Object s );
5   ...
6   void around( Object s ) : tryLock(s) {
7       if (_lock != null && _lock.tryLock( s )) {
8           proceed( s );
9       }
10  }
11 }
```

Por otra parte, las subclases de la jerarquía `LockType` deben implementar el método `tryLock( Object o )`. Como se ejemplifica más adelante, cada subclase implementa un lógica particular. De esta manera una instancia de esta jerarquía será usada como se vio en el listado de código 5.1, para determinar si proceder o no con la ejecución del método.

Para instanciar el mecanismo, el programador necesita subclasificar el aspecto `Mutex` y realizar dos acciones:

1. Definir el *pointcut* que captura el *join point* generador del conflicto y que necesita ser controlado por un *mutex*.
2. Instanciar el `LockType` deseado.

A continuación, se detalla como este mecanismo genérico, fue instanciado para resolver el conflicto de la operación *SetTime*.

5.2.2. Instanciación del mecanismo para el comando Set Time

Para resolver este conflicto, fueron subclasificados el aspecto `Mutex` y la clase `LockType`, como se puede ver en la figura 5.3. La estrategia de *locking* elegida para tratar el conflicto de la operación *set time* se denominó `FirstLockerKeepsLock`. La misma define que sólo el primer objeto en capturar el *lock* puede volver a obtenerlo.

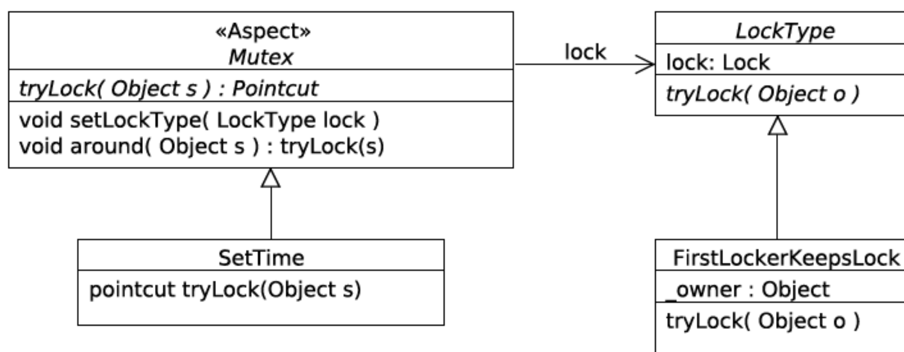

Figura 5.3: Instanciación del mecanismo para el comando *SetTime*.

El fragmento de código 5.2 muestra la implementación del método `tryLock()` de la clase `FirstLockerKeepsLock`. La primera vez que un objeto adquiere el *lock*, pasa a ser el dueño del *mutex* (línea 7). En las próximas capturas de *join points*, se compara la identidad de los objetos para determinar si quien intenta obtener el *lock* es el dueño.

Código 5.2: Método `tryLock` de la clase `FirstLockerKeepsLock`.

```
1 @Override
2 public boolean tryLock( Object o ) {
3   synchronized (lock()) {
4       if (_owner != null) {
5           return (_owner.equals( o ));
6       }
7       _owner = o;
8   }
9   return true;
10 }
```

Definida la estrategia a utilizar, se definió el aspecto `SetTime` que hereda de `Mutex`, en el cual:

1. Se concretizó el *abstract pointcut*, definiendo el mismo sobre el *join point*

`ProtocolCommand.execute()`.

2. Se especificó que `FirstLockerKeepsLock` es el tipo de estrategia elegida.

La codificación de estas acciones se puede ver en el listado de código 5.3, que muestra la implementación del aspecto `SetTime`. En la línea 3 crea la instancia de `FirstLockerKeepsLock` y mediante el método `setLockType()` se indica que esta es la regla a evaluar.
Luego entre las líneas 6 y 9 se concretiza el *pointcut*, indicando que el *Mutex* debe evaluarse sobre el método `ProtocolCommand.execute()`, de la subclase `Timestamp`.

Código 5.3: Código del aspecto *SetTime*.

```
1 public aspect SetTime extends Mutex {
2   {
3       setLockType(new FirstLockerKeepsLock());
4   }
5
6   protected pointcut tryLock( Object s ) :
7               call( void ProtocolCommand.execute() ) &&
8               this( s ) &&
9               target( Timestamp );
10 }
```

La figura 5.4 muestra en un diagrama de secuencia el trabajo del mecanismo para el caso donde el protocolo *G2S* tiene el *lock* para el comando *SetTime*.

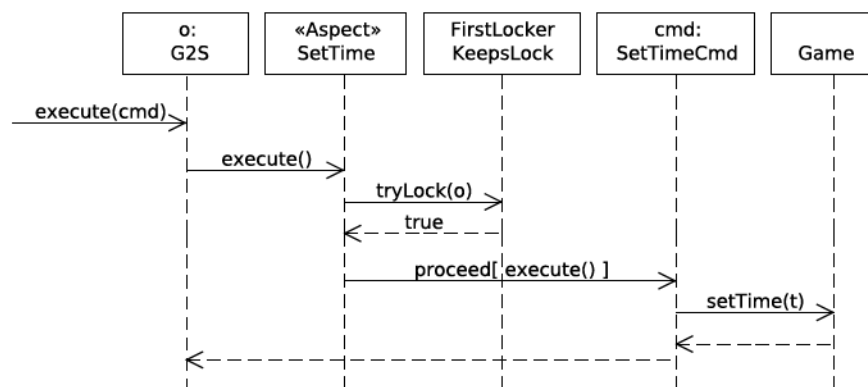

Figura 5.4: Diagrama de secuencia para la ejecución del comando *SetTime*.

De esta manera queda completa la instanciación del mecanismo. Ante un nuevo caso de interacción que requiera de un *Mutex*, basta con subclasificar el aspecto `Mutex`, para concretizar el *pointcut* y especificar el *LockType*.

5.2.3. Estrategias alternativas

Otra estrategia de *lock* implementada se denominó `ProtocolOnlineKeepsLock`. En la misma, se evalúa si el protocolo que posee el *lock* está en línea. Este tipo de *lock*, muestra que es posible definir estrategias más complejas si se conocen detalles del dominio.

La figura 5.5 muestra un ejemplo de uso de esta estrategia, donde el protocolo *PCP* posee el *lock*, pero se encuentra fuera de servicio. Cuando se recibe un comando desde el protocolo *G2S*, este obtiene el *lock* y el comando es ejecutado.

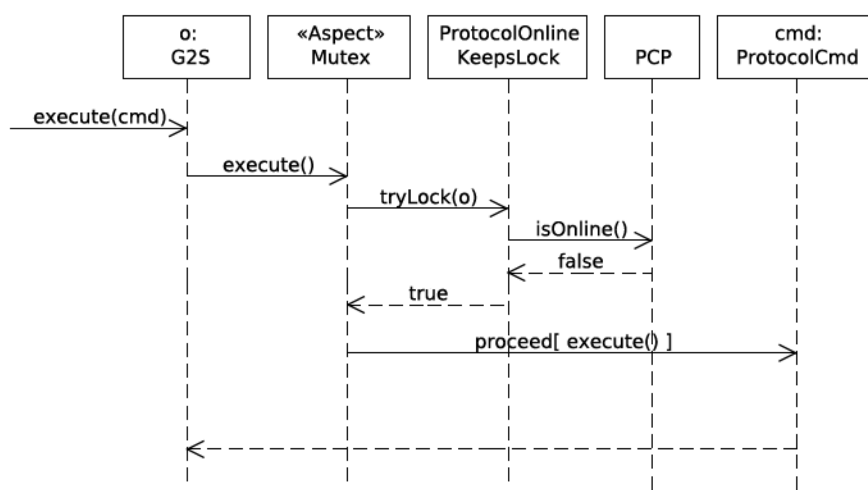

Figura 5.5: Diagrama de secuencia de la estrategia `ProtocolOnlineKeepsLock`.

5.3. Conflict

La interacción de tipo *Conflict* captura las situaciones donde se producen interferencias semánticas entre aspectos. Esto ocurre cuando un aspecto deja de funcionar correctamente ante la presencia de otro, debido a que estos aspectos representan requerimientos contrapuestos.
La necesidad de un mecanismo para tratar este tipo interacción, surge con el objetivo de asegurar que los aspectos que se encuentran en conflicto, funcionen de manera correcta.

Como se vio en el 3.1.7, en el dominio de las *SM* el *concern* de *Demo* atraviesa a:

- **Game:** dado que en modo *demo* debe ser posible forzar la salida de premios.

- **Program Resumption:** los cambios en modo *demo* no deben ser persistidos en este modo.

- **Meters:** las regulaciones exigen que los cambios en los *meters* en este modo no deben mantenerse al volver al modo normal.

- **Game Recall:** idem a *Meters*.

- **Communication Protocols:** para este *concern*, las regulaciones indican que una *SM* en modo *demo* debe reportase como fuera de servicio hacia los *backends*.

Estos *concerns*, entran en conflicto con el *concern* de *Demo* cuando este se activa. Esto genera la necesidad de contar con un mecanismo, que permita resolver estos conflictos, alterando el comportamiento de dichos *concerns*.

5.3.1. Tratamiento de la interacción

Como se detalló en 4.8, el aspecto `Demo` implementa la funcionalidad principal del *concern* de *Demo*. Dicho aspecto es el responsable de alterar la forma en que se realiza el sorteo aleatorio de las posiciones de los *reels*, de forma tal de que se pueda seleccionar que premio ganar en la próxima jugada. El aspecto utiliza una variable que representa el estado del *switch* de *demo*, la cual se actualiza desde dos *advices* que afectan la clase `HAL`, uno sobre el *pointcut* `demoSwitchOn()` y otro sobre `demoSwitchOff()`.

Dado que *AspectJ* es un lenguaje donde el proceso de *weaving* se realiza en tiempo de compilación, no es posible desvincular un aspecto en *run time* como en un lenguaje dinámico. Por ejemplo, **no es posible** evitar el conflicto con el *concern* de *Meters* **desinstalando** el aspecto `MetersAspect`.

En la implementación del *software* de la *SM*, se decidió que el aspecto `Demo` ademas de implementar la funcionalidad principal del *concern*, implemente los mecanismos para tratar las interacciones que producen conflicto. Estas son como fue descripto, con los *concerns*: *Meters*, *Program Resumption*, *Communication Protocols* y *Game Recall*.

Es necesario que el mecanismo para resolver cada interacción, conozca detalles de la implementación del *concern* con el que debe tratar, para poder anular parte de su funcionalidad. La idea general para resolver estos conflictos, se basa en encontrar un *join point* que de ser evitada su ejecución, se anule la funcionalidad del *concern*. A continuación se describen los detalles de implementación de cada caso.

5.3.2. Tratamiento de la interacción entre los concerns: Demo y Meters

Las regulaciones indican que los valores de los *meters* que se modifiquen en modo *demo* no deben verse reflejados como cambios al salir de este modo. En cambio, no especifican que acción realizar con los valores actuales de los *Meters* cuando se pasa a modo *demo*.
En la implementación realizada, se tomo la decisión de reiniciar los valores al activarse el modo *demo* y restaurar los mismos al volver a modo normal.

Como se detalló en 4.3, la implementación del *concern* de *Meters* utiliza una instancia de la clase `MeterManager` para guardar los valores de los *meters*. Esta instancia es utilizada para almacenar los valores de los *meters* por los subaspectos que afectan diferentes partes del sistema. La resolución del conflicto, se basa en cambiar la instancia de `MetersManager` que estos aspectos utilizan, al estar la *SM* en modo *demo*.

Como muestra la figura 5.6, el aspecto `Demo` afecta el método `getManager()` del aspecto `MetersAspect`. Si el modo *demo* se encuentra activo, se retorna una instancia de `MetersManager` que es mantenida por el aspecto `Demo`.

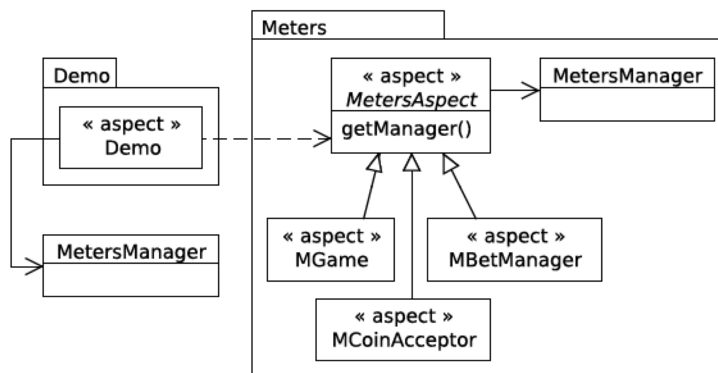

Figura 5.6: El aspecto `Demo` afecta el *join point* `MetersAspect.getManager()`.

El fragmento de código 5.4 muestra el *around advice* utilizado para tratar el conflicto con el *concern* de *Meters*. Cuando el *join point* `MetersAspect.manager()` es alcanzado en tiempo de ejecución, en la línea 4 del *advice*, se evalúa si el modo *demo* se encuentra activo. Si este es el caso, se retorna una instancia del objeto `MetersManager` que es local al aspecto `Demo`, como se puede ver en la línea 5.
Cuando la evaluación de la condición es falsa, se procede a ejecutar de manera normal el método (línea 6). Esto retornará la instancia de `MetersManager` del aspecto `MetersAspect`, que no sufrió alteraciones durante el modo *demo*.

Código 5.4: ***Around advice*** **del aspecto** `Demo` **utilizado para tratar el conflicto con el** ***concern*** **de** ***Meters*****.**

```
1 MetersManager around() :
2               execution( MetersManager MetersAspect.manager()
3 {
4   if (_demoIsOn) {
5       return _fakeMetersManager;
6   }else{
7       return proceed();
8   }
9 }
```

Esta forma de resolver el conflicto, implica que el aspecto `Demo` conozca que reemplazando la instancia del objeto `MetersManager` del aspecto `MetersAspect`, anula el *concern* de *Meters*.
La misma estrategia puede utilizarse para el conflicto con el *concern* de *Game Recall*, donde el aspecto `GameRecallAspect` tiene como colaborador una instancia de la clase `GameRecallManager`.

5.3.3. Tratamiento de la interacción entre los concerns: Demo y Program Resumption

La responsabilidad del *concern* de *Program Resumption* consiste en persistir ciertos datos de la *SM*, por ejemplo los valores de los *meters*. Las regulaciones señalan que los valores de configuración o estados que varíen en modo *demo*, deben ser revertidos al salir de este modo.

Como se vio en la sección 4.4, la implementación del *concern* de *Program Resumption* se compone de una jerarquía de aspectos. Los subaspectos utilizan métodos que el aspecto `ProgramResumptionAspect` provee, para obtener y guardar datos, utilizando los servicios provistos por la *HAL*.
Los métodos del aspecto `ProgramResumptionAspect` retornan un valor de tipo *boolean*, esto indica si la operación pudo ser realizada en la `HAL`. El mecanismo para resolver el conflicto en modo *demo*, consiste en evitar la ejecución de los métodos de la superclase, anulando la funcionalidad del *concern* de *Program Resumption*.

Como se puede ver en el fragmento de código 5.5, se utiliza un *pointcut* que captura la ejecución de todos los métodos que retornan un valor de tipo *boolean* del aspecto `ProgramResumptionAspect`. De la misma manera que para el *concern* de *Meters*, se procede con la ejecución normal si el modo *demo* no se encuentra activo. De no ser así, se retorna `false`, indicando que la operación no pudo ser realizada en el objeto `HAL`.

Código 5.5: ***Around advice*** **del aspecto** `Demo` **utilizado para tratar el conflicto con el** ***concern*** **de** ***Program Resumption.***

```
1 boolean around() :
2         execution( boolean ProgramResumptionAspect.*(..) );
3 {
4   if (_demoIsOn) {
5       return false;
6   } else {
7       return proceed();
8   }
9 }
```

En este caso, el acoplamiento entre el mecanismo y el *concern* de *Program Resumption* es alto. La solución se basa en conocer que los métodos del aspecto base son utilizados por la jerarquía y que anulando su ejecución se evita el conflicto.

5.3.4. Tratamiento de la interacción entre los concerns: Demo y Communication Protocols

Con el objetivo de mantener consistentes los datos en los sistemas de monitoreo, las regulaciones indican que una *SM* en modo *demo* debe reportase como fuera de servicio.
Por esto, no deben enviarse notificaciones de eventos, que de producirse estando la *SM* en modo normal, serían reportados. Un ejemplo de esto, es el evento de *begin game*. Además, una *SM* en modo *demo* que recibe comandos de consulta desde un servidor, debe responder con un mensaje que indique que se encuentra fuera de servicio.

Como se detalló en 4.7, los aspectos del *concern* de *Communication Protocol* monitorean el comportamiento de determinados objetos con el objetivo de reportar eventos a los servidores. Dichos eventos son enviados utilizando el método `sendEvent()` del objeto `CommProtocol`.
Por otro lado, cuando un objeto `CommProtocol` recibe un comando desde los sistemas de monitoreo, dicho comando es procesado si el protocolo se encuentra en línea. En caso contrario, se retorna un mensaje que notifica que el protocolo esta fuera de servicio.

Tanto al recibir un comando, como al enviar un mensaje, un objeto `CommProtocol` valida mediante el método `isOnline()` cual es su estado. Este *join point* es el que utiliza el mecanismo para resolver el conflicto con el *concern* de *Communication Protocol*. El listado de código 5.6, contiene la implementación del *around advice*. De la misma manera que con los casos anteriores, sólo se ejecuta el `proceed()` si el modo *demo* no se encuentra activo, en caso contrario se retorna

false.

Código 5.6: ***Around advice*** **del aspecto** `Demo` **utilizado para tratar el conflicto con el** ***concern*** **de** ***Communication Protocol*****.**

```
1  boolean around() :
2          execution( boolean CommProtocol.isOnline() );
3  {
4      if (_demoIsOn) {
5          return false;
6      } else {
7          return proceed();
8      }
9  }
```

De esta manera, si el modo *demo* se encuentra activo, los cambios de estados en la *SM* no seran reportados por los protocolos. De la misma forma, al recibir un comando desde los sistemas de monitoreo la *SM* se reportará como fuera de servicio.

5.3.5. Generalización del mecanismo

Analizando los casos planteados, se observa que todos los conflictos fueron tratados utilizando un *around advice* que retorna un valor que anula la funcionalidad del *concern* en conflicto. El cuadro 5.1 muestra para cada *concern* en conflicto, el *pointcut* sobre el que se instalo el *around advice* y el valor de retorno utilizado cuando el modo *demo* se encuentra activo.

Cuadro 5.1: *Pointcuts* y valores de retorno utilizados para resolver los conflictos.

***Concern* en conflicto**	***Pointcut* que anula el conflicto**	**Valor de retorno que resuelve el conflicto**
Meters	`MetersAspect.manager()`	*fake* `MetersManager`
Game Recall	`GameRecallAspect.manager()`	*fake* `GameRecallManager`
Program Resumption	`ProgramResumptionAspect.*`	`false`
Communication Protocol	`CommProtocol.isOnline()`	`false`

En los *advices* utilizados para resolver los conflictos se detectó un patrón que se muestra en el listado 5.7. Utilizando el mismo, la resolución de un conflicto se puede resumir en los siguientes pasos:

1. Definir el *pointcut* que captura el o los puntos del código en los cuales alterando su valor de retorno, se "desactiva" el *concern* en conflicto.

2. Detectar cual es el valor de retorno necesario en este *pointcut*, para anular el conflicto.

3. Instanciar un *around advice* sobre el *pointcut* identificado, que retorna el valor detectado en caso de estar en una situación de conflicto o permite la ejecución normal en caso contrario.

Código 5.7: Patrón detectado en los casos implementados.

```
1  RETURN_TYPE around() : POINTCUT()
2  {
3      if (CONFLICT_CONDITION) {
4          return CONFLICT_RESOLUTION_VALUE;
5      } else {
6          return proceed();
7      }
8  }
```

5.4. Dependency

Esta forma de interacción cubre las situaciones donde un aspecto A, para funcionar de manera correcta, depende de otro aspecto B. Si B no está presente, el comportamiento de A podría generar errores en el *software*. Dada la característica de *obliviousness* entre aspectos [8], es necesario desarrollar un mecanismo que garantice que no se produzcan errores entre aspectos dependientes.

En el *software* de las *SM* los protocolos deben responder consultas, como se vio en 3.1.6. Por ejemplo, si el protolo *PCP* recibe una consulta sobre el valor de un *meter* determinado, es necesario consultar al *concern* de *Meters* sobre el mismo. Si este *concern* no está funcionando, es decir los *meters* no están siendo actualizados, el valor que reporte el protocolo será incorrecto. Existe entonces una interacción de tipo *dependency*, entre los *concerns* de: *Meters* y *Communication Protocols*.

5.4.1. Tratamiento de la interacción

A nivel de implementación como se detalló en el capítulo 4, los aspectos del *concern* de *Meters* mantienen actualizados el objeto `MetersManager`. El mismo contiene los valores de los *meters* y tiene la capacidad de retornar el valor actual de un determinado *meter*. Como también fue detallado, los protocolos de comunicación al recibir una instancia de `ProtocolCmd`, proceden a ejecutar el método `execute()` del mismo.

Como se puede ver en la figura 5.7, es necesario que en la ejecución del método `GetMeter.execute()`, se pueda obtener del objeto `MetersManager` el valor del *meter* correspondiente. Esta interacción, genera una dependencia desde el comando de protocolo `GetMeter` hacia el objeto `MetersManager`, objetos de los *concerns* de *Communication Protocol* y *Meters* respectivamente.

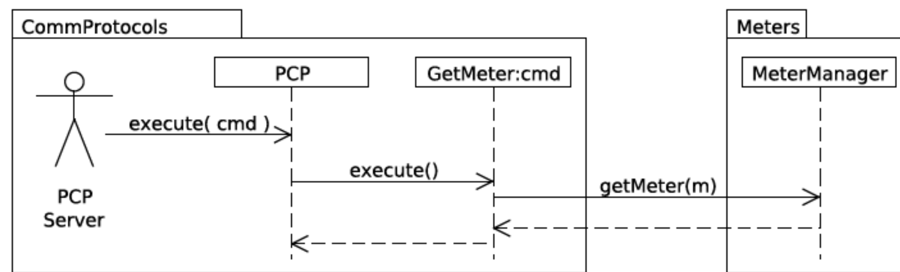

Figura 5.7: Dependencia entre los objetos `GetMeter` y `MetersManager`

Esta es una relación de conocimiento que cruza las fronteras de los *concerns*. A nivel de requerimientos y diseño, es deseable que la dependencia sea documentada de manera explicita. Sin embargo, a nivel de implementación no es necesario el desarrollo de un mecanismo para su tratamiento, ya que es una relación de colaboración que puede resolverse de diferentes maneras.

Para resolver la dependencia, una opción es que el objeto `GetMeter` tenga una referencia a la instancia de `MetersManager`. La misma podría ser recibida por parámetro en el constructor de la clase `GetMeter`. De esta manera, cuando el método `GetMeter.execute()` es procesado, se consulta directamente el valor del *meter* requerido a este objeto.

Otra alternativa, es utilizar un *before advice* sobre el método `execute()`. En el mismo se guarda el valor del *meter* requerido en una variable de instancia del objeto `GetMeter`. De esta manera al ejecutar el método, se dispone del valor actualizado.

5.4.2. La interacción de tipo Dependency no requiere una implementación ad-hoc

Para asegurar que la interacción de tipo *dependency* planteada, no genera problemas en el *software* de una *SM*, es necesario que se cumplan las siguientes condiciones:

1. Que el sistema cuente con la funcionalidad del *concern* de *Meters* para que el *concern* de protocolos pueda realizar la consulta. En este caso es necesario que el objeto `MetersManager` exista.

2. Que el *concern* de *Meters* este funcionando correctamente. Es decir, que los aspectos estén actualizando los valores del objeto `MetersManager`.

Para dar por válido el punto 1, nuestra implementación nos asegura que:

- Una *SM* no puede funcionar sin el módulo de *Meters*, como es indicado por las regulaciones.
- El objeto *MetersManager* siempre va a estar creado e inicializado en *runtime*.

De esta manera, se descartan mecanismos básicos de *runtime check*. Por ejemplo, validar que el objecto `MetersManager` no sea `nil` antes de solicitarle el valor de un *meter*.

Para el punto 2, como el objeto `MetersManager` es actualizado por diferentes aspectos, un posible mecanismo podría consistir en asegurar que estos aspectos esten funcionando y por ende el objeto *MetersManager* tenga los valores correctos.
Pero para que esto no ocurra, deberían cumplirse alguna de las siguientes condiciones:

- Que los aspectos que actualizan el objeto `MetersManager` no esten instalados. Esto puede pasar por estar utilizando un lenguaje donde el proceso de *weaving* sea dinámico, lo cual no se cumple con *AspectJ*. También puede ocurrir si se utilizara *load-time weaving*, lo cual tampoco se cumple dado que la implementación realizada cuenta con una única configuración.
- Que no se cumplan determinadas condiciones que los aspectos del *concern* de *Meters* validan antes de actualizar el objeto `MetersManager` y que el *concern* de *Communication Protocol* desconoce.

De cumplirse esta última condición, los valores de los *Meters* serían inválidos.
Sin embargo, esta necesidad de asegurar consistencia es responsabilidad del *concern* de *Meters* y no de un mecanismo para tratar interecciones de tipo *Dependency*.

Si bien es necesario a nivel de requerimientos y diseño, indicar la relación de dependencia; de lo anterior se concluye que en la implementación no es necesario un mecanismo para tratar la interacción. Si no se cumple alguna de las condiciones planteadas, sería correcto que el sistema falle.
Este puede ser el caso de una *NullPointerException* si el `MetersManager` no está inicializado. Un mecanismo de `try/cath` que proteja al sistema de la posibilidad de que el `MetersManager` sea `nil`, sería redundante. Si pudiera hacer *unweaving* de un aspecto, se podría chequear la presencia del mismo. Pero nuevamente, se está protegiendo al sistema de cosas que no pueden pasar dentro de este contexto.

5.5. Reinforcement

Este tipo de interacción se produce cuando la presencia de un aspecto influye positivamente sobre otro, permitiendo que este último pueda extender su funcionalidad.

Como fue visto en el capítulo anterior, el *software* de una *SM* cuenta con la capacidad de detectar condiciones de error y tratar las mismas de acuerdo a lo que indican las regulaciones. Las *errors conditions* se generan a partir de determinadas señales de *hardware*, por esto, el conjunto de *errors conditions* que puede ser detectado en una *SM*, puede variar de una a otra.

A la vez, las regulaciones indican que los protocolos deben notificar las *errors conditions* a los servidores de monitoreo, cuando la *SM* tenga la capacidad de detectarlas. Cada protocolo tiene un conjunto de *errors conditions* que le interesa reportar, pudiendo este conjunto variar entre diferentes protocolos. De esta manera, los protocolos pueden ampliar su funcionalidad, gracias a la información que provee el *concern* de *Errors Conditions*. Se produce entonces una interacción de tipo *reinforcement* entre los *concerns* de *Errors Conditions* y de *Communication Protocols*.

5.5.1. Tratamiento de la interacción

El objetivo del mecanismo desarrollado es que el programador tenga que tomar una decisión explícita sobre notificar o no una *error condition*. Es deseable que en situaciones en donde el *software* de la *SM* escala, el mecanismo ayude a que el programador no olvide reportar las *error conditions*. Este puede ser el caso en donde se implementa un nuevo protocolo o el sistema tiene la capacidad de detectar una nueva *error condition*.

Como se detalló en el capítulo anterior, el *concern* de *HAL* notifica a los *listeners* registrados ante una *signal*. Como se vio en la sección 4.7, el *listener* que registra el *concern* de *Error Condition* es quien crea un instancia de `ErrorCondition` y luego invoca al método `ErrorCondition.applyActions()`. Este es el *join point* de interés al que se asocia la ejecución del mecanismo que se describe a continuación.

Para tratar esta interacción positiva y ayudar a que la misma se produzca, se desarrolló un mecanismo *ad-hoc*. Dicho mecanismo agrega a una `ErrorCondition` la capacidad de notificarse en **un** protocolo. Cuando un protocolo captura la ejecución del método `ErrorCondition.applyActions()`, solicita a la instancia de esta *error condition* que se notifique.

Para esto, se alteró la estructura de la jerarquía `ErrorCondition` mediante el uso de *inter-type declarations*, desde los aspectos de la jerarquía `CommProtocolAspect`. La figura 5.8, indica la modificación que introduce el aspec-

to G2SAspect en la clase ErrorCondition. En dicho aspecto se define el método abstracto notifyG2S().

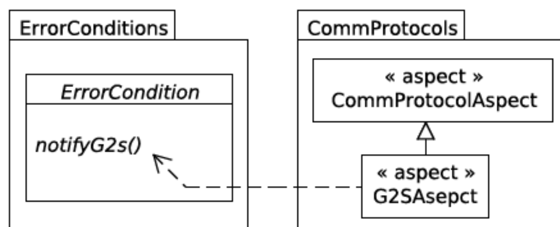

Figura 5.8: El método notifyG2S() es agregado a la clase ErrorCondition, desde el aspecto *G2SAspect*.

La codificación de esta acción para el protocolo *G2S*, se puede ver en el fragmento de código 5.8. De esta manera, cada subclase de la jerarquía ErrorCondition, **deberá** implementar el método abstracto notifyG2S().

Código 5.8: Método agregado a la clase ErrorCondition mediante el uso de *inter-type declarations*.

```
1 public aspect G2SAspect extends CommProtocolAspect {
2   ...
3   public abstract void ErrorCondition.notifyG2S();
4 }
```

Luego, el aspecto G2SAspect define un *before advice*. El mismo se asocia al *pointcut* que captura ErrorCondition.applyActions(). Como muestra el fragmento de código 5.9 en la línea 7, en dicho *advice* se invoca el método notifyG2S() sobre la *error condition* detectada.

Código 5.9: *Before advice* que detecta una *error condition* y solicita la notificación de la misma.

```
1 public aspect G2SAspect extends CommProtocolAspect {
2   ...
3   before( ErrorCondition errorCondition ) :
4           target(errorCondition) &&
5           execution(void ErrorCondition.applyActions());
6   {
7       errorCondition.notifyG2S();
8   }
9 }
```

En cada subclase de la jerarquía de ErrorCondition, el método notifyG2S() contiene la implementación concreta para notificar la *error condition* en el protocolo *G2S*. Como se puede ver en la figura 5.9, para la subclase DoorOpen, este comporta-

miento también es agregado mediante el uso de *inter-type declarations* en el aspecto `G2SAspect`.

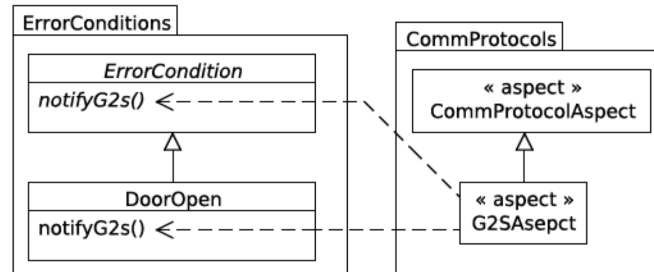

Figura 5.9: El método `notifyG2S()` es agregado a la clase `DoorOpen` por el aspecto *G2SAspect*.

Por cada ocurrencia de una *Error Condition* que un protocolo esté interesado en notificar, debe enviar el evento correspondiente. El fragmento de código 5.10, muestra la implementación del método `notifyG2S()` para el caso de la *error condition* `DoorOpen`. En la línea 4, se crea un evento que representa dicha *error condition* y es enviado por el protocolo *G2S*.

Código 5.10: Implementación del método `notifyG2S()` en la *error condition* `DoorOpen`.

```
1 public aspect G2SAspect extends CommProtocolAspect {
2   ...
3   public void DoorOpen.notifyG2S() {
4       protocol().sendEvent( new DoorOpenEvent() );
5   }
6 }
```

Cuando una nueva `ErrorCondition` es agregada al sistema, se genera un error de compilación si el método `notifyG2S()` no está implementado. De esta manera, el desarrollador deberá redefinir el método, viéndose obligado a decidir explicitamente si dicha `ErrorCondition` debe o no ser reportada.

En el caso de que un nuevo protocolo sea agregado al sistema, o si es necesario que un protocolo existente cuente con la capacidad de notificar *errors conditions*, se deberán realizar las siguientes acciones:

- Introducir mediante *inter-type declarations* el método `notifyProtocolName` en la superclase `ErrorCondition`.
- Definir el *before advice* que solicita a una instancia de `ErrorCondition` que se notifique al capturar el *joint point* `ErrorCondition.applyActions()`.

- Definir el método `notifyProtocolName` en cada *error condition* de la jerarquía. En el cuerpo de este método, se debe completar la funcionalidad requerida para reportar la *error condition* o, dejar el método vacío. En este caso se indica de manera explícita, que esa *error condition* particular no necesita ser reportada.

La figura 5.10, muestra el resultado de agregar la funcionalidad de reportar las *erros conditions* en el protocolo *PCP*. Por cada nuevo protocolo, la superclase `ErrorCondition`, contendrá el método abstracto que las subclases deben definir.

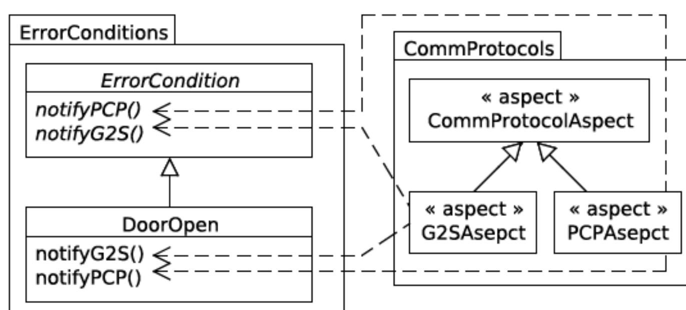

Figura 5.10: Resultado de agregar la capacidad de notificar *error conditions* en el protocolo *PCP*.

De esta manera, queda completa la descripción del mecanismo para resolver la interacción de tipo *reinforcement*. Con la solución descrita, se logró que una decisión explícita deba ser tomada cuando una *Error Condition* es agregada al sistema, favoreciendo a que esta interacción positiva se produzca.

El código que compone la solución se divide entre las jerarquías de `CommProtocol` y `ErrorCondition`. Sin embargo, el comportamiento agregado en la jerarquía de `ErrorCondition` se hizo mediante el uso de *inter-type declarations*, quedando la totalidad del código en el módulo de *Communication Protocols*.

5.6. Análisis de los mecanismos desarrollados

A continuación se presenta un análisis de los mecanimos desarrollados. Se discuten ventajas y desventajas en lo referido a generalización, modularización, escalabilidad y otros factores, para cada una de las soluciones que fueron presentadas en este capítulo.

5.6.1. Modularización

En el mecanismo presentado para tratar la interacción de tipo **Mutex** se obtuvo un desarrolló modular, en el cual los aspectos del dominio son independientes de la implementación de *mutex*.
Cada *mutex* queda encapsulado en un aspecto, por lo tanto, para incluir un *mutex* alcanza con compilar el aspecto que lo contiene. Por el contrario, para eliminar el *mutex*, se debe excluir del proceso de compilación el aspecto que lo implementa. Notar que no hay necesidad de modificar algún objeto u aspecto del sistema y que en *AspectJ* es posible configurar que aspectos forman parte de una determinada compilación.

En lo que respecta a la solución para tratar una interacción de tipo **Conflict**, el mecanismo desarrollado fue incorporado dentro del aspecto que contiene la funcionalidad principal del *concern Demo*.
Desacoplar completamente el mecanismo generaría una interacción de tipo *Dependency* entre el aspecto *Demo* y el aspecto que resuelve el conflicto. Esto es debido a que el modo *demo*, requiere para activarse tanto de su funcionalidad principal como de la resolución de los conflictos que se producen con el resto de los *concerns*.

Como fue detallado, para resolver las interacciones de tipo **Reinforcement**, se definió un mecanismo que requiere ser implementado completamente para cada protocolo. Dado el uso de *inter-type declarations*, el código que lo implementa se encuentra en el *concern* de *Communication Protocols*, más específicamente en el aspecto que implementa la funcionalidad del protocolo. Esta implementación podría modularizarse aún más, separando el código del mecanismo en otro aspecto.

5.6.2. Generalización

De los mecanismos desarrollados, se logró generalizar la solución para las interacciones de tipo **Mutex**.
La implementación del mecanismo *Mutex*, puede ser reutilizada para otros aspectos, ya que la misma es independiente del dominio subyacente. Algunas de las estrategias implementadas para la resolución del *mutex*, también son independientes y pueden ser reutilizadas sin modificaciones.

Para la interacciones de tipo **Conflict** no se obtuvo una generalización del mecanismo, pero sí se detectó un patrón de resolución en los casos implementados. El mismo consiste en tratar cada conflicto con un *around advice*, el cual permite continuar la ejecución normal de no mediar conflicto.
No fue posible implementar un solución genérica para tratar esta interacción. Como se mostró en la tabla 5.1, cada caso necesita de un tratamiento particular y además, no es posible referenciar los *advices* en *AspectJ* dado que son anónimos.

5.6.3. Escalabilidad

Es esperable que se generen nuevos casos donde sea necesario el mecanismo **Mutex**. Esto puede deberse por ejemplo, a nuevos comandos que modifiquen el estado de la *SM* y los mismos puedan ser recibidos por más de un protocolo de comunicación. En este sentido la solución presentada es escalable. Cuando un nuevo comando requiera ser tratado con un *Mutex*, sólo se debe definir un aspecto que define de manera concreta el *abstract pointcut* y especificar cual es la política a utilizar para determinar la obtención del *lock*.

La solución para las interacciones de tipo **Conflict**, puede ser utilizada en nuevas situaciones donde logre identificarse un *join point*, que de evitar su ejecución prevenga la ocurrencia de un conflicto. Utilizando el patrón detectado, se debe definir un *around advice* sobre el *pointcut* en donde el conflicto se produce.

En el caso de las interacciones de tipo **Reinforcement** es necesario contemplar que el sistema puede evolucionar de dos maneras: ante la aparición de una nueva *Error Condition* y el soporte para nuevos protocolos. Por lo tanto vemos que el mecanimo presentado permite extender el sistema correctamente.
En primer lugar, cuando una nueva *Error Condition* es agregada al sistema, se deberán agregar a la misma los método `notifyProtocol()` para cada uno de los *Communication Protocols*.
Por otro lado, se podría dar el caso de que se añada un nuevo protocolo, en esta situación se deberá implementar en la jerarquía de `ErrorCondition` el método `notifyNewProtocol()`.

5.6.4. Mantenibilidad

Una desventaja o carencia del mecanismo **Mutex**, con respecto a la mantenibilidad, es la imposibilidad de notificar la falta del mismo cuando es necesario. Está claro que para nuevos casos, donde sea necesario un *Mutex*, es en la etapa de diseño donde debe indicarse.
Sin embargo, se pueden generar inconsistencias en la *SM* para los casos donde un determinado *pointcut* pase a ser inválido, debido a que se modifica la estructura de la clase que lo contiene. En el caso de *AspectJ*, el desarrollador solo recibirá un *warning* de que no se pudo realizar el *weaving* de manera correcta.
Este es un problema conocido como *fragil pointcut problem* [14, 19] donde dada la limitada expresividad para definir los *pointcuts* que brindan los lenguajes, dichos *pointcuts* suelen estar acoplados a la estructura del código base. Como consecuencia de ello, cambios seguros en el código de base, pueden tener un impacto inesperado e indeseable sobre el comportamiento de los aspectos en el sistema.

El *concern Demo* produce una interacción de tipo **Conflict** con varios *con-*

cerns del dominio, la implementación de esta interacción complica la mantenibilidad. Como fue detallado, en este caso se utiliza un *around advice* que retorna un valor que anula el conflicto. Es por ello, que no sólo se debe prestar atención a casos de *pointcuts* inválidos, sino también a los cambios en la interpretacion que se hace de los valores de retorno, ya que pueden invalidar la resolución del conflicto.

Por otro lado, tareas de *debugging* se verán afectadas por el mecanismo **Conflict**, debido a que los *around advices* se encuentran siempre instalados, aún cuando el modo *demo* no está activo.
Esto podría resolverse de dos maneras: una opción es tener dos binarios, donde uno de ellos tiene el mecanismo desarrollado. Luego, dependiendo del estado del *switch* de *demo*, se decide cual utilizar. Esto duplica el costo de certificar un juego, dado que se cuenta con 2 aplicaciones.
Otra opción, es utilizar una característica de *AspectJ* que permite realizar el *weaving* en tiempo de carga. De esta manera se podría configurar si disponer o no del modo *demo*. Ambas opciones descriptas requieren reiniciar el *software* de la *SM*.

Aunque no hubo necesidad de desarrollar un mecanismo *ad-hoc* para la interacción de tipo **Dependency**, la misma es difícil de mantener, dado que no hay algo explícito en el código fuente que ayude a preservarla.

5.6.5. Weaving dinámico

Para finalizar, dada la experiencia de haber trabajado con *AspectS* [2], se discuten a continuación características de los mecanismos que se verían afectadas si se utilizara una lenguaje de *weaving* dinámico.

Para la intereracción de tipo **Dependency** no hubo necesidad de desarrollar un mecanismo. Como fue descripto, esto se debe en principio al uso de un lenguaje estático. Si este no fuera el caso, sería necesario un tipo de chequeo que permita asegurar que el *concern* del que se depende se encuentre activo.
Para el caso específico de *Meters*, se debería controlar que el objeto `MetersManager` este siendo actualizado y de no ser asi, no reportar los valores en los protocolos de comunicación.

En el caso de la solución propuesta para el tipo **Conflict**, de usar *weaving* dinámico, los aspectos podrían ser instalados y removidos en tiempo de ejecución según el estado del *switch* de *demo*.

Por último, el mecanismo presentado para la interacción de tipo **Reinforcement**, obliga a decidir explicitamente si una *Error Condition* debe o no ser reportada antes de la compilación. De utilizar un lenguaje de *weaving* dinámico y no tomar la decisión de notificar una *Error Condition*, un error en tiempo de ejecución se produciría cuando se intenta notificar una *Error Condition* por primera vez.

Capítulo 6

Conclusiones y Trabajos Futuros

En este trabajo se han estudiado interacciones entre aspectos en un dominio de la industria, como son las *Slot Machines*.
Se identificaron varios *crosscutting concerns* funcionales mediante el estudio de los *concerns* más representativos en el dominio, tomando como punto de referencia los ya estudiados en la etapa de análisis de requerimientos [30].

Sobre los *crosscutting concerns* funcionales y no funcionales del dominio, se identificaron varias interacciones. Las mismas se categorizaron según la clasificación de Sanen *et al.* [23] en *Mutex, Conflict, Dependency* y *Reinforcement*.
Se seleccionó una interacción para cada una de estas categorías, para las cuales se estudió un mecanismo que permite su tratamiento, realizando una implementación concreta del mismo sobre el *software* desarrollado.

Las cuatro interacciones fueron implementadas de forma tal que la *SM* se comporte de la manera deseada. De cada uno de los mecanismos se puede destacar:

- *Mutex*: se desarrolló un mecanismo modular que pudo ser generalizado y cuenta con la ventaja de ser independiente del dominio subyacente. Una desventaja del mecanismo es que puede verse afectado por el *fragil pointcut problem* [14, 19].
- *Reinforcement*: se implementó un mecanismo *ad-hoc* mediante el uso de *inter-type declarations*. El mismo permite que el sistema escale de forma correcta cuando una nueva *error condition* necesita ser reportada o cuando un *SM* utiliza un nuevo protocolo.
- *Conflict*: en este caso también se implementó un mecanismo *ad-hoc*, sobre el cual se detectó un patrón para tratar las interacciones. El mismo es aplicable en situaciones donde logre identificarse un *join point*, que de evitar su ejecución prevenga de una interacción de este tipo.

- *Dependency*: se concluyó que no hace falta un mecanismo para tratar esta interacción en el contexto de trabajo a nivel de implementación.

Para cada uno de los mecanismos desarrollados, se analizaron ventajas y desventajas con respecto a la mantenibilidad, genericidad, escalabilidad y modularización.

En un sistema desarrollado usando orientación a aspectos los aspectos interactúan necesariamente de varias maneras. La implementación de las interacciones puede realizarse de manera manual y ajustada a cada caso, pero esta práctica es propensa a errores y genera problemas de mantenibilidad. Por lo tanto, es necesario contar con soporte para las interacciones, en lo posible a nivel de lenguaje.

Se propone como trabajo a futuro, aplicar los mecanismos propuestos en otros dominios, analizando si los mismos permiten implementar otras interacciones pertenecientes a las categorías presentadas en este trabajo. De esta manera se puede comprobar si las generalizaciones propuestas son reusables. Por otra parte, es necesario profundizar el estudio de los mecanismos planteados e implementar formas alternativas para tratar las interacciones, con el objetivo de eliminar algunas de las limitaciones de los mecanismos propuestos.

En este trabajo se ha logrado estudiar e implementar las interacciones entre aspectos, analizando los desafíos que ellas presentan, lo cual constituye el núcleo de la propuesta de esta tesina. El desarrollo de la misma ha contribuido a afianzar mis conocimientos y adquirir otros nuevos. Además, representa una experiencia muy valiosa, contribuyendo a mi formación profesional y personal.

Apéndice A

Modelo de Objetos

En el capítulo 4 se decriben detalles de implementación de cada *concern* teniendo en cuenta como los objetos interactúan con los aspectos del sistema. A continuación se detallan las reponsabilidades de los objetos involucrados en la solución.

A.1. Paquete Game

Las clases del paquete *Game* se pueden ver en la figura A.1.

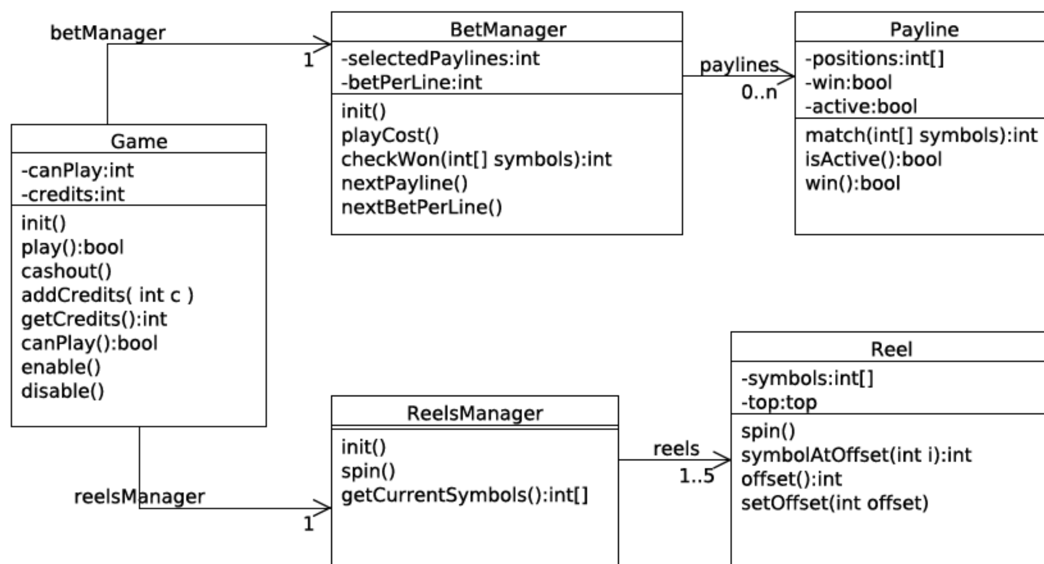

Figura A.1: Clases del paquete *Game*.

- `Game`
    - implementa la interfaz de alto nivel de un juego de apuestas.
    - el juego puede estar habilitado o deshabilitado.
    - un juego necesita de créditos para realizar una jugada, además de estar habilitado.
    - en cada jugada se descuentan los créditos que se apuestan y se suman los ganados, si es que los hubo.
    - se puede jugar mientras queden créditos o hasta que el jugador retira los mismos.

- `BetManager`
    - Maneja el sistema de apuestas del juego: configuración de las líneas de pago, cantidad de líneas apostadas, cantidad de créditos apostados por línea.
    - Conoce el costo de cada jugada (*betPerLine* x *selectedPaylines*).
    - Define una colección de objetos `Payline`, los cuales se utilizan para saber si hay líneas ganadoras entre las seleccionadas.

- `Payline`:
    - Representa una línea de apuesta en un juego de *reels*.
    - Una *payline* puede estar activa, es decir apostada o no.
    - Permite determinar que cantidad de símbolos iguales consecutivos hay en sus posiciones y con esto saber si es o no una línea ganadora.

- `ReelsManager`:
    - Crea e inicializa 5 objetos de la clase `Reel`.
    - La operación de `spin()` delega en cada *reel* el sorteo de la nueva posición.
    - Permite consultar la ventana de símbolos resultante del sorteo.

- `Reel`:
    - Modela una tira circular de símbolos.
    - El método `spin()` genera una nuevo *offset* de manera aleatoria.

A.2. Paquete HAL

Las clases del paquete *HAL* se pueden ver en la figura A.2.

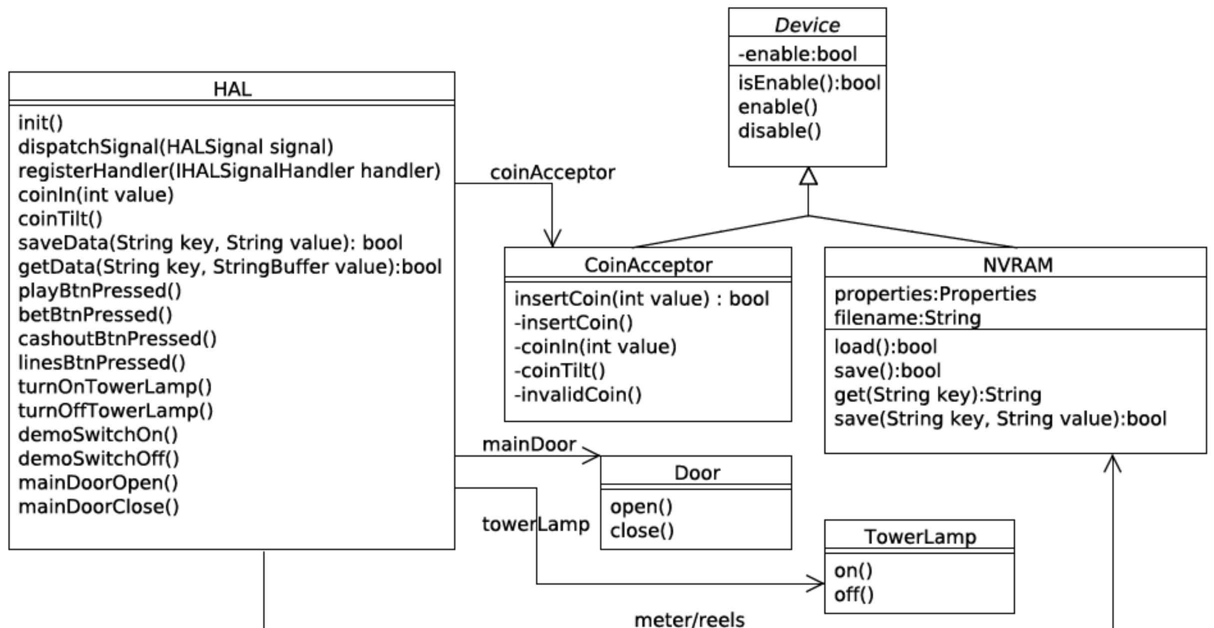

Figura A.2: Clases del paquete *HAL*.

- `Device`
    - Representa un disposivo de *hardware*.
    - Un disposivo puede estar o no activo.

- `CoinAcceptor`
    - Esta clase representa un disposivo que permite el ingreso de monedas al juego a través de la *HAL*.
    - Una moneda ingresada puede:
        - ser válida y tener un valor reconocido.
        - ser inválida (*invalidCoin*).
        - trabar y deshabilitar el dispositivo (*coinTilt*).

- `Door`
    - Esta clase representa un puerta del gabinete, como puede ser la *main door*.

- Una EGM tiene un puerta principal que de ser abierta, debe deshabilitar el juego por completo.

- `NVRAM`
  - Esta clase representa un dispositivo de memoria de acceso aleatorio no volátil.
  - Permite guardar datos como un par (clave, valor).
  - Permite recuperar datos a partir de una clave.

- `TowerLamp`
  - Representa la lámpara que tiene una EGM que puede encenderse o apagarse de acuerdo a los distintos estados de la EGM.

- `HAL`
  - *Hardware Abstraction Layer*
  - Esta clase abstrae de la comunicación con los dispositivos físicos al resto de los componentes del *software*.

A.3. Paquete Meters

Las clases del paquete *Meters* se pueden ver en la figura A.3.

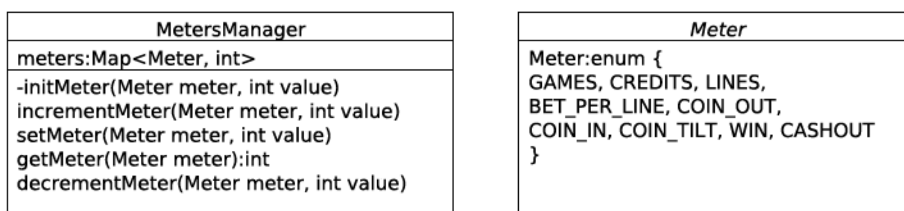

Figura A.3: Clases del paquete *Meters*.

- `Meter`
  - Esta clase se utiliza para definir un tipo enumerativo. El mismo es usado para el manejo de *meters*.
  - Algunos valores son: GAMES, CREDITS, LINES, WIN, CASHOUT.

- `MetersManager`
  - Mantiena un mapa de *meters*.
  - Para el acceso a los mismos se utiliza el tipo enumerativo anteriormente descripto.
  - Para un *meter* en particular esta clase permite:
    - asignar un valor específico.
    - consultar el valor actual.
    - incrementar el valor actual.
    - decrementar el valor actual.

A.4. Paquete Game Recall

Las clases del paquete *Game Recall* se pueden ver en la figura A.4.

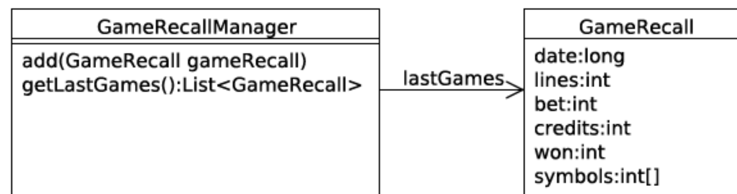

Figura A.4: Clases del paquete *Game Recall*.

- `GameRecall`
  - Un objeto `GameRecall` mantiene información de una jugada en particular.
  - Esta información se compone de:
    - fecha y hora en que se realizó la jugada.
    - datos de la apuesta realizada (*bet*, *lines*).
    - cantidad de créditos que había en el juego antes de realizar la jugada.
    - cantidad de créditos ganados.
    - símbolos de los *reels* que resultaron del sorteo.

- `GameRecallManager`
  - Mantiene una pila con las últimos *game recalls*.
  - Permite agregar un nuevo objeto `GameRecall`.
  - Permite consultar el contenido de la pila.

A.5. Paquete Errors

Las clases del paquete *Errors* se pueden ver en la figura A.5.

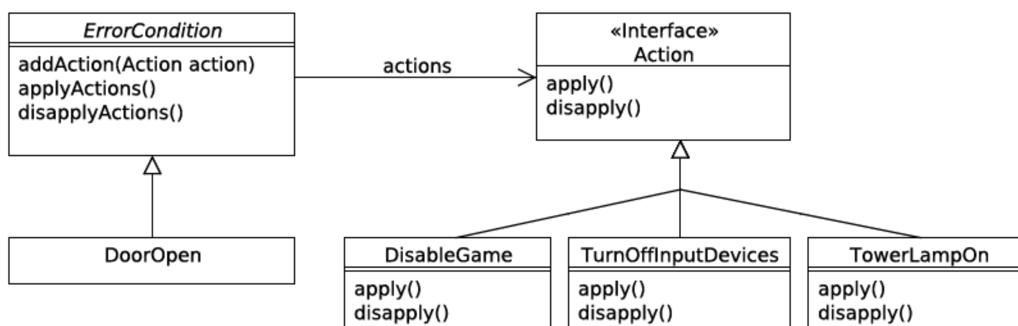

Figura A.5: Clases del paquete *Erros*.

- `Action`
  - Representa una acción a ser realizada ante una *error condition*.
  - Una acción puede aplicarse o desaplicarse.
  - Las clases `DisableGame`, `TurnOffInputDevices` y `TowerLampOn` implementan la interfaz `Action`.
- `ErrorCondition`
  - Es una clase abstracta que contiene las acciones que se toman al generarse una *error condition* particular.
  - Al generarse una *error condition* se aplican las acciones.
  - Cuando una *error condition* deja de estar presente, se desaplican las acciones.
- `DoorOpen`
  - Es un tipo de `ErrorCondition` que se genera al abrirse un puerta y se cancela al cerrarse la misma.

A.6. Paquete Protocols

Las clases del paquete *Protocols* se pueden ver en la figura A.6.

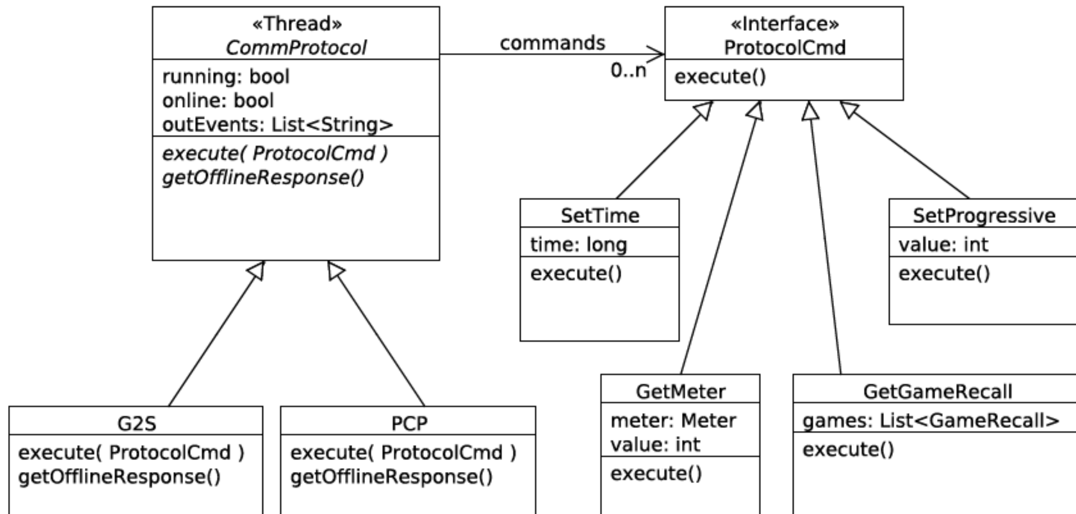

Figura A.6: Clases del paquete *Protocols*.

- `ProtocolCmd`
  - Es una interfaz que deben implementar los comandos que se envían y reciben a través de un protocolo de comunicación.
  - Las siguientes clases implementan dicha interfaz:
    - `SetTime` este comando al ejecutarse configura la hora de la EGM.
    - `GetMeter` al recibir este comando debe retornarse el valor del *meter* requerido.
    - `GetGameRecall` al recibir este comando se retorna la lista de *game recalls*.

- `CommProtocol`
  - Una instancia de esta clase, es un *thread* que esta a la espera de recibir un objeto `ProtocolCmd`.
  - También tiene la responsabilidad de enviar a los protocolos los eventos que ocurren en la *SM*.
  - Al recibirse un *ProtocolCmd* se ejecuta la acción asociada al comando.

Apéndice B

Software desarrollado

El *software* de la *SM* desarrollado se compone de dos proyectos denominados: *egm* y *egmui*. El proyecto *egm* contiene la implementación descripta en los capítulos 4 y 5, mietras que *egmui* implementa una interfaz gráfica que facilita la interacción del usuario con el proyecto *egm*. A continuación se describen detalles de la interacción del usuario con dicho *software*.

B.1. Proyecto Egm

El proyecto *egm* permite la interacción del usuario con el *software* de la *SM* mediante la línea de comandos, sin necesidad de utilizar la *UI*. El usuario debe introducir ciertas letras o números que se corresponden con determinadas acciones. El fragmento de código B.1 muestra parte de la salida del comando h(help) que lista los comandos que son aceptados y que permiten interactuar con la *SM*.

Código B.1: Lista de comando aceptados por la *SM* en modo consola.

```
Game/>: h
[GAME]
 p       play
 l       lines button
 b       bet button
 i       insert coin
 [HAL]
 a       toogle main door open/close
 d       toogle demo switch on/off
 [PROTOCOLS]
 1       G2S set time
 2       PCP set time
 3       G2S get meter
 4       G2S get Meter
```

B.2. Proyecto EgmUI

El proyecto *egmui* utiliza la librería *Qt Jambi*[21]. La misma permite utilizar el *framework Qt* para el desarrollo de interfaces gráficas desde *Java*. A continuación se describen los componentes de la interfaz.

La interfaz desarrollada se divide en tres componentes. La figura B.1 muestra el componente principal de la vista, donde se indican los siguientes elementos:

1. Indicador de la cantidad de créditos que se están apostando (*bet per line* x *selected paylines*).
2. Indicador de la cantidad de créditos ganados en la última jugada.
3. Botón para cambiar la cantidad de líneas apostadas.
4. Botón para cambiar la cantidad de créditos apostados por línea.
5. Botón para realizar la acción de *play*.
6. Botón para realizar la acción de *cashout*.

Figura B.1: Vista principal de la interfaz gráfica.

La interfaz cuenta con un segundo componente que permite el monitoreo de los *meters*, *game recalls* y hora de la *SM*. El mismo se puede ver en la figura B.2, donde se indica

:

1. Es el listado de cada uno de los *meters* con su valor asociado.

2. Muestra la lista de los últimos 10 *game recalls*. Para cada uno se indica *timestamp*, datos de la apuesta y créditos ganados.

3. Indica la hora de la *SM*.

```
WIN=12
COIN_IN=15
LINES=5
COIN_TILT=0
BET_PER_LINE=2
GAMES=66
COIN_OUT=0
CREDITS=125

Time | Lines | Bet | Credits | Won
15:58:12 | 5 | 2 | 171 | 0
15:58:12 | 5 | 2 | 161 | 0
15:58:14 | 5 | 3 | 151 | 0
15:58:20 | 5 | 3 | 136 | 24
15:58:25 | 5 | 4 | 145 | 0
3/1/13 4:02:45 PM
```

Figura B.2: Vista de listado de *meters*, *game recalls* y hora de la *SM*.

La figura B.3 muestra el panel que permite simular el comportamiento de los protocolos de comunicación y algunas funciones de la *HAL*. Para los protocolos *G2S* y *PCP* es posible enviar los comandos:

- *Set time.*
- *Get meter (GAME).*
- *Get game recall.*

Los eventos de la *HAL* que pueden ser simulados desde dicho panel son:

- Dispositivo: *coin acceptor*, evento *insert coin.*
- Dispositivo: *Main door*, eventos: *open, close.*
- Dispositivo: *DIP switch*, eventos: *Demo on, Demo off.*

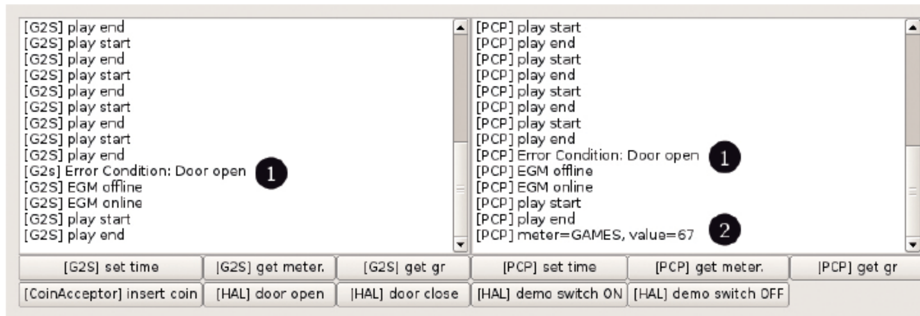

Figura B.3: Panel de control de *HAL*, protocolos *G2S* y *PCP*.

Además este panel cuenta con el *log* de los eventos recibidos por cada protocolo. En **1** de la figura B.3, se puede ver la recepción de la notificación de la *error condition Door open* y en **2** se observa la respuesta al comando *Get Meter* en el protocolo *PCP*.

Bibliografía

[1] Spring framework. http://www.springframework.org.

[2] Alejandro Alvarez, Arturo Zambrano, and Gustavo Rossi. Extending the smalltalk environment with aop. In *36th JAIIO, Students Work Contest*, August 2007.

[3] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[4] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 141–150, New York, NY, USA, 2004. ACM.

[5] Eclipse homepage.
http://www.eclipse.org.

[6] Eclipse ajdt plug-in.
http://www.eclipse.org/ajdt/.

[7] Johan Fabry and Daniel Galdames. Phantom: a modern aspect language for pharo smalltalk. *Software: Practice and Experience*, pages n/a–n/a, 2012.

[8] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[9] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, first edition, 2008.

[10] Gaming Laboratories International. *Gaming Devices in Casinos*. Available at: http://www.gaminglabs.com/.

[11] Gaming Standard Association. *Game to Server (G2S) Protocol Specification*. Available at: http://www.gamingstandards.com/.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Prentice Hall, June 2005.

[14] Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[15] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[19] Christian Koppen and Maximilian Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[20] Mattia Monga, Fatima Beltagui, and Lynne Blair. Investigating feature interactions by exploiting aspect oriented programming. Technical report, Dip . Elettronica e Informazione; Politecnico di Milano, 2003.

[21] Qt jambi homepage.
http://qt-jambi.org/.

[22] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, and W. Joosen. Aspect-oriented software development in practice: Tales from aosd-europe. *Computer*, 43(2):19 –26, feb. 2010.

[23] Frans Sanen, Eddy Truyen, Bart De Win, Wouter Joosen, Neil Loughran, Geoff Coulson, Awais Rashid, Andronikos Nedos, Andrew Jackson, and Siobhan Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[24] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in aop with aspectc++. In *Proceedings of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fourth SoMeTW05*, pages 33–53, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press.

[25] The AspectJ programming guide.
http://eclipse.org/aspectj/doc/released/progguide.

[26] Claire Tristram. The Technology Review 10: Emerging Technologies that Will Change the World. 1:97–103+, 2001.

[27] Xerox PARC. AspectJ home page.
http://eclipse.org/aspectj/.

[28] Arturo Zambrano, Alejandro Alvarez, Johan Fabry, and Silvia Gordillo. Aspect coordination for web applications in java and ruby. In *XXVIII International Conference of the Chilean Computer Society*, November 2009.

[29] Arturo Zambrano, Johan Fabry, and Silvia Gordillo. Expressing aspectual interactions in requirements engineering: Experiences, problems and solutions. *Science of Computer Programming*, 78(1):65 – 92, 2012. Special Section: Formal Aspects of Component Software.

[30] Arturo Zambrano, Johan Fabry, Guillermo Jacobson, and Silvia Gordillo. Expressing aspectual interactions in requirements engineering: experiences in the slot machine domain. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 2161–2168. ACM Press, 2010.