PIP: An Injection Pattern for Inserting Privacy Patterns and Services in Software

Naureen Ali[1], Dawn Jutla[2(✉)], and Peter Bodorik[1]

[1] Faculty of Computer Science, Dalhousie University, Dalhousie, India
{NaureenAli,Bodorik}@cs.dal.ca
[2] Sobey School of Business, Saint Mary's University, Halifax, NS, Canada
Dawn.jutla@smu.ca

Abstract. Increasingly, software engineers in organizations complying with privacy regulations are looking for repeatable ways to embed privacy in their code. We propose the concept of a Privacy Injection Pattern (PIP) for software engineers to use to automate dynamically "injecting" existing privacy patterns in existing or new code. The PIP is composed of a novel tri-abstraction combination of aspect-oriented programming, dependency injection, and mocking. Related work reveals fragmentation in using the software engineering abstractions separately to address privacy, as well as an absence of software injection patterns for privacy. We illustrate our new Privacy Injection Pattern and the simplicity of its implementation with a use case, and downloadable example code, that *injects* well-known de-identification patterns in a banking application. Adoption of our higher-level privacy injection pattern is expected to help software engineers comply more readily with Privacy by Design principles and to enable Privacy by Default. Early evaluation results for the PIP from practising software engineers are yet inconclusive.

1 Introduction

According to Alexander [1], a "pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Privacy patterns [1–4] in software engineering categorize sets of privacy requirements, and their relationships with system architecture and implementation, into repeatable design groupings that may be applied across software applications. Theoretically, and in practice software engineers' productivity improve with the recognition and use of repeatable patterns.

Numerous privacy patterns exist. For example, Kalloniatis et al. [2] identify authorization, authentication, data protection, anonymization and pseudonymization, unobservability, and unlinkability privacy process patterns. Porekar et al. [3] classify organizational privacy patterns as: Obtaining explicit consent" and "Access control to sensitive data based on purpose", "Time limited personal data keeping", "Maintaining privacy audit trails", "Creating privacy policy, Maintaining (versions of) privacy policies", and Privacy negotiation. Doty and Gupta [7] discuss a privacy policy as a pattern and reference Hoepman's work [19] on privacy strategies and categorization of privacy patterns. Others too

B. Berendt et al. (Eds.): APF 2015, LNCS 9484, pp. 144–157, 2016.
DOI: 10.1007/978-3-319-31456-3_8

(e.g. [17, 39]) discuss collections of privacy patterns. Romanosky et al. [39] specify three privacy patterns (informed consent for web-based transactions, masked online traffic and minimal information asymmetry) for software to support individuals when performing some activity online.

Software patterns are also embedded in updated 2015 standard-track specifications and standards such as the Organization for the Advancement of Structured Information Standards' (OASIS) Security Assertion Markup Language (SAML), the XML Access Control Markup Language (OASIS XACML), the Enterprise Privacy Authorization Language (EPAL), Privacy by Design Documentation for Software Engineers (OASIS PbD-SE), the Privacy Management Reference Model and Methodology (OASIS PMRM) specification of its atomic privacy services, and the PRIPARE project.

Once a privacy pattern is identified as per the above approaches, the pattern or its service implementation still has to be "injected" into existing or new software. This injection issue has not been addressed using patterns in the literature. While it is comparatively simpler to incorporate privacy in new applications, software engineers face challenges to implement even existing privacy patterns and their services' mappings in existing applications without affecting other software modules. In some cases, software engineers would prefer to avoid the recompilation and re-deployment of complex programs, such as found in financial and healthcare systems.

To improve software engineers' productivity, we describe a novel master pattern for privacy pattern injection. To the best of our knowledge, a privacy master-pattern for *automating injection of privacy patterns and their mapped privacy services in software* did not exist before this work. The pattern may be used in distributed SOA, cloud, mobile, as well as in non-web services environments, such as desktop and many existing client-server and legacy applications. With the approach described in this paper, privacy can be incorporated in an existing system without modifying its code, or in some cases modifying the code to a very small extent. In further paper sections, we describe our new privacy injection pattern, demonstrate our privacy injection pattern on a banking use case, present related work that highlights the fragmentation and gaps that exist in the privacy patterns universe, and provide a summary and conclusions.

2 Proposal for a Privacy Injection Pattern

A key technical challenge is to *automatically* inject a privacy pattern with its component implementation services in existing software without breaking its functionality and undermining its performance. A complex existing system, for example, should not be altered, or if required, modifications should be minor and minimally affect other existing modules or logic.

To inject privacy in architectures without modifying the existing code, we propose to combine three software engineering abstractions: a mocking framework, dependency injection (DI) pattern, and aspects as defined in aspect-oriented programming [23]. These three concepts exist independently, but have not been composed into a master-pattern before now for use by software engineers to nimbly embed privacy controls in applications.

Table 1 briefly discusses each of the three key techniques in our unifying Privacy Injection Pattern to support software engineers to conduct rapid automated embedding of privacy services in code.

Table 1. Combining aspect-oriented programming, dependency injection and mocking for privacy engineering

Software engineering technique	Terminology and traditional uses
Aspect-Oriented Programming (AOP)	Aspect-oriented programming (AOP [24]) is a programming technique to separate crosscutting concerns, such as privacy, in a unit of modularization called aspects, instead of fusing them with core modules as is traditionally done in object oriented programming.
Mocking	A mock object or isolation framework is implemented as a reusable library. It provides a way to create and configure fake objects at runtime. Isolation frameworks are widely used in test driven development (TTD). The use of dynamic fake object eliminates the need to write classes or provide the implementation of the interfaces.
Dependency Injection (DI)	The concept of dependency injection is based on the inversion of control (IoC) design pattern. IoC is a technique that assigns the responsibility of flow of control of an application to a container or a class [30] Dependency injection is mostly used for loosely coupled designs. It is commonly used for unit testing and validation/exception management [13].

One of the principles of software engineering is that each element of the program (class, method, procedure etc.) should focus on one task and one task only, aka separation of concerns. According to Sommerville (2011), concerns can be defined as "*something that is of interest or significance to a stakeholder or a group of stakeholders*". Core concerns are the software system's primary functionalities and purposes, while cross-cutting concerns are those functionalities whose implementation is spread across different modules of the program. The idea of Aspect Oriented Programming was proposed mainly to resolve the issue of cross-cutting concerns [16]. Aspects are the

abstractions (such as subroutines, methods and objects) that can be used at several places in the program. For example, transaction logging can be implemented using an *aspect* that can be used wherever logging is required for any type of transaction. Aspects can be included before a method, after a method or when an attribute is accessed [41].

In the PIP superpattern, aspects implement known privacy patterns. Dependency injection may also be considered as a design pattern that is useful to reduce the complexity of the system [21]. Mocking has been successfully used to implement a pattern to introduce fake data to protect users' privacy. The PIP proposal generalizes mocking to allow injection of the universe of privacy patterns and not just fake data.

3 PIP: A Privacy Injection Pattern

Combining the three abstractions in Table 1, we develop a new privacy injection pattern to insert known privacy patterns or services in new and existing legacy applications. Figure 1 shows our proposed Privacy Injection Pattern to insert privacy services in a software application using mocking, DI and AOP. It describes our injection pattern's program flow (numbered as 1 to 9) through one pattern instance. The concepts intrinsic to PIP (i.e. combination of AOP, mocking and dependency injection) are extensible to multiple system architectures. However, tightly coupled architectures that lack modularity will require more of a privacy engineer's attention than the more extensible, interoperable, and robust SOA and n-tier architectures.

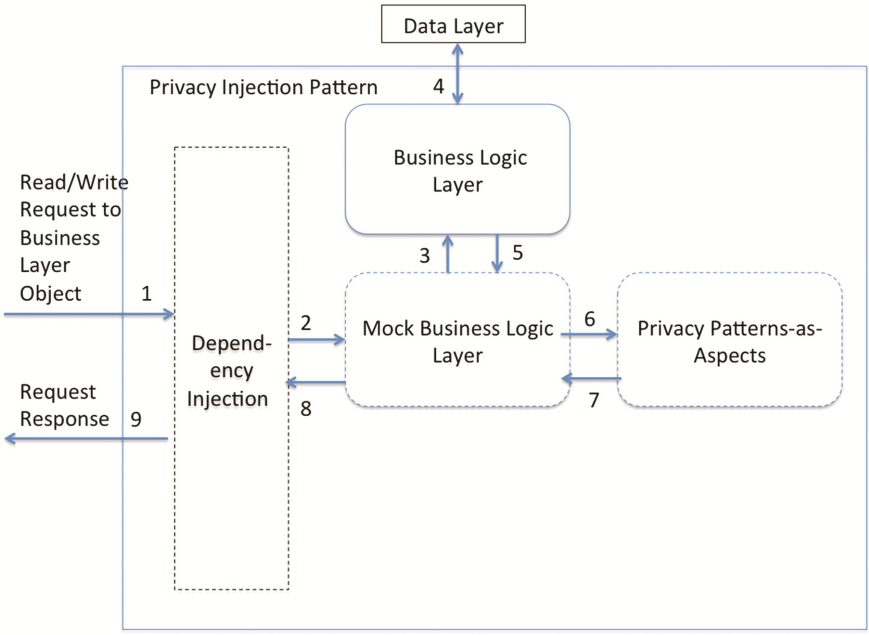

Fig. 1. Privacy injection pattern

Our Privacy Injection Pattern (PIP) implements other privacy-pattern classes in an aspect or privacy service component using AOP. As privacy is a cross-cutting concern across all software collecting or using personal data, software engineers may implement third-party privacy patterns or their components (e.g. masking, encryption) using AOP. Privacy aspects then may be used across software implementation classes.

When using PIP, at the beginning of a software program, software developers load a privacy service DLL (Dynamic Link Library), which consists of privacy pattern services implemented using AOP. An example of such a privacy pattern is obtaining explicit user consent. Dependency injection allows the engineer to load a privacy service DLL without recompiling existing services. A developer simply places the privacy DLL along with other DLLs and the privacy program will automatically load. When the program loads, a mock Business Application Logic (BAL) object of the same type as the original BAL object is created and injected by initializing it. In this way, when a software engineer calls any function of the BAL object (as triggered by (1) in Fig. 1), the mock BAL object function (2) loads. This mock object fetches data from the business layer as normal (3). For de-identification purposes, we use the mock object to apply third-party privacy aspects implementing privacy patterns (6), and to transfer the modified data to the presentation layer (9).

The software engineer may apply privacy patterns or services, implemented as aspects that cater for fine-grain privacy attributes such as role, locations, or any other environmental variables. Thus the PIP enables the software engineer to build rich privacy contexts.

The relationship of the PIP with system architecture includes support from a specialized Privacy Knowledge Base [6]. Generic Privacy Knowledge bases for organizations' applications contain information, such as described in [6, 7, 9]: description of personal data/data cluster, personal information category, personal data classification, source(s) of data, which applications collected the data, and which use them, the data collection method, the format(s) of the data and data repository format(s), the purpose(s) of collection, transfer of data to data minimization or de-identification services, security control during data transfer, data retention policy, and data deletion policy. Currently, privacy engineers and software engineers in firms such as Nokia and Microsoft collect the above Privacy Knowledge Base (PKB) information in spreadsheets, and developers reference these documents when building their software. The PKB is another area for more sophisticated automation that we are currently working on.

4 PIP Implementation and Early Evaluation

To illustrate ease of use and simplicity of implementation of our composite Privacy Injection Pattern (PIP), we employ the PIP in a use case scenario from a banking application that uses de-identification patterns. Data de-identification is a privacy-preserving technique. It is the process of de-identifying sensitive data by removing or transforming information in such a way that we cannot associate a piece of information with an identifiable individual [10, 11, 28, 40]. Some identification techniques are substitution, shuffling, nulling out, character masking and cryptographic techniques. We implement the nulling out and character

masking privacy patterns for illustration using AOP in our example. We show that the mocking and the dependency injection techniques automatically inject the AOP instance of the de-identification service.

Our technical implementation uses Visual Studio.Net (IDE), PostSharp (AOP), the Unity Container (Dependency Injection) and the Mock library to realize an example injection of our de-identification service into a banking application. We note that the PIP may be implemented with other technologies, e.g. multi-platform heterogeneous technologies. This example's implementation code may be downloaded from https://web.cs.dal.ca/~naureen/BankExample.

The banking application's use case scenario contains account information that shows individual and account details. We use two roles, manager and operator, to study the behavior of the system before and after applying the proposed pattern. In this case study, we inject the role-based de-identification pattern for access control such that the operator can view only some information while the manager can view all information.

The de-identification service DLL is loaded in the main program. Figure 2 shows the implementation of this added function to load the de-identification service DLL and initialize the de-identification service. This function is required for desktop-based applications. For web-based application, the software developer simply places the privacy DLL with other DLLs.

```
private static void InjectLibraries()
{
    var anonymizationServiceLibName = "SampleBank.AnonymizationService.dll";

    var currentPath = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);

    var anonymizationServiceLibCompletePath = Path.Combine(currentPath, anonymizationServiceLibName);

    if (!File.Exists(anonymizationServiceLibCompletePath))
    {
        return;
    }

    Assembly assembly = Assembly.LoadFrom(anonymizationServiceLibCompletePath);
    var anonymizationServiceType = assembly.GetType("SampleBank.AnonymizationService.AnonymizationService");

    var serviceInstance = Activator.CreateInstance(anonymizationServiceType);

    anonymizationServiceType.InvokeMember("Initialize", BindingFlags.Default | BindingFlags.InvokeMethod, null, serviceInstance, null);
}
```

Fig. 2. Load de-identification service DLL for desktop applications

When the de-identification service initializes, it creates a mock object of the same type as our business layer object. In our case, our business layer object is CustomerManager, which is an implementation of the ICustomerManager interface. CustomerManager has a method GetCustomer that fetches customer and account details from the database. The software engineer creates a mock object of the ICustomerManager type and then registers it.

The engineer also setups the updated implementation of the GetCustomer method to fetch customer and account details in the same way as the originating object method, and then applies the *de-identification aspect* on this object.

Figure 3 shows the de-identified GetCustomer implementation. Subsequently, when the developer calls CustomerManager.GetCustomer, the updated GetCustomer method is invoked. In the Unity Container, for dependency injection the software engineer first registers the object at the beginning of the program in order to resolve the object to access its methods.

```
public static void Initialize()
{
    SetupCustomerManager();
}

public static void SetupCustomerManager()
{
    if (SampleBank.Common.Ioc.IocContainer.Instance.IsRegistered(typeof(ICustomerManager)))
    {
        return;
    }

    var customerManagerMock = new Mock<ICustomerManager>();

    customerManagerMock.Setup(x => x.GetCustomer()).Returns(() => {

        var customerMgr = new CustomerManager();
        var result = customerMgr.GetCustomer();

        return new CustomerInfoAnonymizedImpl(result);
    });

    SampleBank.Common.Ioc.IocContainer.Instance.Register<ICustomerManager>(customerManagerMock.Object);
}
```

Fig. 3. Inject mocking object and invoke IOC

Figure 4 shows how the software developer resolves the ICustomerManager object to fetch customer information. The developer will call the GetCustomer function to fetch the required information. This action calls the mock object's GetCustomer method and applies the de-identification service on the object. After applying de-identification, the system displays the information on the screen.

```
this.customerInfo = SampleBank.Common.Ioc.IocContainer.Instance.Resolve<ICustomerManager>().GetCustomer();

this.lblCustomerName.Text = this.customerInfo.BankUser.FirstName + " " + this.customerInfo.BankUser.MiddleName +" "
    + this.customerInfo.BankUser.LastName;

this.personalInformationUserControl.ShowBankUserInfo(this.customerInfo.BankUser);

if (SampleBank.Common.Ioc.IocContainer.Instance.Resolve<IRoleManager>().UserRole == Role.Operator)
{
    this.personalInformationUserControl.DisableAllControls();
}

this.accountsInfoUserControl.ShowAccounts(this.customerInfo.Accounts);
```

Fig. 4. Resolve mocking object at runtime to get customer information

We apply the de-identification service by creating a de-identification aspect with properties or methods. In our case, we apply de-identification on the properties. When we try to access the property, it applies the de-identification aspect on the field and returns a value.

```
[LongStringAnonymization(MaskCharacter = '*', VisibleStringLength = 5)]
public string AccountNumber { get; set; }
```

Fig. 5. Apply LongStringAnonymization aspect on AccountNumber

We apply LongStringAnonymization to the AccountNumber property (Fig. 5). In the LongStringAnonymization class, we provide the de-identification logic that will be applied on the field on which we bind as in Fig. 6. We implement the aspect classes for email, date, number, IDs and other fields and then apply these aspects to the properties or methods where required.

```
public override void OnGetValue(LocationInterceptionArgs args)
{
    base.OnGetValue(args);

    if (SampleBank.Common.Ioc.IocContainer.Instance.Resolve<IRoleManager>().UserRole == Role.Manager)
    {
        return;
    }

    string value = (string) args.Value;

    if (String.IsNullOrEmpty(value))
        return;

    if (this.HideFromFront)
    {
        if (value.Length <= this.VisibleStringLength)
            value = this.MaskCharacter.Repeat(this.VisibleStringLength);

        value = string.Format("{0}{1}", this.MaskCharacter.Repeat(value.Length - this.VisibleStringLength),
            value.Substring(value.Length - this.VisibleStringLength));
    }
    else
    {
        if (value.Length <= this.VisibleStringLength)
            value = this.MaskCharacter.Repeat(this.VisibleStringLength);

        value = string.Format("{1}{0}", this.MaskCharacter.Repeat(this.VisibleStringLength),
            value.Substring(0, value.Length - this.VisibleStringLength));
    }

    args.Value = value;
}
```

Fig. 6. De-identification implementation in LongStringAnonymization class

Figure 7 shows an operator screen of the sample bank application that results from the use of the PIP for injection of simple de-identification patterns. Recall the operator role does not have permission to view all the private information about the customer.

Different fields' data are de-identified using different de-identification techniques. For example, for the customer id field, we apply character masking; for date of birth, we use date variance; and we null out the street number.

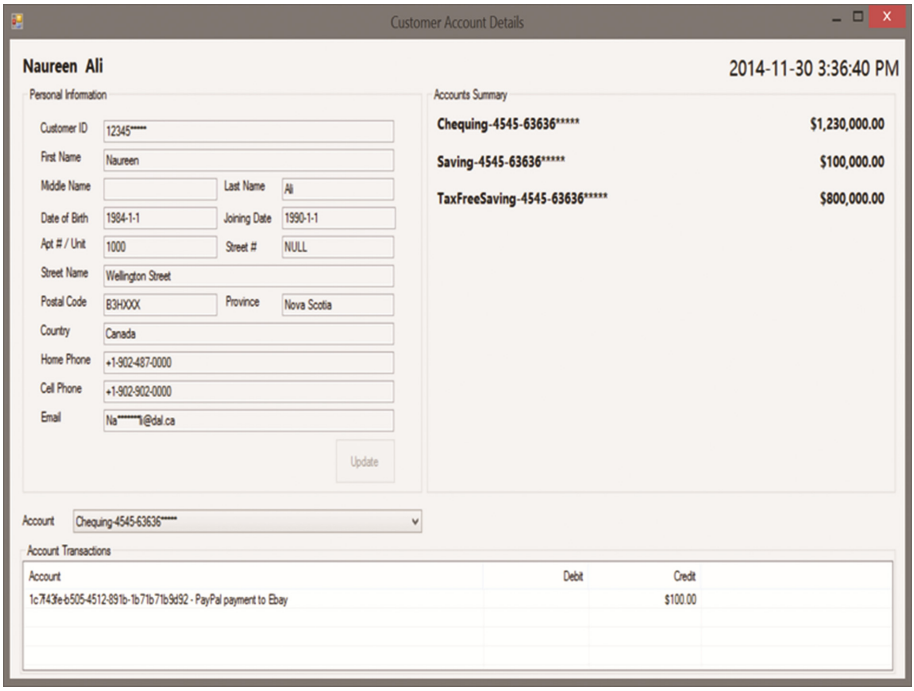

Fig. 7. Operator screen of sample bank application

The power of our new PIP pattern lays in its flexibility to inject any privacy pattern in existing code. The PIP pattern can be used repeatedly in many places in a banking application e.g. to also inject a location privacy pattern that disallows the operator from viewing even more of customers' fields from outside of banking hours.

One study that we are currently conducting to evaluate our proposal gives software engineers from large to small participating software organizations a task to embed (1) a simple privacy pattern, and (2) a complex privacy pattern in legacy software. Software engineers are first provided with guidance on using the PIP. They then evaluate the PIP using a validated Technology Acceptance Model (TAM) survey instrument. We sent out electronic surveys to software engineers in software multinationals such as IBM, Intel, Dell, and end user companies with software engineers (e.g. AT&T). We also sent the survey to small software engineering companies, such as Canada's Newpace.

To date, we have received 18 completed and usable responses. These preliminary responses show that the practicing software engineers evaluate the PIP pattern as easy to use. However, responses were mixed with respect to its perceived benefits. While respondents indicated across the board that the pattern would improve their productivity

when embedding privacy controls, they were ambivalent about the perceived benefits in general. The respondents did not provide us with outlines or descriptions of any or better alternatives. We are doing a follow-up evaluation exercise to determine whether their response around perceived benefits is due to incompatible technologies (e.g. most engineers in an organization not using the aspect-oriented paradigm), architectural standards and policies that exclude the use of mocking or injected third party code, fear of adding to complexity in the management of their development-operations environments, personal preferences to edit their existing code, lower workplace ranking for adding privacy requirements to code versus product feature requirements, or some other factor or combinations of factors.

5 Prior Work

The aspect-oriented programming (AOP) part of our tri-method privacy injection pattern has been used individually in the past to implement security and access control method extensions (e.g. [27, 44]). AOP has been used without automating privacy injection in code via use of mocking and dependency inversion. Sharma et al. [39] propose using AOP for the secure transfer of data over the Internet. They implement privacy patterns for encrypting/decrypting data and key generation using hashing as aspects performed by security agent. Win et al. [42] also use AOP for security and transmission privacy.

Chen and Wang [12] use AOP as a mechanism to implement privacy-aware access control. In their work, application-level access control is extended to enforce privacy policies on personal data using AOP with little impact on the structure of the application. Inter-type declaration (ITD) is used to link privacy preferences of a user with his/her PII, which is then provided to the access control aspect. Inter-type declaration aka member introduction is a mechanism that allows the programmer to modify class members/fields and relationships between classes. Privacy policies are implemented by comparing the purpose of request and the data subject's consent directive. The action manager is used to fetch the purpose of a request while for the data subject's consent or preferences, the preference aspect invokes a preference factory to fetch privacy preferences and link them with the requested data. Lastly, the access control aspect ensures that the requestor is an authorized user, has the authority to perform the requested action, and finally filters user's personally identifiable information (PII) according to privacy preferences attached to the PII.

Many researchers use dependency injection (DI) in their work. Benenson et al. [3] propose a smart card based framework for Secure Multiparty Computation (SMC). Their model consists of multiple processes having a security module that securely interacts with the security modules of other processes. The authors use DI to configure the component that selects the actual algorithm at runtime without recompiling the code. Livne et al. [26] present a health care architecture using dependency injection, AOP, and XML configurations to make the architecture flexible, reusable, loosely coupled and service-oriented. Similarly, Jezek et al. [21] use DI in their work. In their research, they propose a framework that may be used to improve the selection of the injection candidates from multiple candidates based on some extra-functional characteristics such as high performance, low memory

consumption etc. In related research, [2] propose a novel service called VAM-aaS (Vulnerability Analysis and Mitigation as-a-service) to mitigate the security vulnerabilities in the cloud environment. It analyzes the online services and in case of vulnerabilities generates a script to block the services or application that can be vulnerable. A list of mitigation actions is maintained by the system. In case of a particular vulnerability, the vulnerability mitigation component injects calls to the security handler classes at runtime based on the required mitigation actions of that vulnerability.

Bender and McWherter [4] use the term "mock" to refer to a family of similar implementations to replace real external resources during unit testing. Indeed, mocking is used primarily during the testing phase of software engineering. It has not been used to automate sophisticated privacy injection patterns in the past, but to provide a simple fake-data pattern to applications to preserve privacy. Beresford et al. [5] propose a modified version of the Android operating system called MockDroid to mock resources accessed by an application. For example, in an application that requests IP connectivity, location data, read-write access to calendar data, the user may provide mock data instead of actual data to the application. Hornyack et al. [20] and Zhou et al. [43], also propose to provide fake or empty data to software applications that require access to users' personal data. A user may view all the permissions that an application requests at the time of installation of the application and then select one of the four modes (trusted, anonymous, bogus, or empty) for each of the permissions.

The OASIS Privacy Management Reference Model and Methodology (PMRM) [8] propose eight atomic privacy services that may be mapped to privacy patterns: Agreement, Validation, Certification, Security, Access, Enforcement, Interaction, and Usage. An Accountability service is recently proposed for addition to the PMRM suite. Doty and Gupta [14] discuss a privacy policy as a pattern with reference to Hoepman's work [19] on privacy strategies and categorization of privacy patterns.

The closest work to this paper's in terms of privacy pattern injection comes from the same research group in Bodorik et al.'s [7] Privacy Architecture for Web Services (PAWS) work that semi-dynamically injects privacy web services for notice and consent in existing web pages built on ROA architectures. Software patterns that *fully automate injecting* privacy patterns are not found in the literature. The literature discussed in this section shows that our abstraction-unifying, higher-level Privacy Injection Pattern helps remove fragmentation from the software engineering landscape for privacy. We expect that patterns for privacy, its constructs, and desirable properties (e.g. unlinkability and unobservability at the data level, and the 7Cs at the user level such as comprehension, choice, consent, consciousness, consistency, confinement, and context [22] - will become increasingly available, as policy levers, such as Privacy by Design in regulations, begin to work.

6 Summary and Conclusions

Software engineers can inject other privacy patterns and their service representations in new, existing, and legacy systems without affecting existing systems using the PIP's comprehensive triad-pattern of aspect-oriented programming, dependency injection and mocking. Related work reveals the fragmentation of effort in using the abstractions individually and

separately to address privacy. The pattern unification of the three powerful software engineering abstractions to automate the embedding of privacy in applications is expected to increase the productivity of the software engineers tasked with complying with Privacy by Design principles and Fair Information Practices and Principles in code. We illustrate the simplicity of the PIP implementation in a de-identification scenario. This simplicity is at the crux of enhancing its chances of adoption by software engineers.

We will scientifically report on the human performance of our proposed PIP pattern in various use case contexts in future work. We choose to examine the human adoption of our new PIP pattern for two reasons. Not only does the state-of-the art in privacy engineering presently not lend itself readily to automated external verification, engineers' adoption of privacy tools is significant and essential to closing policy-technology gaps. The software engineer is an important stakeholder with respect to the privacy of software applications. Her/his education and the availability of tools in the privacy space remain a major key to progress for Privacy by Design and Default.

Acknowledgments. This material is supported by N. Ali's post-graduate scholarship from the Government of the Province of Nova Scotia, Canada, and D. Jutla's Federal Natural Sciences and Engineering Research Council of Canada (NSERC) grant for privacy and accessibility.

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Constructions. Oxford University Press, Oxford (1977)
2. Almorsy, M., Grundy, J., Ibrahim, A.S.: VAM-aaS: online cloud services security vulnerability analysis and mitigation-as-a-service. In: Wang, X., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 411–425. Springer, Heidelberg (2012)
3. Fort, M., Freiling, F.C., Penso, L.D., Benenson, Z., Kesdogan, D.: TrustedPals: secure multiparty computation implemented with smart cards. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 34–48. Springer, Heidelberg (2006)
4. Bender, J., McWherter, J.: Professional Test Driven Development with C#: Developing Real World Applications with TDD. Wrox Press Ltd, Birmingham (2011)
5. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: Mockdroid: trading privacy for application functionality on smartphones. In: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile, pp. 49–54 (2011)
6. Bodorik, P., Jutla, D.N., Dhillon, I.: Privacy compliance with web services. J. Inf. Assur. Secur. **4**(5), 412–421 (2009)
7. Bodorik, P., Jutla, D.N., Bryn, A.: Privacy engineering with PAWS: injecting RESTful privacy web services. Report - 2015–06, Faculty of Computer Science, Dalhousie University (2015)
8. Brown, P.F., Janssen, G., Jutla, D.N., Sabo, J., Willett, M.: Privacy management reference model and methodology (PMRM) version 1.0, OASIS Committee Specification 01, July 2013
9. Cavoukian, A., Carter, F., Jutla, D., Sabo, J., Dawson, F., Fieten, S., Fox, J., Finneran, T.: Annex guide to privacy by design documentation for software engineers version 1.0 OASIS committee note draft 01, 25 June 2014. http://docs.oasis-open.org/pbd-se/pbd-se-annex/v1.0/cnd01/pbd-se-annex-v1.0-cnd01.pdf. Accessed 30 April 2015

10. Cavoukian, A., Emam, K.E.: De-identification protocols: essential for protecting privacy, 25 June 2014. http://www.privacybydesign.ca/content/uploads/2014/09/pbd-de-identifcation-essential.pdf. Accessed 30 November 2014
11. Cavoukian, A., Emam, K.E.: Dispelling the myths surrounding de-identification: anonymization remains a strong tool for protecting privacy, June 2011. https://www.futureofprivacy.org/wp-content/uploads/2011/07/Dispelling. The myth surrounding de-identification anonymization remains strong tool for protectin privacy.pdf. Accessed 15 May 2015
12. Chen, K., Wang, D.-W.: An aspect-oriented approach to privacy-aware access control. In: Proceedings of the Sixth International Conference on Machine Learning and Cybernetics, pp. 3016–3021. IEEE, Hong Kong (2007)
13. Culp, A.: The dependency injection design pattern, 4 May 2011. Retrieved from MSDN: https://msdn.microsoft.com/en-us/library/vstudio/hh323705(v=vs.100).aspx
14. Doty, N., Gupta, M.: Privacy design patterns and anti-patterns. In: Patterns Misapplied and Unintended Consequences. Trustbusters Workshop at the Symposium on Usable Privacy and Security, July 2013
15. Fowler, M.: Inversion of control containers and the dependency injection pattern, 23 de January de 2004. Obtenido de Martin Fowler: http://martinfowler.com/articles/injection.html
16. Groves, M.D.: AOP in.NET Practical Aspect-Oriented Programming. Manning Publications Co., New York (2013)
17. Hafiz, M.: A collection of privacy design patterns. In: Proceedings of the Pattern Languages of Programs Conference (2006)
18. Haque, H.: A curry of Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) and IoC container, 12 de March de 2013. Obtenido de Code Project: http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inversion
19. Hoepman, J.-H.: Privacy design strategies. In: Cuppens-Boulahia, N., Cuppens, F., Jajodia, S., Abou El Kalam, A., Sans, T. (eds.) SEC 2014. IFIP AICT, vol. 428, pp. 446–459. Springer, Heidelberg (2014)
20. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: "These aren't the droids you're looking for": retrofitting android to protect data from imperious applications. In: 18th ACM Conference on Computer and Communications Security. ACM, Chicago (2011)
21. Jezek, K., Holy, L., Brada, P.: Dependency injection refined by extra-functional properties. In: IEEE Symposium on Visual Languages and Human-Centric Computing: Poster and Demos, pp. 255–256 (2012)
22. Jutla, D.N., Bodorik P.: Sociotechnical architecture for online privacy. In: IEEE Security and Privacy, vol. 3, no. 2, pp. 29–39, March–April 2005. doi:10.1109/MSP.2005.50
23. Kalloniatis, C., Kavakli, E., Gritzalis, S.: Using privacy process patterns for incorporating privacy requirements into the system design process. In: The Second International Conference on Availability, Reliability and Security, ARES 2007, pp.1009–1017 (2007)
24. Kiczales, G., Lamping, L., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M.: Aspect Oriented Programming, ECOOP 1997—Object-Oriented Programming, pp. 220–242 (1997)
25. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., New York (2003)
26. Livne, O.E., Schultz, N.D., Narus, S.P.: Federated Querying Architecture with Clinical and Translational Health IT Application. Springer Science + Business Media, USA (2011)
27. Mourad, A., Laverdière, M.-A., Debbabi, M.: An aspect-oriented approach for the systematic security hardening of code. Comput. Secur. **27**(3–4), 101–114 (2008)

28. Narayanan, A., Shmatikov, V.: Robust de-anonymization of large sparse datasets. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008)
29. Porekar, J., Jerman-Blazic, A., Klobucar, T.: Towards organizational privacy patterns. In: 2008 Second International Conference on Digital Society, pp. 15–19, February 2008
30. Prasanna, D.: Dependency Injection. Manning Publications Co., New York (2009)
31. Raghunathan, B.: Complete Book of Data Anonymization from Planning to Implementation. CRC Press Taylor and Francis Group, Boca Raton (2013)
32. van Rest, J., Boonstra, D., Everts, M., van Rijn, M., van Paassen, R.: Designing privacy-by-design. In: Preneel, B., Ikonomou, D. (eds.) APF 2012. LNCS, vol. 8319, pp. 55–72. Springer, Heidelberg (2014)
33. Romanosky, S., Acquisto, A., Hong, J., Cranor, L., Friedman, B.: Privacy patterns for online interactions. In: Proceedings of the Pattern Languages of Programs Conference (2006)
34. Sadicoff, M., Larrondo-Petrie, M., Fernandez, E.: Privacy-aware network client pattern. In: Proceedings of the Pattern Languages of Programs Conference (2005)
35. Schumacher, M.: Security patterns and security standards - with selected security patterns for anonymity and privacy. In: European Conference on PaBern Languages of Programs (EuroPLoP 2002)
36. Schümmer, T.: The public privacy – patterns for filtering personal information in collaborative systems. In: CHI 2004 (2004)
37. Seemann, M.: Dependency Injection in.NET. Manning, New York (2012)
38. Seemann, M.: Mock Objects to the Rescue! Test Your.NET Code with NMock. MSDN Magazine, October de 2004
39. Sharma, N., Batra, U., Mukherjee, S.: Enhancing security in service oriented architecture driven EAI using aspect oriented programming in healthcare IT. Int. J. Sci. Eng. Res. **5**(3), 50–55 (2014)
40. Shapiro, S.: Separating the baby from the bathwater - toward a generic and practical framework for anonymization. IEEE (2011)
41. Somerville, I.: Software Engineering. Pearson Education, UK (2011)
42. Win, B.D., Joosen, W., Piessens, F.: Developing secure applications through aspect-oriented programming. In: Aspect-Oriented Software Development, pp. 633–650. Addison-Wesley (2005)
43. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)
44. Zhu, Z.J., Zulkernine, M.: A model-based aspect-oriented framework for building intrusion-aware software systems. Inf. Softw. Tech. **51**, 865–875 (2009)