Exadat: Análisis de Variabilidad en Persistencia de Productos de Software

Nicolas Bortolotti[1,2], Silvio Gonnet[1], Horacio Leone[1]

[1] INGAR-CIDISI, Universidad Tecnológica Nacional, CONICET
Avellaneda 3657, Santa Fe, Argentina
[2] Facultad Regional San Francisco, UTN
{nbortolotti, sgonnet, hleone}@santafe-conicet.gov.ar

Abstract. Existe una tendencia mundial hacia el desarrollo y evolución de familias de productos en lugar de la creación de un producto de software para un cliente específico. Sin embargo, es común la construcción de tal familia a partir de varios sistemas a medida. Además, estas implementaciones usualmente carecen de la documentación de la arquitectura implementada. A partir de esta problemática se plantea en este trabajo la identificación de familia de productos de software desde una perspectiva de persistencia. El enfoque propuesto utiliza mecanismos de ingeniería reversa para reconstruir la arquitectura del producto implementado. A partir de la arquitectura se identifican posibles puntos de variación y variantes implicadas en la persistencia de datos.

Keywords: familia de productos de software, persistencia, variabilidad, arquitectura de software.

1 Introducción

Las organizaciones productoras de software tienden hacia el desarrollo y la evolución de familias de productos (visión de producto) en vez de la creación de un producto de software para un cliente específico [1, 2]. Una familia de producto está constituida por un núcleo central común pero posee variaciones específicas que permiten su adaptación a las necesidades específicas de clientes, diversos ambientes de interacción, y a distintos tipos de dispositivos computacionales. Sin embargo, la dinámica comercial de empresas desarrolladoras de software latinoamericanas y el grado de experiencia y conocimientos en el dominio de aplicación necesario para construir una familia de productos hace que las empresas locales que se lanzan a la construcción de una familia de productos inicien el desarrollo a partir de varios sistemas a medida construidos de manera "ad-hoc" para clientes específicos (visión de sistema). Estos sistemas evolucionan con el objeto de responder a los nuevos requerimientos impuestos por el cliente, generando un conjunto de productos de software con funcionalidades específicas a diversos entornos de interacción, como pueden ser ambientes móviles, web, "wap", o de gestión tradicional (por ejemplo

entorno win32). Generalmente, la documentación arquitectónica de estos artefactos, si es que existe, no refleja la arquitectura implementada, el activo principal de implementación que posee la empresa es un conjunto de ensamblados y componentes que poseen diversas dependencias entre sí.

Una empresa que emplee una visión de sistema y quiera evolucionar a una visión de producto debe transitar por un proceso complejo. Este proceso requiere de herramientas y técnicas para el procesamiento de artefactos del proceso de desarrollo que permitan extraer, abstraer, y "refactorizar" los mismos y de esta manera ayudar en la construcción de una familia de productos. En particular, deben brindar soporte en la reconstrucción arquitectónica [3], y a partir de la arquitectura, asistir al diseñador en el proceso de identificación de puntos de variación y sus respectivas variantes. Los puntos de variación y sus variantes derivan del concepto de variabilidad que surgió como resultado del trabajo con familias de productos, el mismo se vincula con los componentes y/o partes del producto de software, que por su reemplazo o adecuación, permiten dar origen a los diferentes productos integrantes de la familia. A partir del trabajo de Jacobson, Griss, y Jonsson [4] numerosas contribuciones abordan los mecanismos para poder analizar variabilidad, es decir, cómo los componentes de software pueden ser extendidos, cambiados, personalizados y configurados [5]. Algunas propuestas analizan el código fuente para detectar variabilidad mediante la duplicación de código y el empleo de polimorfismo [6], otras manipulan variables, parámetros, enlaces, configuración y comportamiento polimórfico [7].

En esta contribución se propone un método para el análisis de los distintos ensamblados que componen distintas soluciones de software individuales y que pueden ser propuestas como familia de productos de software. El método brinda soporte en la identificación de aspectos comunes entre un conjunto de productos de software, centrando el análisis en los distintos ambientes de persistencia e intentando identificar mecanismos clásicos de variabilidad [4, 7]. En una contribución previa [8] se empleó un enfoque "top-down" para analizar los distintos ambientes de interacción (las capas de presentación de los productos) y brindar soporte a la detección de puntos de variación que permitan una transición arquitectónica suave de crecimiento a una posible familia de productos de software. Para enfocarnos en la identificación de los puntos de variación y sus variantes, es fundamental definir las características que posee una familia de productos de software en un ambiente heterogéneo de soluciones de software, en donde se aplican técnicas de ingeniería reversa y reconstrucción arquitectónica. Estas características son presentadas en la Sección 2 y el método propuesto es definido en la Sección 3. La Sección 4 ilustra el método propuesto mediante la realización de un caso de estudio. Finalmente, las conclusiones y posibles líneas de acciones futuras son presentadas en la Sección 5.

2 Familia de Productos de Software

Las organizaciones productoras de software con una visión de sistemas poseen un portfolio compuesto por una gran cantidad de productos de software desarrollados a medida. Es decir, su activo principal está formado por un conjunto de ensamblados y

componentes que poseen diversas dependencias entre sí. En la Fig. 1 se representa este escenario, donde una familia de productos de software (*ExFamilia*) desarrollados de manera individual está conformada por diversos ensamblados (*Assembly*). Estos componentes de código pueden usualmente clasificarse empleando una arquitectura de dos o tres capas en: interfaz de usuario (*ExEnsambladoUI* en Fig. 1), lógica de negocios (*ExEnsambladoGenerico* en Fig. 1), y persistencia de datos (*ExEnsambladoData* en Fig. 1). Además, es usual identificar en las implementaciones las siguientes características:

- Existen múltiples componentes en el nivel de presentación (UI). Un conjunto de productos de la familia (*ExFamilia* en Fig. 1) está formado por al menos dos ensamblados UI (*ExEnsambladoUI* en Fig. 1), esto implica múltiples escenarios de UI en donde se podrían observar clientes ricos, presentaciones web, escenarios móviles, sistemas tradicionales de escritorio o "plugins" para tecnologías específicas [8].
- La capa lógica de negocios está formada por ensamblados genéricos (*ExEnsambladoGenerico* en Fig. 1) de manera tradicional. Un ensamblado de estas características es categorizado como regular (*ExEnsambladoRegular* en Fig. 1) cuando puede ser utilizado en múltiples soluciones y exhibe un comportamiento uniforme en las distintas aplicaciones que componen la familia.
- Los componentes de persistencia (*ExEnsambladoData* en Fig. 1) explicitan la tecnología empleada para administrar la persistencia (*ExTecnologiaData* en Fig. 1), pudiendo implementar más de una tecnología posible.

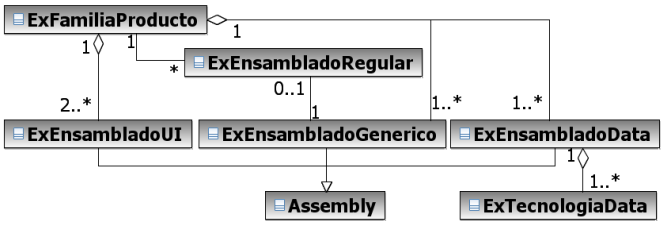

Fig. 1. Modelo de representación de familias de productos de software a medida.

3 Exadat

Sin una adecuada representación de la arquitectura de software, identificar posibles variantes a una familia de productos es un proceso complejo, o imposible de concretar. Esta propuesta consiste en aplicar ingeniería reversa a los ensamblados de productos para luego poder representar arquitectónicamente un producto de software. A partir de la representación arquitectónica es posible guiar diversas actividades de identificación de puntos de variación y sus variantes. La presente contribución se centra en el análisis de los ambientes implicados en la persistencia, identificando en los niveles superiores dependencias de distintos servicios de almacenamiento brindados por la capa de persistencia. Este análisis se estructura en cuatro fases: a) fase de incorporación, b) fase estructural, c) fase interna, y c) fase de

retroalimentación. Estas fases son detalladas a continuación en ambientes .NET [9], sin embargo, el método es extensible a otras tecnologías.

3.1 Exadat: Fase de Incorporación

La *fase de incorporación* inicializa el proceso incorporando los ensamblados que se desean analizar y clasificándolos en función de los distintos tipos de ensamblados. Esta fase se compone de las siguientes dos actividades:

- Incorporación de ensamblados: se incorporan a Exadat los ensamblados a analizar, es decir, se define una potencial familia de productos (*ExFamilia* en Fig. 1), y los distintos ensamblados que la componen.
- Clasificación de ensamblados: se clasifican los ensamblados incorporados a Exadat, determinando su tipo. Para esto se desensambla el código y a partir de sus características de persistencia, o no, se determina el tipo de ensamblado (*ExEnsambladoData*, *ExEnsambladoGenerico*, o *ExEnsambladoUI* en Fig. 1). Los ensamblados de persistencia son representados por *ExEnsambladoData* (Fig. 1) tienen características particulares con respecto a los ensamblados tradicionales. Los ensamblados *ExEnsambladoData* (Fig. 1) representan componentes implicados en la persistencia y poseen referencias a bibliotecas que permiten la interacción con bases de datos.

Estas actividades están soportadas por el modelo de componentes ensamblados introducido en la Fig. 2. El elemento base es el ensamblado (*Assembly* en Fig. 1, Fig. 2). Un ensamblado está formado por un conjunto de bibliotecas de enlace dinámico (*DLL* en Fig.2). Cada *DLL* posee un agrupamiento de elementos, denominado *Namespace* en Fig. 2. En un *Namespace* se definen las distintas clases que forman el ensamblado (*Class* en Fig. 2) y sus métodos (*ClassMethod* en Fig. 2).

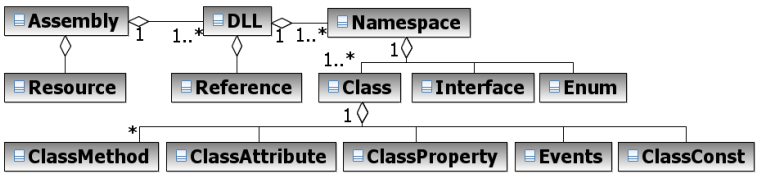

Fig. 2. Modelo de representación de los componentes ensamblados.

3.2 Exadat: Fase Estructural

En la *fase estructural* se analizan las dependencias entre los distintos ensamblados, y a partir de tal análisis se identifican diversos ambientes de persistencia que favorecen a la detección de regularidad de ensamblados. Finalmente, se analiza la regularidad de los ensamblados que participan en actividades de persistencia. En esta fase se involucran las siguientes actividades:

- Detectar dependencias en ensamblados de datos. Se analiza las dependencias estructurales de los ensamblados que participan del análisis. De cada uno de los

ensamblados del nivel de persistencia (*ExEnsambladoData* en Fig. 1) pueden depender ensamblados del nivel de lógica de negocio (arquitectura tres o más capas), o de los niveles de presentación (arquitectura dos capas). Para reflejar las dependencias arquitectónicas entre ensamblados se utiliza una matriz de dependencia estructural (DSM) [10].

- Detectar ambiente de persistencia. Se identifican y analizan diversos tipos de escenarios de representación: estructurales o lógicos, directos o derivados.
    - Los ambientes estructurales poseen dos o más ensamblados de datos implicados en la persistencia.
    - Los ambientes lógicos tienen un único ensamblado de datos que encapsula en su comportamiento a dos o más tecnologías de persistencia, nativas o externas.
    - Los ambientes directos tienen la particularidad de que el ensamblado implicado en la persistencia dependen directamente de ensamblados de presentación.
    - Los ambientes derivados tienen como característica que del ensamblado implicado en la persistencia depende al menos un ensamblado genérico, generalmente implicado en lógica intrínseca de la solución de software, del cual depende ensamblados de presentación.

La Fig. 3 presenta el modelo conceptual empleado para representar los distintos ambientes implicados en la persistencia. El concepto *ExADataEstructural* (Fig. 3) define un ambiente estructural caracterizado por el empleo de más de un ensamblado de persistencia (*ExEnsambladoData* en Fig. 3), y *ExADataLogico* (Fig. 3) define el ambiente lógico, el cual está formado por un único ensamblado del tipo *ExEnsambladoData* (Fig. 3) pero con diversas tecnologías de persistencia. Un ambiente *ExADataEstructural* (Fig. 3) se especializa en: *ExAEstructuralDirecto* (Fig. 3), cuando la interacción desde la presentación a *ExEnsambladoData* (Fig. 3) se visualiza en un nivel, con una dependencia directa; y *ExAEstructuralDerivado* (Fig. 3), cuando la interacción desde la presentación se visualiza con dos o más niveles en donde participan *ExEnsambladoGenerico* (Fig. 3). La Fig. 4 (a) ilustra un ejemplo de estructural directo, es identificado directo ya que *ExEnsambladoUI* depende directamente de los ensamblados *ExEnsambladoData* (Fig. 3) y además es identificado como estructural porque justamente los ensamblados implicados en la persistencia, consumidos desde ambientes de presentación, son dos o más. Un ejemplo de estructural derivado es incluido en la Fig. 4 (c).

En forma análoga un ambiente lógico (*ExADataLogico* en Fig. 3) se clasifica como *ExALogicoDirecto* (Fig. 3), o *ExALogicoDerivado* (Fig. 3). Ambos casos se ilustran en las Fig. 4 (b), y 4 (d), respectivamente. Estos casos son lógicos dado que poseen un único ensamblado implicado en la persistencia, pero estos ensamblados contienen dos o más tecnologías de persistencia de datos. Este aspecto es representado en Exadat por medio del concepto *ExTecnologiaData* (Fig. 3) que se especializa en *ExTecnologiaDataNativa* y *ExTecnologiaDataExterna* (Fig. 3). El primero representa la tecnología propia de cada entorno de ejecución, en cambio, el último modela una tecnología importada al entorno de ejecución utilizado.

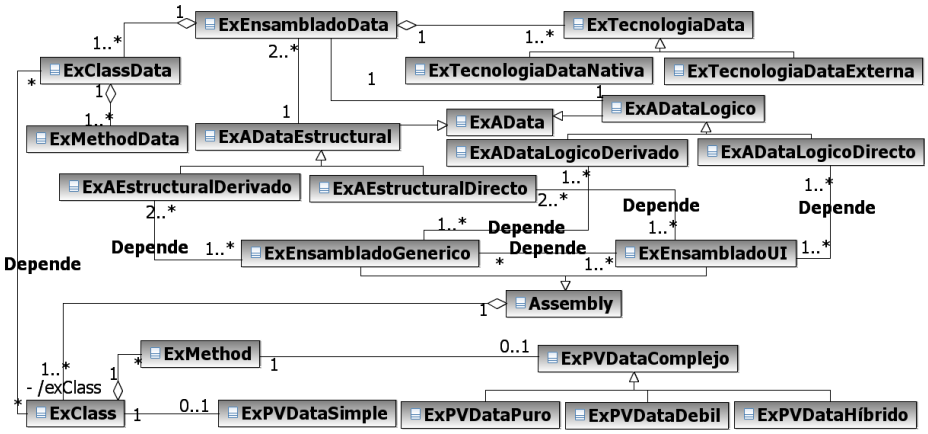

Fig. 3. Modelo conceptual empleado para representar los distintos ambientes implicados en la persistencia.

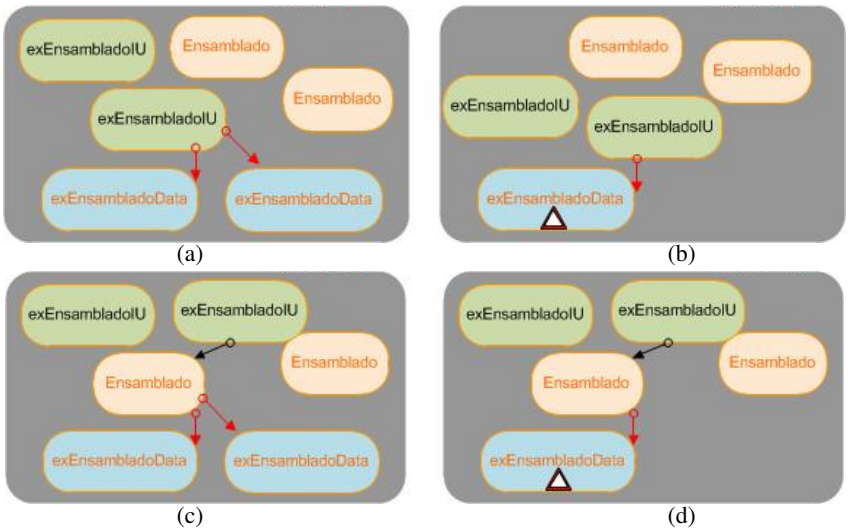

Fig. 4. Ambientes de persistencia: (a) estructural directo, (b) lógico directo, (c) estructural derivado, y (d) lógico derivado.

3.3 Exadat: Fase Interna

En la *fase interna* se analiza la composición interna del ambiente detectado. Se exploran las clases que componen los ensamblados para identificar clases y métodos implicados en la persistencia, representados como *ExClassData* (Fig. 3) y *ExMethodData* (Fig. 3), respectivamente. Durante esta fase se realizan las siguientes actividades:

- Identificación de clases de tipo *ExClassData* (Fig. 3). Se analiza cada una de las clases que forman parte de *ExEnsambladoData* (Fig. 3) buscando referencias a bibliotecas nativas de datos en la declaración de la clase, si el resultado es positivo se valida que estas referencias se estén utilizando en el código para ser clasificadas como *ExClassData* (Fig. 3).
- Identificación de métodos *ExMethodData* (Fig. 3) de las clases *ExClassData* (Fig. 3). En este paso se analiza el tipo de métodos de cada una de las clases *ExClassData* (Fig. 3) determinando cuales de los mismos están implicados en la persistencia, a estos métodos los clasificamos como *ExMethodData* (Fig. 3).
- Analizar propagación de comportamiento de persistencia en clases. En función del ambiente identificado en la etapa previa se analizan las llamadas a las clases *ExClassData* (Fig. 3) desde las clases (*ExClass* en Fig. 3) que componen a los ensamblados genéricos *ExEnsambladoGenerico* (Fig. 3) o de UI (*ExEnsambladoUI* en Fig. 3). Al encontrar que una clase en alguno de los ensamblados genérico o de UI posee llamadas a dos o más clases en ensamblados de persistencia diferentes se identifica un punto de variación, el cual es considerado simple (*ExPVDataSimple* en Fig. 3), representando un ambiente estructural *ExADataEstructural* (Fig. 3). Otra opción es identificar que una clase, en los ensamblados genéricos o de UI, posee llamadas a dos o más clases en el mismo ensamblado pero de tecnología diferente. En este caso también se identifica un punto de variación simple (*ExPVDataSimple* en Fig. 3) pero representado en un ambiente lógico *ExADataLogico* (Fig. 3).
- Analizar propagación de comportamiento de persistencia en métodos. Al identificar puntos de variación simples en ambientes lógicos o estructurales, Exadat analiza internamente las clases clasificadas como puntos de variaciones simples. Se analiza cada una de las clases clasificadas como *ExPVDataSimple* (Fig. 3) de un ensamblado genérico (*ExEnsambladoGenerico* en Fig. 3) o de un ensamblado UI (*ExEnsambladoUI* en Fig. 3), dependiendo del ambiente en el que esté implicado el punto de variación simple. Luego, en cada una de estas clases se identifican los métodos (*ExMethod* en Fig. 3) que realizan llamadas a métodos de las clases de los ensamblados de persistencia (*ExMethodData* en Fig. 3). En caso de desencadenarse desde un método llamadas a dos o más métodos del nivel de persistencia (*ExMethodData* en Fig. 3), identificamos al método (*ExMethod* en Fig. 3) como un punto de variación complejo. Por último, se analiza si el método está sobrecargado. Si entre las llamadas identificadas no existe sobrecarga, el punto de variación es identificado como *ExPVDataDebil* (Fig. 3). Si existe sobrecarga para todas las opciones implicadas, el punto de variación es identificado como *ExPVDataPuro* (Fig. 3). En cambio, si existe sólo para algunas operaciones, el punto de variación es identificado como *ExPVDataHíbrido* (Fig. 3).

3.4 Exadat: Fase de Retroalimentación

En la *fase de retroalimentación* presenta al arquitecto los puntos de variaciones y las variantes de persistencia identificadas en la fase previa. Su representación visualiza un mapa que favorece la identificación de persistencia *ExEnsambladoData* (Fig. 3) y

su importancia dentro del ambiente arquitectónico de una solución de software. El objetivo de esta representación es brindar soporte a la toma de decisiones sobre la variabilidad implicada en persistencia analizando en dos niveles distintos. Un primer nivel considera las clases y se identifican puntos de variaciones simples (*ExPVDataSimple* en Fig. 3). El otro nivel identifica puntos de variaciones complejos (*ExPVDataComplejo* en Fig. 3) en los métodos de las clases.

4 Caso de Estudio, Ecommerce Suite

El caso de estudio se realiza sobre un conjunto de productos basados en el concepto de e-commerce. Para la realización de la implementación de Exadat se utilizaron las herramienta Reflector .NET y Enterprise Architect. La tecnología de desarrollo del caso de estudio utilizada es Dotnet Framework 3.5/4.0 con lenguaje de implementación C#. Por razones de espacio y claridad los ensamblados seleccionados forman parte de un conjunto reducido de la plataforma comercial completa.

4.1 Caso de Estudio: Fase Incorporación

En la Fase de incorporación, particularmente, en la actividad de incorporación de ensamblados, se utilizaron los elementos referenciados en la Tabla 1, cada uno de estos elementos forma parte de la una Suite comercial por lo tanto se ha seleccionado el prefijo "Ecommerce" para su nomenclatura.

Tabla 1. Elementos ensamblados que forman parte de la Ecommerce Suite Analizada.

EcommerceBusiness.dll	EcommerceMobile6.dll
EcommerceCarroDeCompras.dll	EcommercePago.dll
EcommerceControlesPersonalizados.dll	EcommerceEscritorio.dll
EcommerceData.dll	App_Web_clientes.aspx.257ca45b.dll
EcommerceFachada.dll	

4.2 Caso de Estudio: Fase Estructural

Se clasificaron los ensamblados buscando elementos de persistencia (*ExEnsambladoData* en Fig. 3), elementos genéricos (*ExEnsambladoGenerico* en Fig. 3) y de interfaz de usuario (*ExEnsambladoUI* en Fig. 3). El resultado se presenta en la Fig. 5. En la figura se incluyen los ensamblados de presentación (*ExEnsambladoUI* en Fig. 5) *App_Web_clientes.aspx.257ca45b.dll*, *EcommerceEscritorio.dll*, y *EcommerceMobile6.dll* y el ensamblado de persistencia *EcommerceData.dll* (Fig. 5). Además, se ilustra la identificación del ensamblado implicado en la persistencia (*EcommerceData.dll*). A partir de este ensamblado se inicia el mecanismo de detección de dependencias implicadas en la persistencia de Exadat, identificándose los ensamblados ilustrados en Fig. 6. La Fig. 6 muestra el entorno lógico derivado que se detecta en el caso de estudio propuesto.

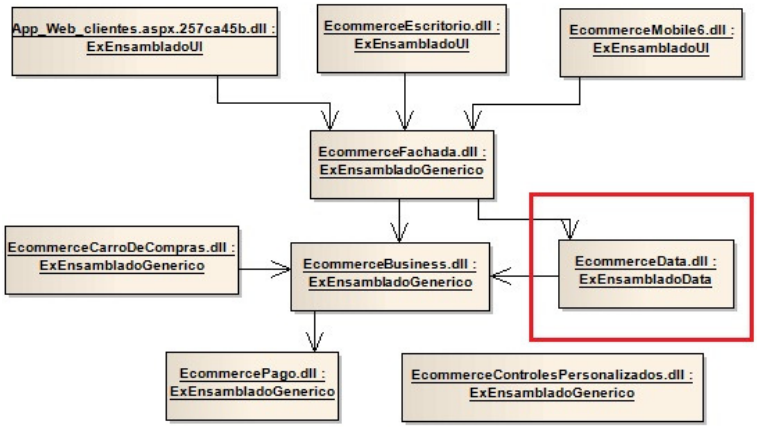

Fig. 5. EcommerceData.dll - Ensamblado de Persistencia – exEnsambladoData.

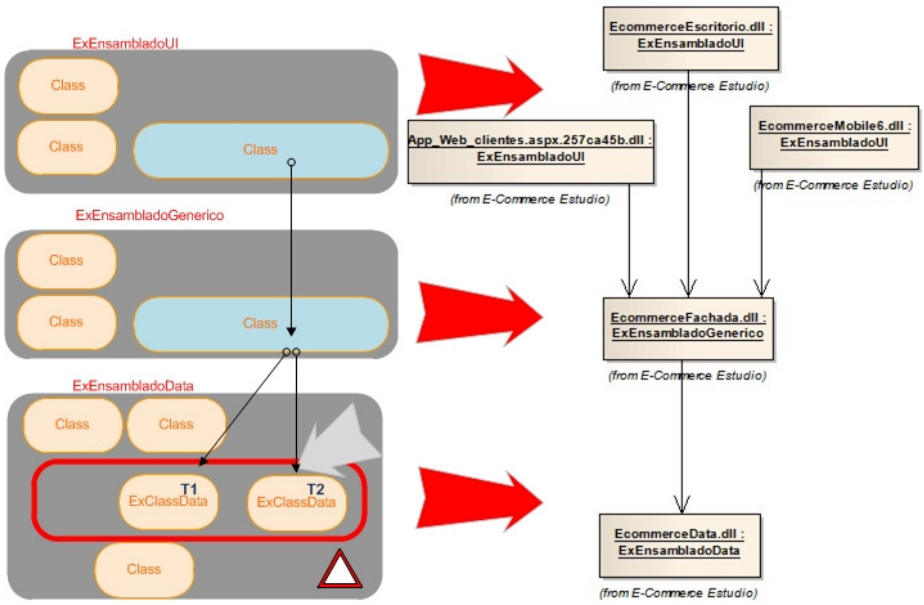

Fig. 6. EcommerceData.dll - Ensamblado de Data – exEnsambladoData

4.3 Caso de Estudio: Fase Interna

En esta fase se analiza la estructura interna del ambiente detectado en la fase estructural (en este caso un ambiente lógico). Por razones de espacio en la descripción del caso de estudio se considera parcialmente los elementos clasificados como *ExClassData*. En particular, sólo se consideran las clases *SQLClienteDataAccessObject.cs* y *OracleClienteDataAccessObject.cs* del ensamblado *EcommerceData.dll*. De estos dos elementos seleccionados, en la

actividad de análisis de la propagación de comportamiento de persistencia en métodos se visualiza el método (*ExMethodData*) *GetClientes()* de la clase *SQLClienteDataAccessObject.cs* (Fig. 7). Este método muestra un comportamiento de sobrecarga que permite categorizar al método *GetClientes()* de la clase *FachadaCliente* como un punto de variación puro*, ExPVCompletoPuro*.

```
[DataObject(true)]
    public class FachadaCliente {
        ...
        [DataObjectMethod(DataObjectMethodType.Select)]
        public IList<Cliente> GetClientes() {
                  return objClienteDataAccessObject.GetClientes();
        } ...
[DataObjectMethod(DataObjectMethodType.Select)]
public IList<Cliente> GetClientes(string ExpOrdenamiento,string
tipoPresentacion) {
            switch (tipoPresentacion) {
                case "web":
                    return objClienteDataAccessObject.GetClientes();
                case "wap":
                    return objClienteDataAccessObject.GetClientes();
                case "escritorio":
                    return objClienteDataAccessObjectOracle.GetClientes();
                case "mobile":
                    return objClienteDataAccessObject.GetClientes();
                default:
                    return objClienteDataAccessObject.GetClientes();
         }} ...
```

Fig. 7. Método "GetClientes()" dentro de la clase "FachadaCliente"

En La Fig. 7 se expone el punto de variación *ExPVCompletoPuro* en el ambiente derivado lógico, particularmente sobre una clase *FachadaCliente* dentro de un ensamblado *ExEnsambladoGenerico* (*EcommerceFachada*). Se resaltan dos variantes determinadas por la utilización de dos clases de tipo *ExClassData* (referenciadas por los objetos *objClienteDataAccessObject* y *objClienteDataAccessObjectOracle*), ubicadas en el *ExEnsambladoData EcommerceData*, cada una con tecnología diferenciada. En estas clases se utilizan métodos sobrecargados y esto determina el tipo de variación puro.

4.4 Fase de Retroalimentación

En esta fase se visualiza el ambiente de representación de variabilidad detectado. Para la visualización se creó un prototipo, *Exadat Assembly Graph*, que visualiza la

variabilidad en ambientes de persistencia derivados del método Exadat. Estas aplicaciones emplean las herramientas *Reflector.NET, Code Model Viewer, DSM y Code Analyzer*. La Fig. 8 ilustra a nivel estructural las dependencias entre los ensamblados analizados, resaltando en rojo al ensamblado que contiene el punto de variación y en azul al ensamblado que posee las variantes en el nivel de persistencia.
Luego, es posible profundizar en otras vistas de la herramienta, que permiten remarcar a nivel de clases y métodos los puntos de variación representados en el presente trabajo.

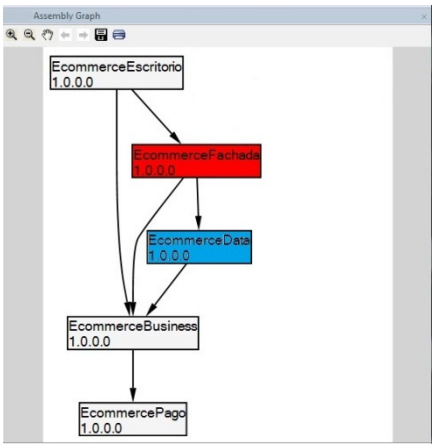

Fig. 8. Representación de las dependencias entre ensamblados y los puntos de variaciones identificados en *Exadat Assembly Graph*.

5 Conclusiones

El presenta trabajo propone un método para brindar soporte a la identificación de puntos de variación y sus respectivas variantes en un contexto donde no existe una representación concreta de las arquitecturas de productos de software. Se propone un enfoque al reuso productivo, la extensibilidad a diversos ambientes implicados en la persistencia y la transición suave arquitectónica. Se lograron resultados exitosos en más de cuarenta escenarios reales, exponiendo parcialmente un caso en la Sección 4. En este caso en particular se identificó un ambiente de persistencia lógico y puntos de variaciones puros. Esto brinda soporte a la toma de decisiones de los ingenieros de software. Generalmente los arquitectos de software deben decidir cómo extender productos de software que constan de millones de líneas de código a diversos ambientes de presentación, o en muchas ocasiones extender un producto de software a un ambiente nuevo de persistencia. Exadat brinda un primer soporte de representación de la variabilidad en ambientes implicados en la persistencia requerida por los arquitectos de software. Con esta información de soporte se pueden determinar líneas arquitectónicas que favorezcan la tendencia al desarrollo de familias de productos.

La identificación de puntos de variabilidad donde se consumen elementos de persistencia desde una vista estructural y lógica permite brindar soporte a decisiones arquitectónicas implicadas en la persistencia de ambientes emergentes, como lo es la utilización de escenarios "datastore" en soluciones de "cloud computing". Actualmente, se tiende a utilizar servicios en la nube. El identificar dónde se encuentran los puntos de variaciones candidatos a poder consumir un ambiente de persistencia en la nube (variante) es de gran valor a los arquitectos de software.

El método propuesto ha sido implementado en un ambiente Microsoft con su plataforma .NET. Sin embargo, se plantea extender su aplicación a diversas tecnologías en trabajos futuros, principalmente J2EE.

Se reconocen algunas limitantes del método que no están incluidas en el presente trabajo, planificadas para abordar en trabajos futuros, como es la utilización de ensamblados en ambientes de web services entre las capas de presentación y datos de los productos de software.

Referencias

1. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software families: a case study. Journal of System and Software, 74, 173--194 (2005)
2. Stoiber, R., Glinz, M.: Feature Unweaving: Efficient Variability Extraction and Specification for Emerging Software Product Lines. In: Fourth International Workshop on Software Product Management (IWSPM), pp. 53--62. IEEE Press (2010)
3. Arcelli, F., Tosi, C., Zanoni, M., & Maggioni, S.: The MARPLE Project – A Tool for Design Pattern Detection and Software Architecture Reconstruction. In: Proc. of ECOOP International Workshop on Advanced Software Development Tools and Techniques. Elsevier Science (2008)
4. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
5. Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer (2007)
6. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Elsevier Science (2003)
7. Kim, S.D., Her, J.S., Chang, S.H.: A theoretical foundation of variability in component-based development. Information and Software Technology, 47, 663--673 (2005)
8. Bortolotti, N., Gonnet, S., Leone, H.: Exavar: Análisis de Variabilidad de Productos de Software.In: XXXVI Conferencia Latinoamericana de Informática (2010)
9. Microsoft. Microsoft .NET (2010), http://www.microsoft.com/net/
10. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: Proc. of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 167--176 (2005)

Agradecimientos. Este trabajo ha sido financiado en forma conjunta por CONICET, la Universidad Tecnológica Nacional y la Agencia Nacional de Promoción Científica y Tecnológica (PAE-PICT-2007-02315, IP-PRH 2007). Se agradece el apoyo brindado por estas instituciones.