PIP: A (Privacy) Injection Pattern for Inserting Privacy Patterns and Services in Software

by

Naureen Ali

Submitted in partial fulfilment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
November 2015

Table of Contents

List of Tables .... **v**

List of Figures .... **vi**

Abstract .... **ix**

List of Abbreviations Used .... **x**

Acknowledgement .... **xi**

Chapter 1 : Introduction **1**

1.1 Research Problem 2

1.2 Research Objectives 3

1.3 Outline 3

Chapter 2 : Background and Related Work **4**

2.1 Privacy Overview 4

2.1.1 Privacy 4
2.1.2 Privacy Policy 6
2.1.3 Privacy Standards and Laws 7

2.2 Privacy Research 8

2.2.1 Privacy Engineering: 8
2.2.2 Patterns on Privacy 13
2.2.3 De-identification through Data Transformation Techniques 14
2.2.3.1 Character Masking Method 14
2.2.3.2 The Randomization Method 15
2.2.3.3 The k-anonymity Model 15
2.2.3.4 The l-diversity Model 17
2.2.3.5 The t-closeness Model 19
2.2.4 Aspect Oriented Programming 21
2.2.4.1 Core Concerns and Cross-Cutting Concerns 21
2.2.4.2 Aspect Oriented Programming Key Terminologies 23
2.2.4.3 AOP and Privacy 24
2.2.5 Dependency Injection 31
2.2.6 Mocking 38

2.3 Discussion 40

Chapter 3 : Privacy Services Injection Pattern **42**

3.1 Design Criteria 42

3.2 The Privacy Injection Pattern (PIP) 42

3.3 Architecture of the Privacy Injection Pattern (PIP) 44

3.4 Software Engineer's Learning Curve Perspective 47

3.4.1 Dependency Injection (DI) 47
3.4.2 Mocking 49
3.4.3 Aspect oriented programming 52

Chapter 4 : Prototype for Proof of Concept 56

4.1 Banking System Use Case 56

4.2 Hospital System Use Case 64

Chapter 5 : Results and Evaluation 74

5.1 Evaluation Methodology 74

5.1.1 Survey Objectives 76
5.1.2 Survey Questions 76
5.1.3 Survey Approach 76
5.1.3.1 Survey protocol 77
5.1.3.2 Survey instrument 77
5.1.3.3 Survey Design 77
5.1.3.4 Intended Data Analysis 79
5.1.3.5 Recruitment 80
5.1.3.6 Participants 80

5.2 Participants 80

5.3 Perceived Usefulness and Perceived Ease of Use 81

5.3.1 Qualitative Analysis 83
5.3.1.1 Reliability 83
5.3.1.2 Validity 85

5.4 Results for Perceived Usefulness and Ease of Use 88

5.5 Improvements in PIP Pattern 92

5.6 Overall Motivation to Use the Pattern 93

5.7 Limitations 94

Chapter 6 : Conclusions and Future Work 95

Bibliography **97**

Appendix A – Letter of Approval **107**

Appendix B – Recruitment Script **108**

Appendix C – Informed Consent **109**

Appendix D – Signature Page **112**

Appendix E – Survey Questions **113**

List of Tables

Table 2.1 Seven Types of Privacy [adopted from (Finn et al., 2013)] 5

Table 2.2. Approaches of Aspect Weaving [adopted from (Sommerville, 2011)] 24

Table 5.1. Demographics of the participants 81

Table 5.2. Mean, Standard Deviation and Cronbach's Alpha for Variables 84

Table 5.3. Cronbach’s Alpha of Perceived Usefulness and Perceived Ease of Use 84

Table 5.4. Correlation Matrix for Perceived Usefulness 86

Table 5.5. Correlation Matrix for Perceived Ease of Use 86

Table 5.6. Factor loadings after Varimax rotation 87

Table 5.7. Average of Scale of Perceived Usefulness and Perceived Ease of Use........... 89

List of Figures

Figure 2.1. Health related data [adopted from Zhong et al. (2005)] 15

Figure 2.2. 2-Anonymous health data [adopted from Zhong et al. (2005)] 17

Figure 2.3. Sensitive and non-sensitive information in health data 17

Figure 2.4. 4-anonymous health data [adopted from (Machanavajjhala et al., 2007)] 18

Figure 2.5. 3-diverse table [adopted from (Machanavajjhala et al., 2007)] 19

Figure 2.6. Salary/disease data [adopted from Li et al. (2007)] 20

Figure 2.7. 3-diverse salary/disease data [adopted from Li et al. (2007)] 20

Figure 2.8 Core concerns and cross-cutting concerns in Internet Banking System 22

Figure 2.9. Tangling and Scattering for Internet Banking System (adopted from [(Sommerville, 2011)]) 23

Figure 2.10. Privacy Control Access Control [adopted from (Chen & Wang, 2007)] 26

Figure 2.11. Life cycle and flow of personal data [adopted from (Berghe & Schunter, 2006)] 28

Figure 2.12 Architecture of theme-park location service [adopted from (Scheffler et al., 2012)] 29

Figure 2.13 Access permissions to the object on the basis of identity of the calling service [adopted from Scheffler et al. (2012)] 30

Figure 2.14. An AOP based reference monitor [adopted from Scheffler et al. (2012)] ... 31

Figure 2.15. Relationship between DIP, IoC and DI [adopted from (Haque, 2013)] 32

Figure 2.16 Tightly coupled classes and code 33

Figure 2.17 ClassA has reference of interface of ClassB 34

Figure 2.18 Dependency injected by Container class 37

Figure 3.1. Architecture of a 3 tier desktop application 43

Figure 3.2. Architecture for injecting privacy in legacy application 45

Figure 3.3. Register concrete type to Unity container 48

Figure 3.4. Instantiation of CustomerManager object 49

Figure 3.5. CustomerManager class and GetCustomer function 51

Figure 3.6. ICustomerManager interface 51

Figure 3.7. *customerManagerMock* implementation 52

Figure 3.8. LongStringDeidentification class 54

Figure 3.9. Application of LongStringDeidentification aspect class 55

Figure 4.1. Sequence Diagram of User Account before Applying PIP 57

Figure 4.2. Load de-identification service DLL for desktop applications 58

Figure 4.3. Inject mocking object and implementation of the *GetCustomer()* method 59

Figure 4.4. Resolve mocking object at runtime to get customer information 60

Figure 4.5. Apply *LongStringAnonymization* aspect on *AccountNumber* 60

Figure 4.6. De-identification implementation in LongStringAnonymization class 61

Figure 4.7. Sample Bank Application – User log in with Operator role 62

Figure 4.8. Sample Bank Application – User log in with Administrator role 63

Figure 4.9. Main User Account Sequence Diagram after Applying PIP 64

Figure 4.10. Sequence Diagram of Patient Search before Applying PIP 66

Figure 4.11. Load de-identification service DLL for desktop applications 67

Figure 4.12. Inject mocking object and invoke IOC 69

Figure 4.13. Resolve mocking object at runtime to get patient information 70

Figure 4.14. Apply Deidentification aspect on *PatientRecords* 71

Figure 4.15. De-identification implementation in *DeidentificationAspect* class 71

Figure 4.16. Patient Search Screen of Hospital Application 72

Figure 4.17. Patient Search Sequence Diagram after Applying PIP................................ 73

Figure 5.1. Theory of Reasoned Action (Fishbein & Ajzen, 1975)................................ 75

Figure 5.2. Survey Design 79

Figure 5.3. Participants understanding of the PIP pattern 82

Figure 5.4. Correlation between Factors and Variables after Varimax 88

Figure 5.5. Participants Groups' Perceived Usefulness 89

Figure 5.6. Participants Groups' Perceived Ease of Use.. 90

Figure 5.7. Overall Stacked Bar Chart of the Perceived Usefulness 91

Figure 5.8. Overall Stacked Bar Chart of the Perceived Ease of Use.............................. 92

Figure 5.9. Participants' Intention to Use the PIP Pattern.. 93

Abstract

Sensitive data may be leaked in many ways, and misuse of personal data from information systems is very common. It is challenging to implement privacy services in existing applications without affecting other modules. We propose the concept of a master privacy injection pattern (PIP) for software engineers to use to automate dynamically “injecting” existing privacy patterns in existing or new software without modifying its code, or in some cases modifying the code to a very small extent. We illustrate our new PIP and the simplicity of its implementation with the use cases that inject well-known de-identification patterns in a banking application and a hospital management system. Early evaluation results for PIP from a small survey of practising software engineering professionals are encouraging. The majority of respondents believe that the PIP is beneficial, easy to implement, and 85% of the participants stated their intention to use the pattern.

List of Abbreviations Used

SDLC	Software Development Life Cycle
PMRM	Privacy Management Reference Model and Methodology
PAWS	Privacy Architecture for Web Services
OASIS	Organization for the Advancement of Structured Information Standards
UML	Unified Modeling Language
FIPP	Fair Information Practice Principles
PIPEDA	the Personal Information Protection and Electronic Documents Act
FTC	Federal Trade Commission
DAL	Data Access Layer
BAL	Business Access Layer
DI	Dependency Injection
AOP	Aspect-Oriented Programming
OOP	Object-Oriented Programming
TDD	Test Driven Development
IoC	Inversion of Control
DLL	Dynamic Link Library
IDE	Integrated Development Environment
REST	REpresentational State Transfer
PII	Personally Identifiable Information

Acknowledgement

With a deep sense of gratitude, I wish to express my sincere heartfelt acknowledgements to my supervisor, Dr. Dawn Jutla, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the problem. Her profound insight into privacy, excellent research experience, perpetual energy and enthusiasm in research had motivated me a lot. In addition, she was always accessible and willing to help her students with their research. As a result, research life became smooth and rewarding for me.

Moreover, I would like to thank Dr. Peter Bodorik to timely help me with all sorts of documentation and writing. I am also thankful to the entire faculty and staff members of Department of Computer Science of their direct and indirect unconditional help and cooperation that made my stay at Dalhousie University memorable.

Moreover, I would also like to thank my parents and my husband for their prayers and support and to all my friends for always encouraging me and believing in me.

Chapter 1 : Introduction

According to Alexander, Ishikawa, & Silverstein (1977), 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 in software engineering represent sets of privacy requirements, and their relationships with system architecture and implementation, into repeatable design groupings that may be applied across software applications (Romanosky, Acquisti, Hong, Cranor, & Friedman, 2006; Kalloniatis, Kavakli, & Gritzalis, 2007; Porekar, Jerman-Blazic, & Klobucar, 2008; Bier & Krempel, 2012; Jeroen van Rest, Daniel Boonstra, Maarten Everts, Martin van Rijn, & Ron van Paassen, 2014). Theoretically and in practice software engineers' productivity improves with the recognition and use of repeatable patterns.

Numerous privacy patterns exist. For example, Kalloniatis et al., (2007) identify authorization, authentication, data protection, anonymization and pseudonymization, unobservability, and unlinkability privacy process patterns. Porekar et al., 2008 classify organizational privacy patterns as: "Obtaining explicit consent", "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 (2013) discuss a privacy policy as a pattern and reference Hoepman's work (Hoepman, 2014) on privacy strategies and categorization of privacy patterns. Others too (e.g. (Hafiz, 2006), (Romanosky et al., 2006)) discuss collections of privacy patterns. Romanosky et al., 2006 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 OASIS' Security Assertion Markup Language (SAML), the XML Access Control Markup Language (XACML), the Enterprise Privacy Authorization Language (EPAL), Privacy by Design Documentation for Software

Engineers (PbD-SE), and the Privacy Management Reference Model and Methodology (PMRM) specification of its atomic privacy services (Ali, Jutla, & Bodorik, 2015).

Once a privacy pattern is identified as per the above approaches, the pattern or its service's implementation still has to be "injected" into existing or new software. Incorporating privacy requirements during a system's design and implementation can protect sensitive data in information systems. 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.

1.1 Research Problem

As businesses are increasingly adopting computer based solutions, challenges such as handling of private data, adoption and implementation of privacy industry standards, and how to allow privacy auditors and officers to verify the promised privacy implementation into business solutions arise.

Currently, software solution providers are implementing privacy on an adhoc basis and do not provide the clear insight into the organization's compliance to the promised privacy policies. The way software has been written does not focus more on securing the customers' private data, but, rather privacy services are adopted on an as-needed basis. This model makes it complicated and less transparent for software developers to maintain the software over years and to keep it up-to-date with the new enhancements in the privacy standards. This also becomes nightmare for the privacy officers to audit the system.

Software engineers do not have a software engineering pattern that allows them to easily integrate other privacy patterns in legacy and/or new software systems. This thesis' research problem and key technical challenge is how to inject a privacy pattern and its accompanying service(s) automatically in new and existing systems.

1.2 Research Objectives

The objectives of this thesis are:

1. Creation of a master pattern to automatically inject other privacy patterns in existing applications without modifying existing applications or modifying it to a small extent, if required.
2. Demonstrate the applicability and use of the injection pattern via its application in simple and complex use cases within the context of de-identification.
3. Study the usefulness and ease of use of the proposed pattern by conducting a survey with software engineers using the Technology Acceptance Model (TAM) instrument.

1.3 Outline

Chapter 2 provides a literature review. In this chapter, we will discuss related work on privacy policies and engineering, aspect oriented programming, mocking, dependency injection, privacy using data mining, and connections between privacy engineering and data mining. Chapter 3 explains the proposed privacy injection pattern, discusses its components and describes its architecture. Chapter 4 then describes an implementation of our proposed pattern in a banking use case and a health care use case using Microsoft .NET platform while Chapter 5 describes the survey instrument and methodology that is used to evaluate the pattern and discusses the result of the survey conducted by the software engineers. Conclusions and future work are covered in Chapter 6.

Chapter 2 : Background and Related Work

Rapid growth in Internet technologies is encouraging business organizations to focus more on customers and thus helping them to attract and target more customers (Rodríguez, Piattini, & Fernández-Medina, 2006). The customers are sharing their personal information with organizations to acquire organization’s services. Protection of personal data is becoming essential for business to improve their customers’ acceptance and satisfaction (Berghe & Schunter, 2006). This makes the protection of customers’ personal data a leading concern for organizations that need to comply with customer needs, and privacy practices and regulations (Ghazinour & Barker, 2009). Organizations are publishing privacy policies and statements that promise sound ways of handling customers’ personal data. However, having privacy policy does not guarantee that these organizations have privacy technology to enforce the promises within organization (Karjoth & Schunter, 2002). Researchers have provided various tools, technologies, frameworks and methodologies to incorporate privacy in the system for the purpose of protecting customers’ personal data. In section 2.1, we discuss about the overview of the privacy. Section 2.2 briefs about the existing privacy patterns and services. We also discuss about the research carried out on the individual components that we are using in our proposed privacy injection pattern.

2.1 Privacy Overview

2.1.1 Privacy

Privacy in technology is not something new. In 1980, Louis Brandeis and Samuel Warren formulated privacy in their paper as the right “to be let alone” (Warren & Brandeis, 1890). But, the concept of privacy develops with the evolution in technologies. Finn, Wright, and Friedewald (2013) categorized privacy into seven types which are listed in Table 2.1.

Table 2.1 Seven Types of Privacy [adopted from (Finn et al., 2013)]

Privacy Type	**Description**
Privacy of the person	Privacy of the person is also referred as “bodily privacy”. It is the privacy of the body functions and body characteristics, genetic codes and biometric information etc.
Privacy of behavior and action	The ability to control and monitor oneself’s action performed either in public or in private space is the privacy of behavior and action. This include privacy of personal matters such as religious practices, political preferences, and other activities etc.
Privacy of personal communication	Privacy of personal communication is to restrict the interception of communications. Communications mode can be a telephone calls, face-to-face conversations, exchange of emails, or text messages etc. Examples of intercepting communication includes recording conversation through hidden camera or microphones, eavesdropping, use of computer viruses and bugs and so on.
Privacy of data and image	This type of privacy is commonly referred to as “information privacy”. Information privacy deals with the concept of collecting and sharing of individual’s information such as social security number, credit card number, bank account numbers, email address etc. Individual’s information also includes images and pictures. We can further break down information privacy into many other privacy groups such as financial privacy, internet privacy, etc. Failure to protect the privacy of individual’s data and image can make the user’s identity

	vulnerable or even put the user at risk.
Privacy of thoughts and feelings	People have a right to share their thoughts and feelings, and they should have the right to share whatever they feel as it can help societies in many ways. Protection of such thoughts and feelings is important because it creates balance of power between the state and the individual (Goold, 2009).
Privacy of location and space	Privacy of location and space is to avoid the disclosure of individual's current or past locations to third parties. It also deals with the issue of the automatic access to user's location by services such as navigation systems (Beresford & Stajano, 2003). For services that require the user's location as an input parameter, such as finding out all restaurants that are close to the user's location, privacy of the location and space should be carefully defined to satisfy user's desire of keeping their location secret while providing the service (Beresford & Stajano, 2003).
Privacy of association	It is the right of the people to be a part of any association they want, without any monitoring and controlling. This association can be political, religious, societal, etc. Privacy of association is important as it encourages groups to become vocal which otherwise cannot be achieved because of political and economic pressures.

2.1.2 Privacy Policy

A privacy policy can be defined as a statement or a legal document that discloses the ways an organization collects, stores, manages and uses the user's private information (Karjoth & Schunter, 2002). It also describes what information will be kept secret by an organization and what data will be shared with business partners or third parties. Sometimes privacy policies are so abstract that it is difficult to determine who is authorized to access what data and under what conditions. This raises the need of a formal semantic language to express the privacy policies in an unambiguous way that can easily be understood by all the stakeholders. Many countries have well defined privacy laws and standards to protect the privacy of an individual, private sector, government operations, or enterprises. Individual organizations and enterprises also have regulations and policies to protect the privacy of their customers and users.

2.1.3 Privacy Standards and Laws

According to Pew Research Center report of 2013 (Rainie, Kiesler, Kang, & Madden, 2013), 68% of the internet users believe that current laws are not providing reasonable protection of their privacy over the Internet and around 50% of the Internet users are worried about their personal information which is available online.

Federal Trade Commission (FTC) is an agency of the United States government since 1914. When the commission was first created, its purpose was to "prevent the unfair methods of competition in commerce" (Federal Trade Commission, n.d.). More functions were attached to the agency over the years. One of the goals of the FTC is to protect their customer from fraud, deception and unfair business practices (Federal Trade Commission, n.d.). FTC releases their reports regularly to guide organizations about Fair Information Practice Principles (FIPPs), and to motivate organization to implement FIPPs core principles.

In Canada, concerns about privacy protection started in late 60s (Holmes, 2008). In 1977, Canada's first federal privacy protection act, *Canadian Human Rights Act,* came into force. But, later it was realized that this act is not the best fit for privacy, so in 1983, *Privacy Act* was enacted with the *Access to Information Act.* The Act is also called an "information handler's code of ethics". The *Privacy Act* is applicable to federal government agencies and it specifies that personal information will be collected from the individual directly and only if it is related to the operational activity of the institution.

Furthermore, an individual should be informed about the purpose of information collection.

Another privacy law in Canada, *The Personal Information Protection and Electronic Documents Act (PIPEDA),* was introduced in late 1990s. PIPEDA is applicable to organizations in private sector for commercial activities and to the employees of federally regulated organizations. The Act guides organizations in maintaining customers' online privacy.

2.2 Privacy Research

Many researchers have carried out research on privacy issues. This section will shed light on some of the related research done in the field of privacy technologies.

2.2.1 Privacy Engineering:

The growth in collection, use and disclosure of personally identifiable information (PII) in the last decade has motivated increased need for protection of private information in public and private organizations. In response, Fair Information Practices Principles (FIPPs) and other international privacy standards emerged. The US Federal Trade Commission (FTC) proposed Fair Information Practices Principles (FIPPs) – guidelines that focus on minimizing personal data collection, communicate to the user about collection of data, and effectively maintaining the collected data (Spiekermann & Cranor, 2009; Pitofsky, Anthony, Thompson, Swindle, & Leary, 2000). These principles focus on fair information practices in all information systems: electronic, manual, and hybrid. Organizations draft their privacy regulations based on FIPPs.

Unfortunately, it has always been difficult to bridge the gap between legal language and computer language. Most of the time privacy breaches are not due to malicious intents, but due to unintended misuse of technologies or use of technologies that do not have proper security and privacy controls. These breaches may not only have direct financial impact, but also have negative impact on customer trust and loyalty. In order to address these issues to incorporate privacy in the processes, tools, and technologies, privacy engineering has become an emerging discipline. Kenny & Borking (2002) defined privacy engineering "*as a systematic effort to embed privacy relevant legal primitives into technical and governance design*".

We now discuss briefly the Privacy Management Reference Model and Methodology (PMRM), which is one the models and methodologies proposed by researchers to incorporate privacy in software. We base this discussion on (Sabo, Willett, Brown, & Jutla, 2012, 2013). PMRM is based on FIPPs and it helps organizations to navigate complexities in managing customer information in today's networked environment. It also helps organizations to improve privacy management and compliance in day-to-day business where customer information is protected by laws and regulations, and where the privacy enhancing technologies are not sufficient. The PMRM is used to analyze and understand complex and composite use-cases, to derive and implement appropriate privacy functionality, and to attain system-wide privacy compliance. This model also helps in the selection of appropriate privacy controls in order to ensure that processes are in line with organizations privacy policies. This model's service functionality is unaffected by the barriers such as multiple jurisdictions, regulations, business laws and practices, and often conflicting laws etc. The PMRM provides a standards-based model and is neither a static model nor a fixed set of defined rules as software engineers; architects and developers have flexibility in implementing privacy and security policies defined in PMRM. It serves as an analytical tool and helps to assess the completeness of proposed privacy in the actual business processes and it also helps in the analysis and design of the system functionality that are required to implement a set of privacy requirements. This model also provides stakeholders, such as software architects, developers, and policy makers, a tool to embed privacy management and compliance in software systems.

Different models are being used by organizations to implement privacy components in software systems and they were categorized into: Privacy by Architecture, Privacy by

Policy, and Privacy by design (Cavoukian, Shapiro, & Cronk, 2014). In Privacy by Architecture, customers become powerful or in control because data processing occurs at customers' machines instead of organization's infrastructure. Thus, it eliminates the need to transfer and store the data in an organization's database. This significantly reduces the secondary usage of customers' data. In the extreme case, if an organization decides to refrain from collecting personal information or bases its business on unidentifiable users, it is not mandatory to provide a customer with notifications and choices. There are systems based on this approach, such as the collaborative filtering system proposed in (Cavoukian, 2013), in which individuals' private data is stored on their own systems and an '*aggregate*' is computed on their data, which can later be shared with an organization and third parties. Privacy by Policy approach is completely opposite of privacy-by-architecture. FIPPs come into play when an organization decides to implement FIPPs principles by making customers feel comfortable about their private information, providing adequate security mechanism, and giving customers enough degrees of control over their information. This approach is widely practiced in software industry by those who collect individual's personal information, which is most for profit organization. Privacy by Design (PbD) (Cavoukian, 2013) states that the software development team should incorporate privacy in the early software development phase as an integral part of the software solution. PbD promotes enhanced accountability and user trust (Cavoukian et al., 2014). Privacy by Design Foundational Principle builds upon universal FIPPs in a way that updates and adapts them to modern information management needs and requirements (Gutwirth, Leenes, & de Hert, 2015).

Different research has been conducted for feature and role-based privacy implementation. Kaindl (2000) proposed a scenario-based design model that combines scenarios with functions and goals. In this model, purpose of data collection acts as an intermediary between goals and functions. System functions have purposes that are matched with the goals of the users, while the functions are performed by users with goals. This model lacks the concept of permission that links the purpose with the user which provides data protection. This issue is addressed in (He & Antón, 2003) wherein, authors present a model for privacy engineering involving roles and permissions as well. But this paper has not considered organization structure and different obligations that need to be performed by the users, who are collecting the data, in privacy context. Another framework was proposed by Tumer, Dogac, and Toroslu (2003) for the specification of the privacy policy and data subject preferences. According to the framework, an organization will specify mandatory and optional information required from the data subjects and data subjects can specify the type of access (free, limited, not given) they want to assign to each information. At the time of performing some functionality, system will verify enterprise policies with the data subjects' preferences and act accordingly.

Research has also been performed on the representation of privacy policies. A privacy policy refers to the promise made by the organization to the data subjects about how their personal information will be used and to whom it will be disclosed. Platform for Privacy Preferences (P3P) (Cranor, Langheinrich, Marchiori, & Reagle, 2002; Cranor L. F., 2002) is a W3C standard that allows different websites to state their privacy practices in a well-defined format that can be easily interpreted by automated user agents. P3P enable browsers can fetch the policy automatically and can compare it with the user's set of privacy preferences (Karjoth & Schunter, 2002). This model has developed standards for the user agents in order to automatically examine the privacy policy of e-business. But, P3P does not provide the procedure to check an access request against the stated privacy policy (Karjoth & Schunter, 2002). Besides, P3P does not prevent the specification of deceptive and unfair practices (Guarda & Zannone, 2009). The other XML-based languages used for privacy policies of organizations include EPAL and DPAL. Enterprise Privacy Authorization Language (EPAL), developed by IBM, is a formal language to write enterprise privacy policies (Ashley, Hada, Karjoth, Powers, & Schunter, 2003). Privacy policies expressed in EPAL are used to enforce the organization privacy policies through the enforcement engine (Md. Moniruzzaman, Ferdous, & Hossain, 2010). EPAL's main focus is on the core privacy authorization while keeping data models and user authentication on the abstract level away from deployment details (Ashley et al., 2003). EPAL policy is similar to access control rules or permissions. EPAL follows sequential semantics that is the sequence of the statements determines the order in which query will be answered. Each EPAL statement contains a condition that applies only to queries satisfying this condition. The authorization system examines the policy statements in order. The system stops when it reaches an applicable statement containing an “allow” or “deny” ruling. Hence it terminates evaluation mid-policy. The combination of two compatible EPAL policies cannot be expressed in EPAL. DPAL is Declarative Privacy Authorization Language that does not terminate evaluation mid-policy. The authorization system collects requirements from all applicable statements and enforces all their statements. In DPAL, concatenating two policies produces a policy that enforces each statement from each policy (Barth, Mitchell, & Rosenstein, 2004). But P3P and other languages does not deal with privacy policy compliance, i.e., it does not deal with checking whether an organization’s business processes comply with the organization stated privacy policies (Bodorik & Jutla, 2008; Barthet al., 2004).

The correctness of privacy-aware systems requires the compliance among privacy artifacts that includes organizational goal, privacy policy, customer preference, and data protection policy. Several researchers have proposed different ways to represent privacy policies using UML for requirement engineering. Jürjens (2005) proposed Role-Based Access Control (RBAC) for the secure system and he used stereotype {rbac} in UML to specify RBAC in the systems. Basin, Doser, and Lodderstedt (2006) used UML based modeling language to model access control policies. Jutla, Bodorik, and Ali (2013) proposed a way to represents privacy requirements using UML use case diagram. The paper is based on Privacy by Design principle and implemented privacy services using Microsoft Visio ribbon.

Privacy is still a growing field and much needs to be done. With the increase in the awareness and need of privacy, organizations are focusing on implementing privacy services and patterns in their existing or new systems. There are various privacy patterns identified by the researchers to implement in the systems to protect customers’ personal information. In the following section, we discuss about some of the privacy patterns identified by the researchers. We then discuss about the technologies that compose our master privacy injection pattern and help in injecting privacy patterns into the existing applications. We briefly explain the related work which reveals fragmentation in using the software engineering abstractions, that individual abstractions are used but not in combination, to address privacy, and an absence of software injection patterns for privacy.

2.2.2 Patterns on Privacy

Privacy patterns 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 (Romanosky et al., 2006; Kalloniatis et al., 2007; Porekar et al., 2008; Bier & Krempel, 2012; Jeroen van Rest et al., 2014). Theoretically, and in practice software engineers' productivity improve with the recognition and use of repeatable patterns.

There are various privacy patterns identified by the researchers to implement in the systems to protect the personal information of the customer. For example, Kalloniatis et al., (2007) identify authorization, authentication, data protection, anonymization and pseudonymization, unobservability, and unlinkability privacy process patterns. Porekar et al., (2008) 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*". The OASIS Privacy Management Reference Model and Methodology (PMRM) (Brown, Janssen, Jutla, Sabo, & Willett, 2013) emerging privacy standard proposes eight privacy service patterns: Agreement, Validation, Certification, Security, Access, Enforcement, Interaction, and Usage. For the thesis, we propose a master privacy injection pattern (PIP) for software engineers to use to automate dynamically "*injecting*" existing privacy patterns in existing or new code without modifying the legacy code, or in some cases modifying the code to a very small extent. We use the de-identification pattern and its service mapping as example privacy pattern/service to inject in the existing application. In the next section, we discuss different techniques of de-identification.

2.2.3 De-identification through Data Transformation Techniques

With the advancement in hardware technology, capacity to store the personal data also increases. This has led to concerns that the personal data can be misused for a variety of purposes. In response, numbers of de-identification techniques for privacy preserving have been discussed in number of papers. 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 (Cavoukian & Khaled El Emam, 2014; Shapiro, 2011; Narayanan & Shmatikov, 2008).

Character Masking, Randomization (Agrawal & Srikant, 2000), *k*-anonymity (Samarati & Sweeney, 1998; Agrawal & Srikant, 2000), *l*-diversity (Machanavajjhala, Gehrke, Kifer, & Venkitasubramaniam, 2007), *t*-closeness (Li, Li, & Venkatasubramanian, 2007), and many other data transformation techniques associated with the privacy are discussed by the researchers. These data transformation techniques reduce the granularity of representation in order to reduce the privacy which in turn results in the information loss. This is the natural trade-off between information loss and privacy. For the thesis, we use character masking and k-anonymity techniques to de-identify the personal data. We discuss these and other techniques in the next section.

2.2.3.1 Character Masking Method

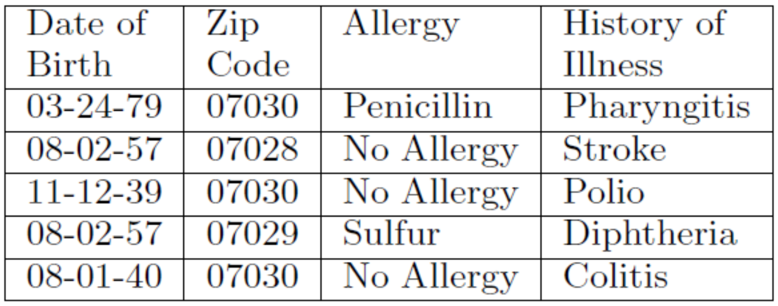
De-identification of the sensitive information can be achieved by Character masking. Character masking is a technique of replacing all or partial characters in the data with the special characters such as *, X, & etc. The length of the masked data remains the same after masking. For example to mask credit card number 6565 1111 5050 7896 can be partially masked with the character X to give the value XXXX XXXX XXXX 7896 (Raghunathan, 2013).

2.2.3.2 The Randomization Method

The randomization technique is a data transformation technique in which noise is added to the data to mask the attribute values so that it become difficult to recover individual record (Agrawal & Srikant, 2000). If the variance of the added noise is large enough, then it becomes difficult to guess original data from transformed data. Randomization method is simple to implement as it does not require knowledge of other records in the data in comparison with k-anonymity where it requires the knowledge of other records in the data. Randomization has some weaknesses also. It does not consider local density of a record that make outlier susceptible to adversarial attacks as compare to records in denser region (Aggarwal & Yu, 2008).

2.2.3.3 The k-anonymity Model

The *k*-anonymity is a de-identification method and it helps to preserve the sensitive information. The motivating factor for the *k*-anonymity is how to avoid from uniquely identifying record when used in conjunction with public records. Many applications remove key identifiers (name, social security number, medical record number etc.) from the data to avoid identifying the records. But we can identify a record using some attributes in the data such as birthdate and zip-code in Figure 2.1 can be used to indirectly infer the identity of the individual. These attributes are called quasi-identifiers (Zhong, Yang, & Wright, 2005).



Date of Birth	Zip Code	Allergy	History of Illness
03-24-79	07030	Penicillin	Pharyngitis
08-02-57	07028	No Allergy	Stroke
11-12-39	07030	No Allergy	Polio
08-02-57	07029	Sulfur	Diphtheria
08-01-40	07030	No Allergy	Colitis

Figure 2.1. Health related data [adopted from Zhong et al. (2005)]

The idea behind *k*-anonymity is to reduce the granularity of the representation of the data in such a way that a given record cannot be distinguished from at least (k-1) other records (Aggarwal & Yu, 2008). This granularity is reduced using techniques such as generalization and suppression. In generalization, we replace the attribute value with a generalized value. For example, birthdate can be generalized to birthyear in order to avoid identifying an individual from the data. Attributes can be generalized by replacing their values with that of their (common) parent. For example to generalized ZIP codes 84117, 84118, 84120, we can replace it with the generic ZIP 841**. This technique "blurs" the data to prevent identification of individual record but also continue to provide statistical utility (Aggarwal & Yu, 2008).

Suppression is the technique where attribute value is removed completely. For example, suppose we have a dataset consisting mostly of one zipcode except for few records that have 84120 zipcode. To prevent outlier tuples inferring any individual record, we can suppress outlier tuples. Using suppression method risk of identifying an individual record is minimized but its utility is also reduced.

In a k-anonymous table, each value of the quasi-identifiers should appear at least k times so that each entity or individual's record will be hidden inside the data with at least k peers. Figure 2.2 show 2-anonymous of the health data shown in Figure 2.1. As shown in the figure, we have performed generalization on ZipCode and suppression on Date of Birth to make it 2-anonymous. After 2-anonymization, each values of quasi-identifiers (Date of Birth, ZipCode) is appearing at least 2 times for example {*, 07030} is appearing 3 times and {08-02-57, 0702*} is appearing 2 times.

Date of Birth	Zip Code	Allergy	History of Illness
*	07030	Penicillin	Pharyngitis
08-02-57	0702*	No Allergy	Stroke
*	07030	No Allergy	Polio
08-02-57	0702*	Sulfur	Diphtheria
*	07030	No Allergy	Colitis

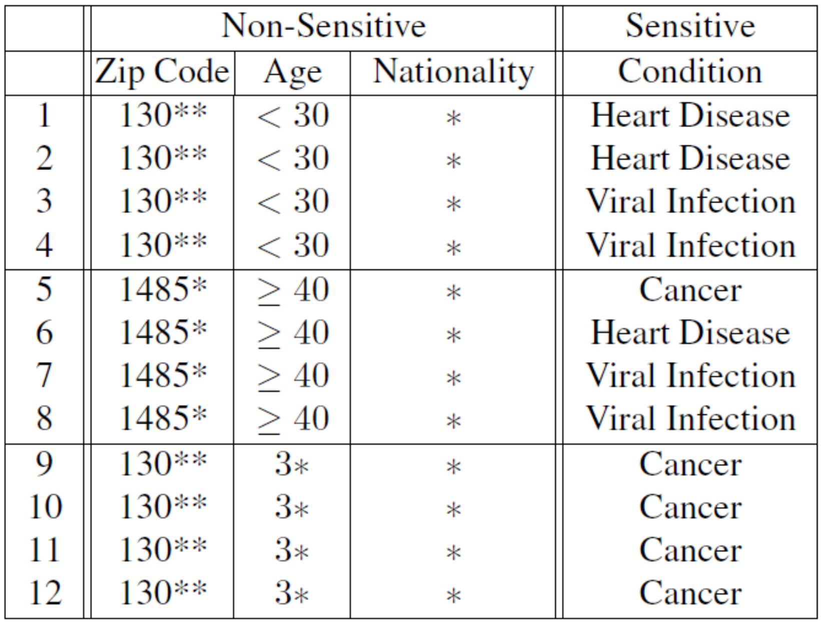
Figure 2.2. 2-Anonymous health data [adopted from Zhong et al. (2005)]

2.2.3.4 The l-diversity Model

The k-anonymity is susceptible to attacks when individual's background knowledge is available to the attacker. The two such kinds of attacks are: homogeneity attack and background knowledge attack. Suppose we have health related data as mentioned in Figure 2.3. As mentioned in the figure, medical condition is the sensitive information while zipcode, age and nationality are the non-sensitive information. We called medical condition as sensitive information because it values for any individual in the dataset should not be discover by any adversary. In the figure, zipcode, age, and nationality are quasi-identifiers and so its 4-anonymous table is shown in Figure 2.4.

	Non-Sensitive			Sensitive
	Zip Code	Age	Nationality	Condition
1	13053	28	Russian	Heart Disease
2	13068	29	American	Heart Disease
3	13068	21	Japanese	Viral Infection
4	13053	23	American	Viral Infection
5	14853	50	Indian	Cancer
6	14853	55	Russian	Heart Disease
7	14850	47	American	Viral Infection
8	14850	49	American	Viral Infection
9	13053	31	American	Cancer
10	13053	37	Indian	Cancer
11	13068	36	Japanese	Cancer
12	13068	35	American	Cancer

Figure 2.3. Sensitive and non-sensitive information in health data
[adopted from (Machanavajjhala et al., 2007)]



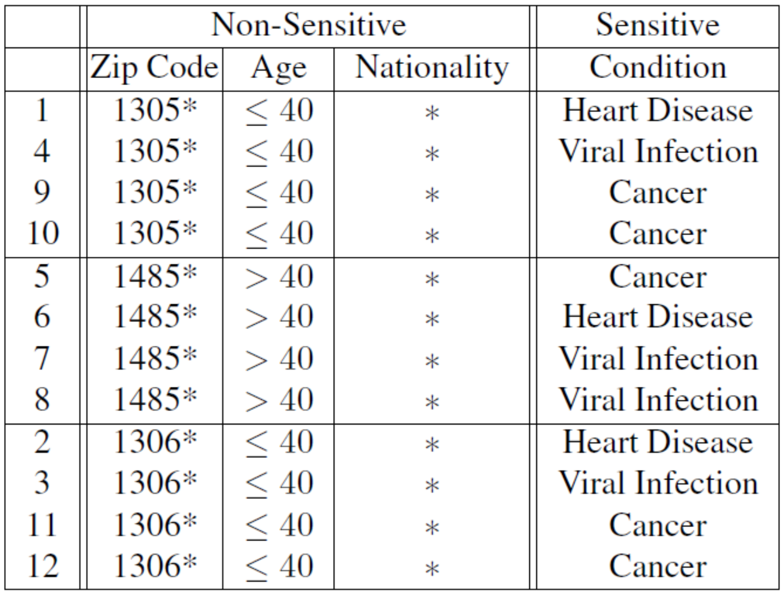
	Non-Sensitive			Sensitive
	Zip Code	Age	Nationality	Condition
1	130**	< 30	$*$	Heart Disease
2	130**	< 30	$*$	Heart Disease
3	130**	< 30	$*$	Viral Infection
4	130**	< 30	$*$	Viral Infection
5	1485*	≥ 40	$*$	Cancer
6	1485*	≥ 40	$*$	Heart Disease
7	1485*	≥ 40	$*$	Viral Infection
8	1485*	≥ 40	$*$	Viral Infection
9	130**	$3*$	$*$	Cancer
10	130**	$3*$	$*$	Cancer
11	130**	$3*$	$*$	Cancer
12	130**	$3*$	$*$	Cancer

Figure 2.4. 4-anonymous health data [adopted from (Machanavajjhala et al., 2007)]

Suppose attacker found 4-anonymous health data as mentioned in Figure 2.4. Attacker also has some background information about his target for example; he knows that his attacker lives in the zipcode 13068 and his age will be in thirties i.e. 9th, 10th, 11th or 12th record. From the 4-anonymous data, attacker can easily identify that his target is suffering from cancer. This type of attack is called homogeneity attack where sensitive information within a group of k records is the same. Now suppose, attacker knows that his target is Japanese, his age is less than 30 and he lives in zipcode 13068 i.e. 1st, 2nd, 3rd or 4th record. But it is also known that Japanese has low rate of heart disease so attacker can easily predict that his target has viral infection. This type of attack is called background knowledge attack.

Machanavajjhala et al. (2007) proposed l-diversity which not only maintain k-anonymity but also maintain the diversity of the sensitive information within the group of size k. According to the author, suppose q*-block is a set of tuples whose non-sensitive values are generalize to q*. Each q*-block should have l >=2 different sensitive values such that *l* most frequent values (in the q* block) have roughly the same frequency i.e. each q*-block is well represented by *l* sensitive values.

Figure 2.5 shows 3 diverse data of the previous health data mentioned in Figure 2.3. Here each q*-block ({1,4,9,10}{5,6,7,8}{2,3,11,12}) have 3 different sensitive values and their frequencies in each q* block is (50%, 25%, 25%) i.e. each block is well represented by 3 sensitive values.



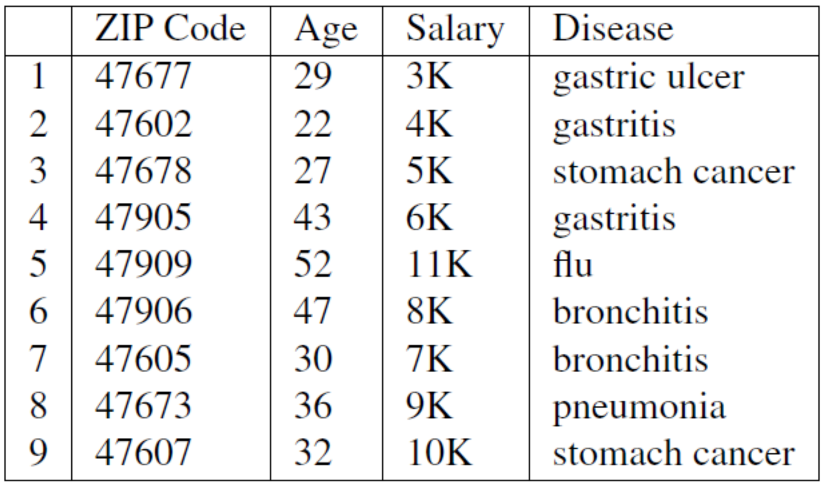
	Non-Sensitive			Sensitive
	Zip Code	Age	Nationality	Condition
1	1305*	≤ 40	$*$	Heart Disease
4	1305*	≤ 40	$*$	Viral Infection
9	1305*	≤ 40	$*$	Cancer
10	1305*	≤ 40	$*$	Cancer
5	1485*	> 40	$*$	Cancer
6	1485*	> 40	$*$	Heart Disease
7	1485*	> 40	$*$	Viral Infection
8	1485*	> 40	$*$	Viral Infection
2	1306*	≤ 40	$*$	Heart Disease
3	1306*	≤ 40	$*$	Viral Infection
11	1306*	≤ 40	$*$	Cancer
12	1306*	≤ 40	$*$	Cancer

Figure 2.5. 3-diverse table [adopted from (Machanavajjhala et al., 2007)]

2.2.3.5 The t-closeness Model

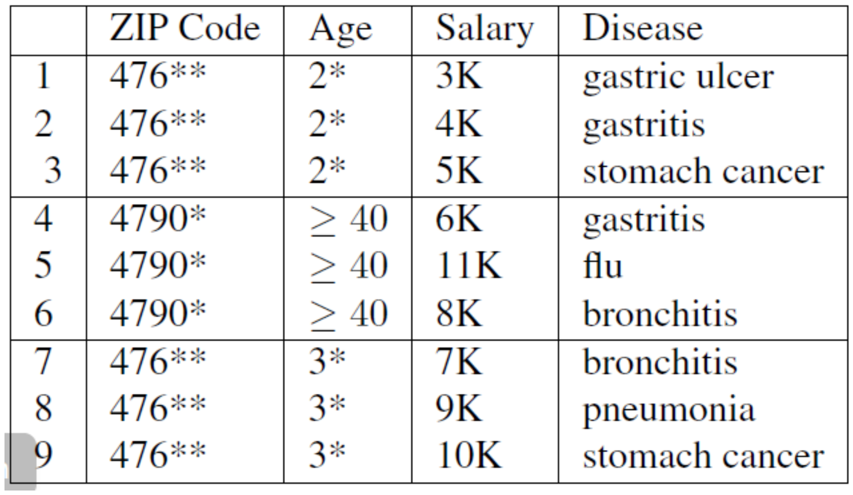
The l-diversity also has some limitations. It does not consider the distribution of the values of an attribute in the overall data which is not the case in real data; attribute values can be skewed (Aggarwal & Yu, 2008). This skewed data will make it difficult to correctly represent data in *l*-diverse form. Suppose we have a dataset consisting of 10000 records out of which 99% of them are negative while 1% is positive. If we have a q*-block (an equivalence class) with 1 negative record and 49 positive records. Even after l-diversity, each record in the equivalence class will be considered as 98% positive rather than overall 1% positive.

When equivalence class has distinct sensitive values but semantically similar sensitive values, in that case also attack can be possible. Consider salary and disease related data mentioned in Figure 2.7, whose original data is shown in Figure 2.6Figure 2.7. Suppose attacker knows that his target lives in 47677 and whose age is in twenties, he can easily predict that his target's salary is between 3,000 and 5,000 and he has some stomach related issue as gastric ulcer, gastritis, and stomach cancer are stomach related disease.



	ZIP Code	Age	Salary	Disease
1	47677	29	3K	gastric ulcer
2	47602	22	4K	gastritis
3	47678	27	5K	stomach cancer
4	47905	43	6K	gastritis
5	47909	52	11K	flu
6	47906	47	8K	bronchitis
7	47605	30	7K	bronchitis
8	47673	36	9K	pneumonia
9	47607	32	10K	stomach cancer

Figure 2.6. Salary/disease data [adopted from Li et al. (2007)]



	ZIP Code	Age	Salary	Disease
1	476**	2*	3K	gastric ulcer
2	476**	2*	4K	gastritis
3	476**	2*	5K	stomach cancer
4	4790*	≥ 40	6K	gastritis
5	4790*	≥ 40	11K	flu
6	4790*	≥ 40	8K	bronchitis
7	476**	3*	7K	bronchitis
8	476**	3*	9K	pneumonia
9	476**	3*	10K	stomach cancer

Figure 2.7. 3-diverse salary/disease data [adopted from Li et al. (2007)]

Li et al. (2007) propose t-closeness model. According to the model, distance between the distributions of the sensitive attributes within each equivalence class and the overall distribution of the data should not be more than a threshold t. The Earth Mover distance metric (Rubner, Tomasi, & Guibas, 2000) can be to calculate the distance between the two distributions. Furthermore, the t-closeness approach tends to be more effective than many other privacy-preserving data mining methods for the case of numeric attributes.

Once a privacy pattern is identified, the pattern or its service implementation still has to be "*injected*" into existing or new software systems. It is comparatively easy to incorporate privacy in the new application, but for existing application, it requires significant changes in the existing application to incorporate the privacy services. For this thesis, we propose the concept of a master privacy injection pattern (PIP) for software engineers to use to automate dynamically "injecting" existing privacy patterns in existing or new code without modifying the legacy code, or in some cases only modifying the code to a very small extent. PIP is composed of a novel tri-abstraction combination of aspect-oriented programming, dependency injection, and mocking. In the next section, we discuss these abstractions in detail.

2.2.4 Aspect Oriented Programming

The first abstraction in our proposed privacy injection pattern is Aspect Oriented Programming aka AOP. Before we discuss the privacy related work using Aspect Oriented Programming, we will understand aspect oriented programming in the next section.

2.2.4.1 Core Concerns and Cross-Cutting Concerns

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, which is also called 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 system's primary functionalities and purposes while cross-cutting concerns are those functionalities whose implementation is spread in different modules of the program. Figure 2.8 shows core concerns and cross-cutting concerns of Internet Banking System. As mentioned in the Figure 2.8, requirements related to new customer, account and customer management are core concerns while security and failure recovery requirements are cross-cutting requirements as they may influence the implementation of all of the other system requirements.

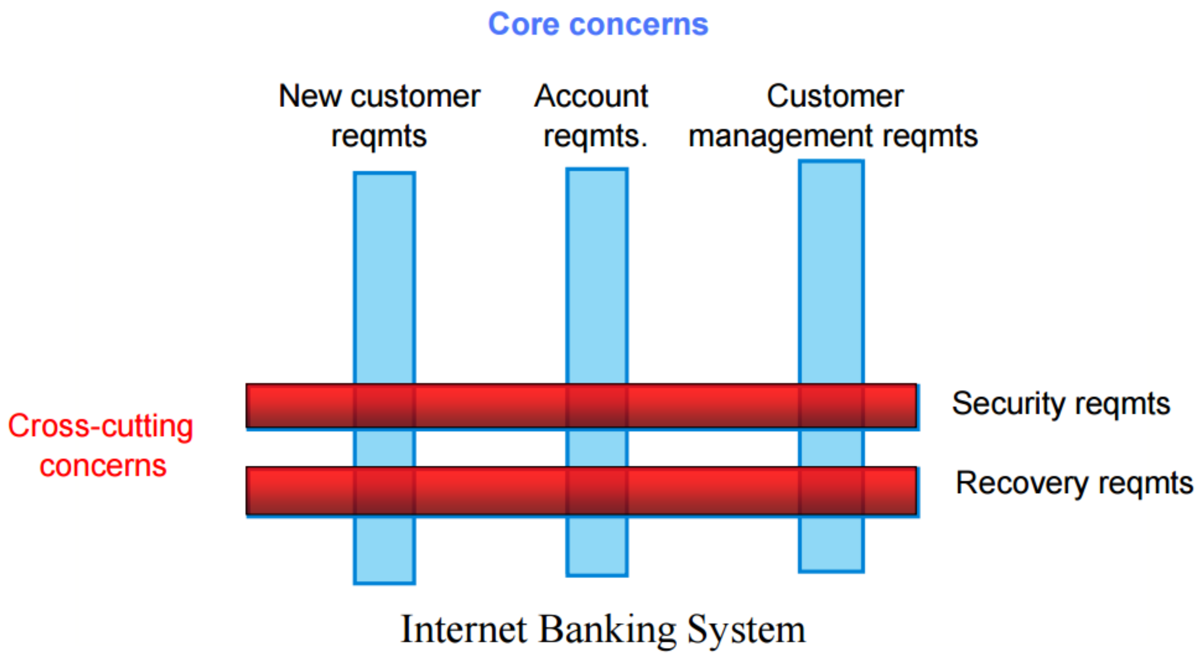


Figure 2.8 Core concerns and cross-cutting concerns in Internet Banking System [adopted from (Sommerville, 2011)]

Object Oriented Programming (OOP) is most commonly used to implement systems' core functionality. However, OOP is not sufficient for repeating cross-cutting solutions because it typically creates a strong coupling between core and cross-cutting concerns, such as logging and transaction management (Laddad., 2003). Using OOP, the software engineer needs to modify the core modules and repeat the same code in many modules to incorporate cross-cutting concerns. Cross-cutting concerns are not suitable using OOP because of program modification it result in many issues due to tangling and scattering. Tangling occurs when a module implements multiple requirements for example, in the case of Internet Banking System shown in Figure 2.8, new customer component is implementing two more secondary concerns, security and recovery requirements, other than its core concern of maintaining new customer information. Scattering occurs when more than one module implement a requirement for example, in the same Figure 2.9, all three components, new customer, account and customer management, are implementing recovery requirements (Sommerville, 2011). This is the reason many programming problems cannot properly be implemented using object-oriented programming or procedural programming and which require aspect-oriented programming to clearly implement the design decisions in the program.

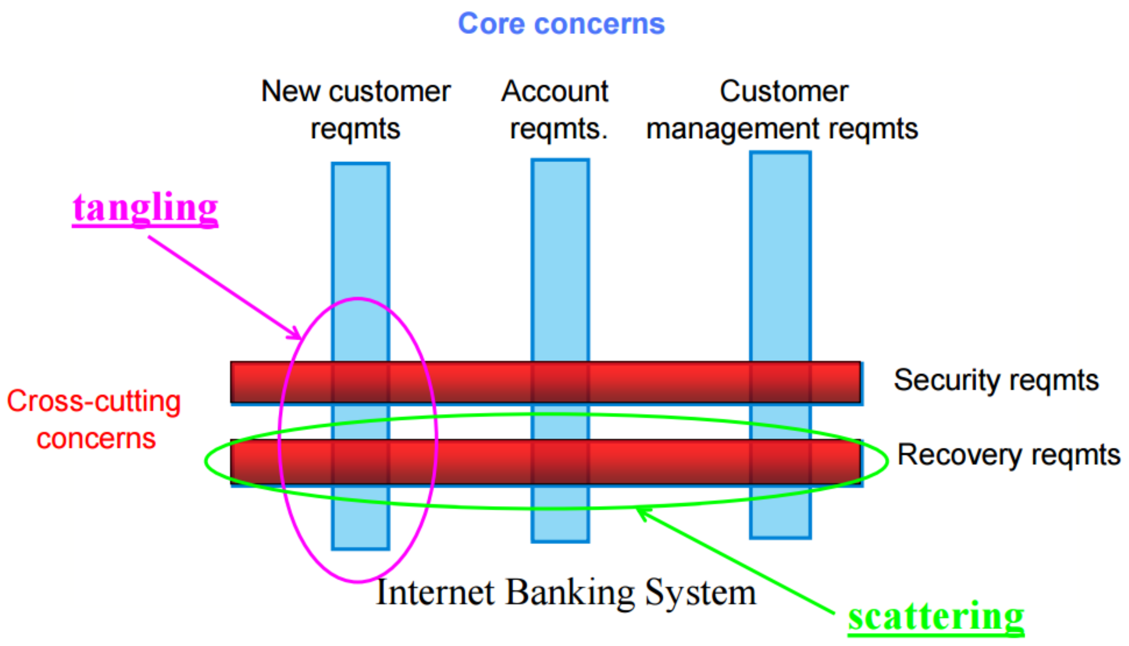


Figure 2.9. Tangling and Scattering for Internet Banking System (adopted from [(Sommerville, 2011)])

2.2.4.2 Aspect Oriented Programming Key Terminologies

Aspect-oriented programming (AOP) is a programming technique to separate crosscutting concerns in a new unit of modularization called **aspects**, instead of fusing them with core modules. Aspect Oriented Programming aka AOP is not a new concept, but it has been around for a number of years. AOP has recently started gaining more attention from the development community (Deiters, 2005). AOP was developed by Gregor Kiczales and colleagues at Xerox Company in 1997, now known as PARC (Palo Alto Research Center). The idea of Aspect Oriented Programming was proposed mainly to resolve the issue of cross-cutting concerns (Groves, 2013). Aspect consists of crosscutting code that needs to be executed called **advice**. The events for which crosscutting code will be included in the program is called **pointcut**. The events specified by pointcut are called **join points**. Crosscutting code is combined with the program using **weaving** process. 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 represent an aspect that can be used wherever logging is required for any type of transaction. They can be included before a method, after a method or when an attribute is accessed (Sommerville, 2011). Executable aspect-oriented program is created by combining a project implementation with aspects that handles cross-cutting concerns related to the project using aspect weaver. There are three different approaches to aspect weaving.

Table 2.2. Approaches of Aspect Weaving [adopted from (Sommerville, 2011)]

Approach	**Description**
Source code pre-processing	Weaver takes source code and combines it with aspects code to generate final source code which is then compile using language compiler.
Link time weaving	Aspect weaver included with language compiler that generates Java bytecode which is then be executed by Java interpreter.
Dynamic weaving	When joint points occur, corresponding advice will be called.

The goal of AOP is to keep concerns localized rather than scattered. If we want to alter an aspect in the program, we simply need to modify the aspect without modifying the core functionalities. This helps to avoid making any mistakes or introducing any errors in the program.

2.2.4.3 AOP and Privacy

AOP has been used to implement security by many researchers. Sharma, Batra, and Mukherjee (2014) proposed using AOP for the secure transfer of data over the internet. According to the authors, privacy of the information can be protected by encrypting/decrypting the data using hashing. This hashing will be performed by security agent implemented as an aspect. Secret key is generated at both ends using hash function. Integrity, confidentiality, authenticity, privacy and transmission can be achieved using the approach. Win, Joosen, and Piessens (2002) also used AOP for a cross cutting concern i.e. security. In this paper, two applications (a Personal Information Management (PIM) system and a server for file transfer) are discussed where AOP is applied for security. (Mourad, Laverdière, & Debbabi, 2008) and (Zhu & Zulkernine, 2009) also proposed using aspect-oriented programming for secure application.

Chen and Wang (2007) in their paper used aspect-oriented programming as a mechanism to implement privacy-aware access control. In this work, application-level access control implemented using AOP, is extended to enforce privacy policies on personal data with little impact on the structure of the application. Inter-type declarations or commonly called member introduction is a mechanism that allowed the programmer to modify class members/fields and relationship between classes. 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. Privacy policies are implemented by comparing the purpose of request and data subject's consent which he/she agreed on. As mentioned in Figure 2.10, action purpose manager is used to fetch purpose of request while for data subject's consent or preferences, preference aspect invoke preference factory to fetch privacy preferences and link it with the requested data. Lastly, access control aspect ensures that requestor is authorized user, have authority to perform requested action and finally filter user's PII according to privacy preference attached with the PII.

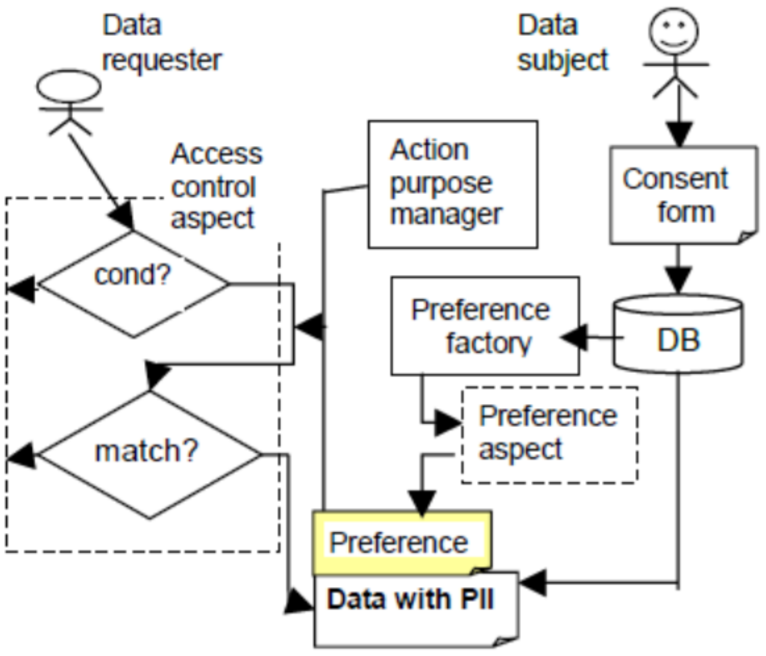


Figure 2.10. Privacy Control Access Control [adopted from (Chen & Wang, 2007)]

This concept can be used with different policy languages such as EPAL, but the discussed sample implementation contains hard coded logic blocks instead of a policy language. With the application of ITD, virtual multiple inheritance can be achieved. This is powerful, but also very fragile and bug prune because it somehow breaks the fundamental Java encapsulation rules. This mechanism used in this paper helps to enforce privacy mechanism but is applicable to the application or similar structure applications. The master privacy injection pattern is applicable to any applications as it is a pattern which helps to resolve similar type of problem.

The work of Berghe and Schunter (2006) proposes to use aspect technology for privacy enforcement in existing application. The paper introduced Privacy Injector that consists of two parts, a privacy metadata tracking and a privacy policy enforcement part. "Sticky policy paradigm" (Karjoth, Schunter, & Waidner, 2002) is implemented to protect the personal data. This paradigm suggests that privacy promise made to a data subject will remain with the data to enforce consented privacy policy later when the data is used. Privacy metadata tracking part of Privacy Injector is the practical implementation of the sticky policy paradigm and consists of three sub-components, privacy metadata assignment, metadata-preserving data operation and metadata persistence. Privacy metadata assignment component assigns privacy metadata to personal data that enters the system through web requests, emails etc. Metadata-preserving data operation preserves and updates privacy metadata when certain operations are performed on the data. In this paper, personal data stored as string is only covered for proof-of-concept but can be applied to any data type. Privacy metadata is assigned for string fragment instead of whole string and data operations are intercepted at the primitive data type level of the execution platform. Metadata persistence component preserves, restores, modifies and deletes privacy metadata when data is made persistent, retrieved, altered and deleted, respectively. For the implementation of this component, persistence service is separated from the application so that application will not directly perform any SQL query but rather depends on persistent service for all persistent data related operations. These components are application and enterprise-independent and can be used for any application. Privacy policy enforcement part enforces the sticky policies by verifying the usage of the data according to privacy policy.

Aspect technology is used to create connection between Privacy Injector and target application. Privacy Injector is based on the idea of context-sensitive string evaluation (CSSE) (Pietraszek & Berghe, 2005), a method that help detect and prevent injection attacks and is mainly focused on prevention of unwarranted disclosure and over-retention of personal data. Figure 2.11 depicts the life cycle of personal data provided by data subject DS to enterprise A. The data is stored in storage A and shared with enterprise B. Arrows represents the flow of the data with action and parameters required for that action. Changes in the parameter is represented using quotes for example, consent is initial consent given by data subject, consent' is the consent sent to enterprise B and consent'' is consent requested from data subject by enterprise B. Privacy Injector is available in the boxed area.

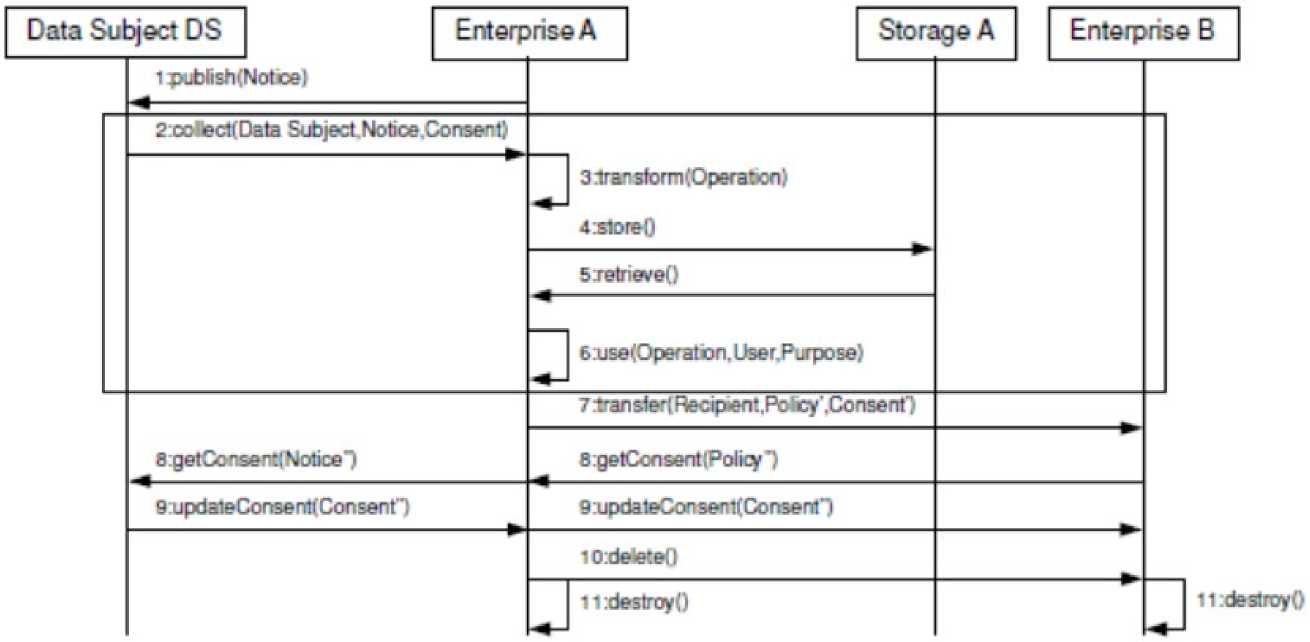


Figure 2.11. Life cycle and flow of personal data [adopted from (Berghe & Schunter, 2006)]

Berghe and Schunter (2006) approach associate privacy metadata with each data. All data manipulation operation should work according to the attached privacy metadata. This approach will impact the overall performance of the system as checking each action against metadata will be inefficient.

Scheffler, Schindler, & Schnor (2012) proposed the use of aspect-oriented programming and sticky policy approach to enforce privacy in location based services. In this paper, authors implemented data-owner defined privacy policies, where different data subject can have different privacy preferences that are enforced by the service. For the use case, theme-park location service is used. Usually theme park areas are widely dispersed. People visit theme parks usually in groups and due to dispersed area, can lose contacts with other group members. Theme-park location service will help to locate the group members in the park area and it can also help operator for advertisements of certain attractions to the visiting members. Both these features require user consent to use his/her location information. For the implementation of this service, either an electronic device can be provided to people before entering the park or an application can be installed in their mobile phone. With the help of the mobile phone or electronic devices, location signals are sent to server along with attached individual policy which will then be stored and used by the group members or operator. Additional information like nearby restaurant or events information can also be sent using this service. Model-view-controller paradigm is used for the theme park application. New groups, visited location can be analyzed using web interface. Business logic is implemented on service layer which uses Data Access Object (DAO) to access the data source. Overall architecture of the theme-park application is mentioned in Figure 2.12.

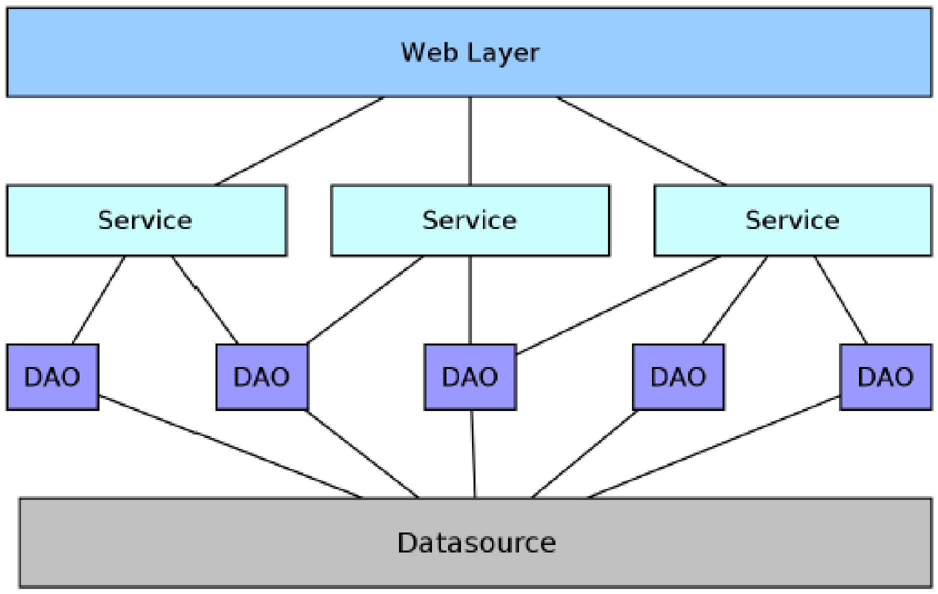


Figure 2.12 Architecture of theme-park location service [adopted from (Scheffler et al., 2012)]

Scheffler, Geiß, & Schnor (2008) presented the idea of enforcing privacy using reference monitor, based on Java Security Framework (Gong, Mueller, Prafullchandra, & Schemers, 2007). If the architecture is implemented using this reference monitor then two issues can occur. First, DAO classes need to be reloaded every time new customer enters the park and register for the location service as the application won't be able to update the policy dynamically. Second issue is that, there is no possible way to consider calling service when evaluating a policy. As depicted in Figure 2.13, two services, service 1 and service 2, tries to access location information of a visitor (object O3) to which policy P3 is attached using DAO2. P3 allows service 1 to access the object O3 but does not allow service 2 to access O3. We cannot implement the policy P3 without using AOP as both the services are using DAO2 to access O3.

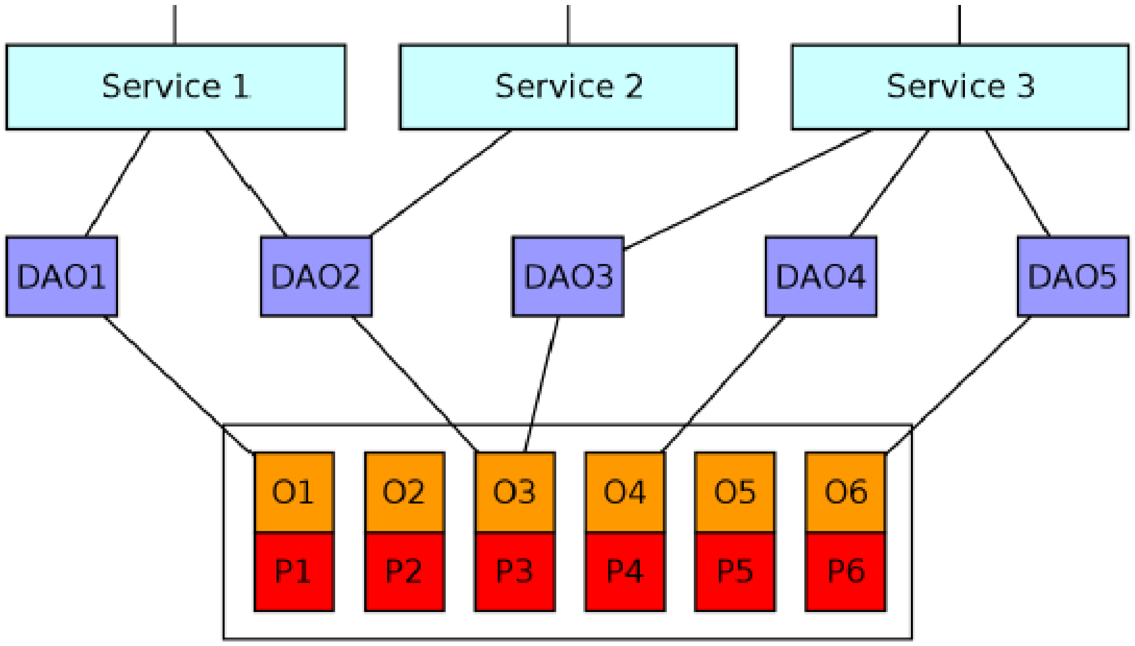


Figure 2.13 Access permissions to the object on the basis of identity of the calling service [adopted from Scheffler et al. (2012)]

Scheffler et al. (2012) extend this idea using AOP to build an AOP-based reference monitor. Figure 2.14 shows the updated reference model using AOP. It indicates that when an object tries to access a resource R that is attached to a sticky-policy P. Reference model first evaluates the sticky-policy P using task-specific advices (I) and then grant the access. Context object can be used by the reference monitor to get additional information for the evaluation of the policy.

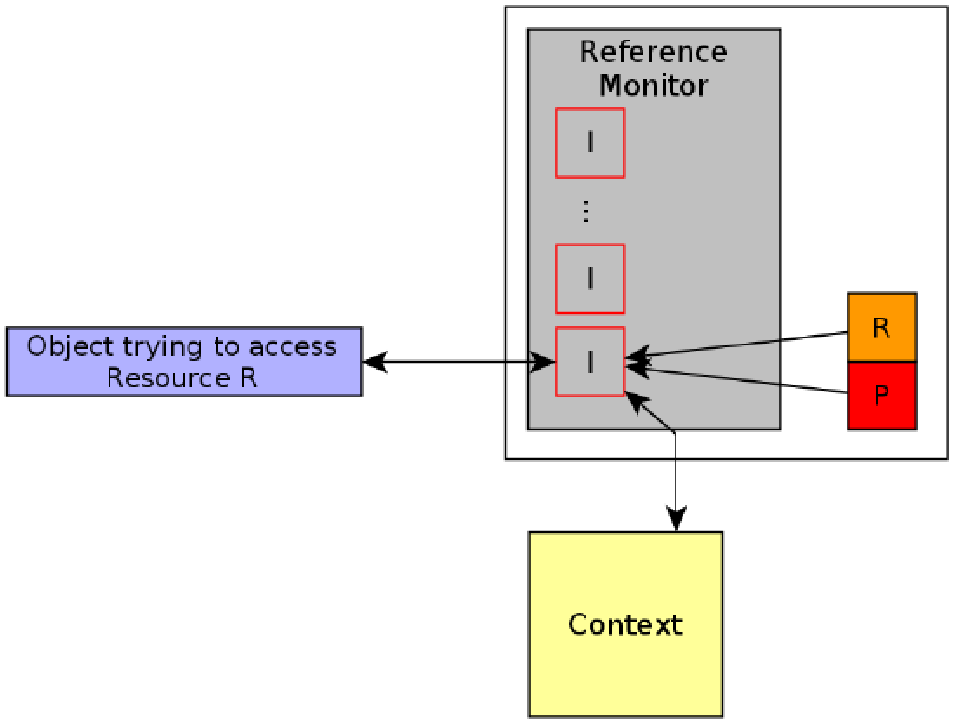


Figure 2.14. An AOP based reference monitor [adopted from Scheffler et al. (2012)]

2.2.5 Dependency Injection

Increase in open source APIs is motivating developers to use readily available APIs and wire them together with different components of a solution to form a cohesive architecture. But as the size of the application increases, so is the complexity and dependencies between different components, which ultimately make wiring of different components difficult. To mitigate this problem, we can use a design pattern called Dependency Injection (DI) which allows developers to inject dependency objects into a class, rather than relying on the class to create the dependent object itself.

Dependency Inversion Principle (DIP) was first proposed by Robert Martin in 1992 (Champatiray, 2014). According to him, "high-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions." According to this principle, a class should not depend on another class, it should depend on the abstraction i.e. interface, abstract etc. This principle can be implemented using Inversion of Control (IoC) design pattern. The IoC is a technique that assigns the responsibility of flow of control of an application to a container or a class (Johnson & Foote, 1988). The concept of dependency injection is based on the inversion of control (IoC) design pattern. Figure 2.15 depicts the relationship between DIP, IoC and DI. As shown in the diagram, there are other methods to implement IoC for example, factory, service locator etc. In this paper, we will focus on Dependency Injection to implement IoC.

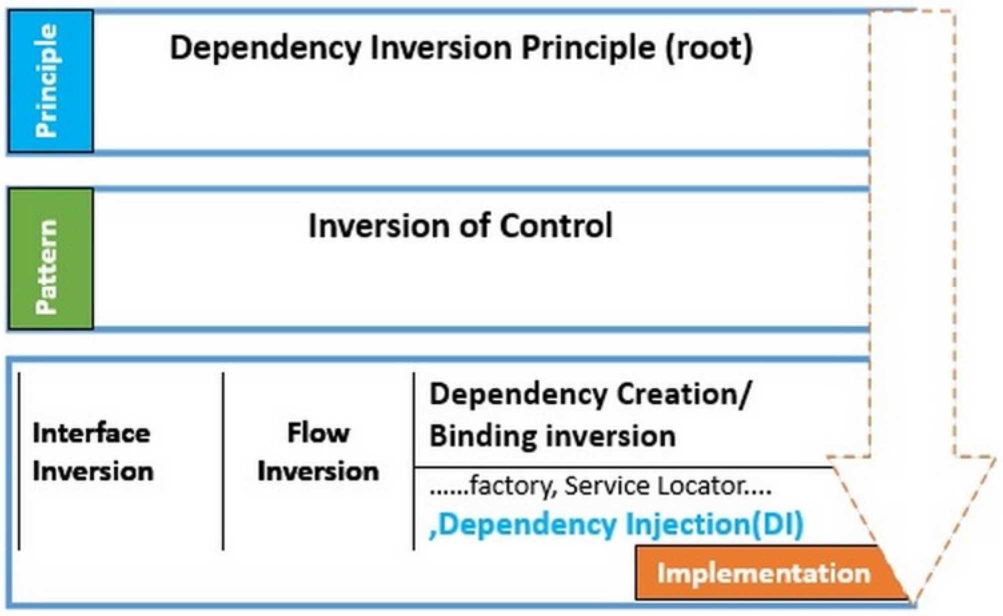
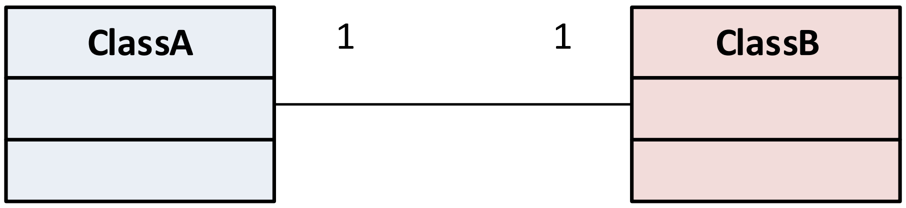

Principle
Dependency Inversion Principle (root)
Pattern
Inversion of Control
Interface Inversion
Flow Inversion
Dependency Creation/ Binding inversion
......factory, Service Locator....
,Dependency Injection(DI)
Implementation

Figure 2.15. Relationship between DIP, IoC and DI [adopted from (Haque, 2013)]

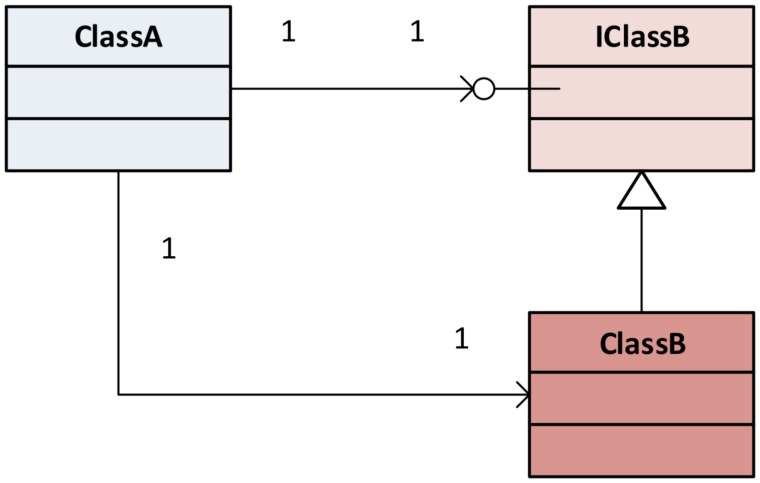
Instead of the traditional pull model where a class depends on another class that is responsible for the creation or locating the object it wants to consume, dependency injection takes a push approach. In this approach the responsibilities of initialization, assembly, and wiring of objects are not implemented by the system developers, but instead provided by the framework, thus inverting the flow of control.

Dependency injection is a design pattern that is useful to reduce the complexity of the system (Ježek, Holý, & Brada, 2012). Tightly coupled architecture is the one that has classes which are linked with a binary association. For example, when a class ClassA has an object of another class ClassB; it signifies that ClassA is dependent on ClassB as shown in Figure 2.16.



```
/// <summary>
/// Implementation of ClassA
/// </summary>
class ClassA
{
    /// <summary>
    /// private instance of ClassB
    /// </summary>
    private ClassB classB = new ClassB();

    /// <summary>
    /// Constructor to initialize the intance
    /// </summary>
    public ClassA() {  }
}

/// <summary>
/// Implementation of ClassB
/// </summary>
class ClassB
{
    /// <summary>
    /// Constructor to initialize the instance
    /// </summary>
    public ClassB() { }
}
```

Figure 2.16 Tightly coupled classes and code



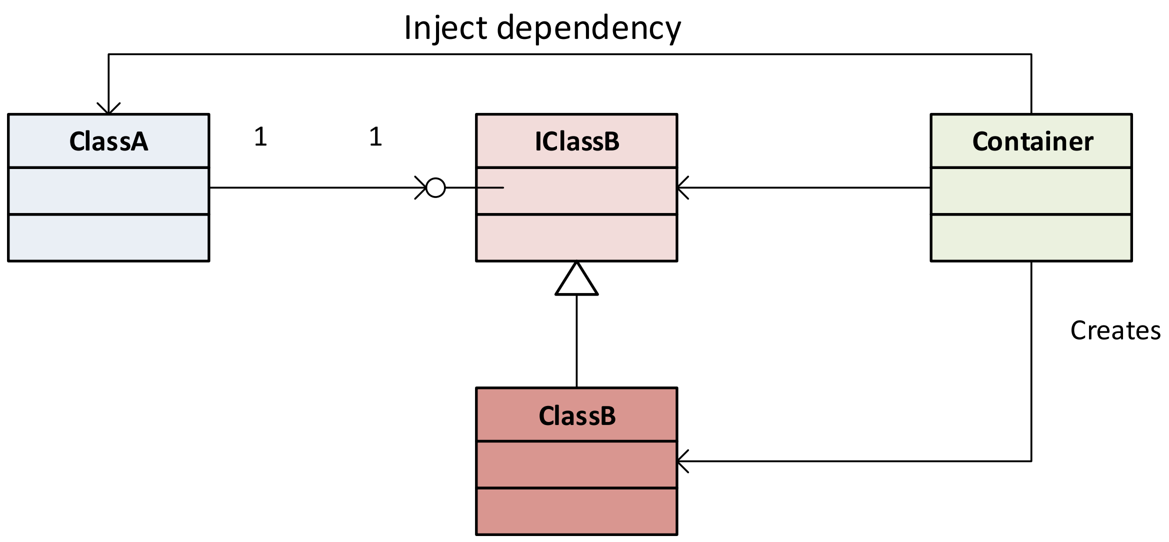
```
/// <summary>
/// Defines contract for implementation
/// </summary>
public interface IClassB
{
}

/// <summary>
/// Implementation of ClassA
/// </summary>
class ClassA
{
    /// <summary>
    /// Private reference of type IClassB
    /// </summary>
    private IClassB classB;

    /// <summary>
    /// Constructor to initialize the members of this class
    /// </summary>
    public ClassA()
    {
        classB = new ClassB();
    }
}

/// <summary>
/// Provides implementation of interface type IClassB
/// </summary>
class ClassB : IClassB
{
    /// <summary>
    /// Constructor to initialize the instance
    /// </summary>
    public ClassB() { }
}
```

Figure 2.17 ClassA has reference of interface of ClassB

To remove the binary relation between classes shown in Figure 2.16, we replace a reference of the dependent class with its interface in the ClassA. As shown in Figure 2.17, ClassA is not directly dependent on ClassB, but ClassA is dependent on an interface of the ClassB.

But still it is not completely loosely coupled as ClassA is still referring to ClassB inside its constructor to initialize an instance of ClassB. This issue can be resolve by injecting ClassB object by Dependency Injection Container (DIC). DIC is a framework that provides a way to configure dependencies and helps to resolve these dependencies. DIC works in three-step (Seemann M. , 2012). First, mapping rules defining the abstract type and concrete type are registered within the container. Then, using these rules concrete types are resolved and injected into the application. Lastly, container disposed concrete objects when they are no longer in use.

The dependency injection will allow us to inject the ClassB object at runtime and we can also inject other concrete implementation of IClassB as program evolves. As mentioned in Figure 2.18, ClassA only has reference of interface IClassB. Container class creates an object of ClassB, and then it injects the dependent object (ClassB) to ClassA.



```
/// <summary>
/// Defines contract for implementation
/// </summary>
public interface IClassB { }

/// <summary>
/// Implementation of ClassA
/// </summary>
class ClassA
{
    /// Private reference of type IClassB
    private IClassB classB;

    /// <summary>
    /// Constructor to initialize the members of this class
    /// </summary>
    /// <param name="classBInstance">Instance of type IClassB</param>
    public ClassA(IClassB classBInstance)
    {
        classB = classBInstance;
    }
}

/// <summary>
/// Provides implementation of interface type IClassB
/// </summary>
class ClassB : IClassB
{
    /// <summary>
    /// Constructor to initialize the instance
    /// </summary>
    public ClassB() { }
}
```

```
/// <summary>
/// Class that creates and injects dependency into ClassA
/// </summary>
class Container
{
    /// Holds the reference of type IClassB
    private IClassB objB;

    /// Instance of ClassA
    private ClassA objA;

    /// <summary>
    /// Contstructor responsible for initializing the member objects
    /// </summary>
    public Container()
    {
        this.objB = new ClassB();
        this.objA = new ClassA(this.objB);
    }
}
```

Figure 2.18 Dependency injected by Container class

Dependency injection is mostly used for loosely coupled design. It is commonly used for unit testing and validation/exception management (Culp, 2015). For proper unit test, a class should not depend upon any external class. If any such dependencies exist, we can replace it with the mock implementation of that external class. Validation/Exception management can be done by injecting validation/exception code into the class. DI makes the program more extensible, modifiable, reusable and maintainable by reducing the dependencies between classes (Fowler, 2004; Seemann M. , 2012). Dependency injection is best explained with coding examples in the next chapter.

Many researchers have used dependency injection for privacy and security in their work. Benenson, Fort, Freiling, Kesdogan, & Penso (2006) proposed a smart card based framework for Secure Multiparty Computation (SMC). This model consists of multiple processes having their security module which securely interact with security modules of other processes. In this paper, authors have used DI to configure the component that selects the actual algorithm at runtime without recompiling the code. Livne, Schultz and Narus (2011) presented an architecture where data from multiple heterogeneous health informatics data sources can be queried using a federated query engine. In this work, dependency injection, AOP, XML configurations and other such best practices were used to make the architecture as flexible, reusable, loosely coupled and service-oriented. These technologies also ease the deployment of the application by using simple user-friendly web console. Similarly, Je ˇzek et al. (2012) has also used DI in their work. In their research, they proposed a framework that can 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. Almorsy, Grundy and Ibrahim (2012) proposed 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, vulnerability mitigation component inject calls to security handler classes at runtime based on the mitigation actions of that vulnerability. Related work suggests there is no work done related to dependency Injection in the field of privacy. This paper will contribute in this dimension.

2.2.6 Mocking

For unit testing, we want to test a class without interacting with other class. For real world application, it is challenging as a class or component interacts with many other classes. Mocking plays an important role in unit testing as it separate the external dependencies from the unit that needs to be tested. Instead of providing actual dependencies, programmer provides mock objects for dependencies to the unit under test. In this way, unit in question can be tested in isolation.

Bender and McWherter (2011) used the term mock to refer to a family of similar implementations to replace real external resources during unit testing. Other related term used with mock is stub (Fowler, 2007). But there is a difference between mock and stub. Fowler (2007) called stub useful for state verification as it acts as stand-in resource and only provide necessary data for the unit test. While mock also includes behavior verification. Mock has the ability to verify which method, how frequent and in which order the method is called and it reacts according to this information (Bender & McWherter, 2011).

A mock object or isolation framework is a reusable library which provides a way to create and configure fake objects at runtime. These fake objects are referred as dynamic stubs and dynamic mocks. Stub objects replace existing objects so that we can test other objects in our code without any problem. We classify fake objects as mocks when we want to verify whether a test failed or passed. The easiest way to distinguish between the mock and stub objects is that the stub can never fail a test. Isolation frameworks are widely used in test driven development (TDD). The use of dynamic fake object eliminates the need to write classes or provide the implementation of the interfaces, because this framework facilitates the developer to generate the code at runtime.

Dependency Injection can increase the testability of the software by mocking or faking dependent objects or external resources such as database and web service thus allowing testing the component in isolation from other components. Mocking is mostly used for unit testing. It makes the testing fast due to decoupling with external resources. It also helps tester to replicate the error easily. Suppose many developers are working on a product and are using the same database to change the state of the data. Unit testing using real object may result in the failure of a developer’s test because another developer might have altered the data. Mocking ensure that unit test is localized by replacing dependent external resources with fake object or data required for the unit test which provide controllability, observability and predictability. By using mocking, developers can provide varied input and can ensure their result accordingly (Bender & McWherter, 2011; Mackinnon, Freeman, & Craig, 2001).

In literature, mocking is often used for testing and for privacy of the personal information. Beresford, Rice, Skehin, & Sohan (2011) propose modified version of Android operating system called MockDroid to mock resource accessed by an application. For example, application that requests IP connectivity, location data, read-write access to calendar data, user may provide mock data instead of actual data to the application. This fake data might impair the application functionality for example; not providing location information to Google Maps application will not show the correct results on the map. By using this operating system, users are allowed to revoke access to particular resource, enabling them to choose between the application functionality and the disclosure of the personal information while using the application. MockDroid is helpful to control additional features of the applications, control sharing of personal data, control costs by avoiding expensive operations performed by an application and for testing.

Hornyack, Han, Jung, Schechter, & Wetherall (2011) also propose to provide fake or empty data to application that required access to it. Data can also be marked as local-only that is, it cannot be transferred over a network.

In another paper, Zhou, Zhang, Jiang, & Freeh (2011) also proposed the use of fake data for application that request users' personal data. User can view all the permissions that application was requesting 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. Trusted mode provides complete real data, while bogus mode provides some of the valid information. Anonymous mode returns useful but unidentifiable data in comparison to empty mode that does not return any of the data. Much of the related works are performed in the field of security. There is not much work done related to mocking in the field of privacy. This paper will contribute in this dimension.

2.3 Discussion

In this section, we have summarized the different researches carried out by industry leading researchers, but mostly all researchers described the methodology specific to a type of application. None of the research focused on how to implement any privacy pattern in any application. And also it is unclear how one can effectively ensure correct data handling without completely redesigning the applications. It also does not demonstrated how to embed the privacy services in the existing applications. Some researchers have used AOP for the protection of the individual's data but none of the researchers have conducted a user study to analyze the ease of use and benefit of using

AOP for the privacy protection. Related work reveals fragmentation in using the software engineering abstractions separately to address privacy, and an absence of software injection patterns for privacy. In this research we will provide a master privacy injection pattern to allow software engineers to incorporate privacy into both legacy and new systems.

Chapter 3 : Privacy Services Injection Pattern

We discussed a number of research methods in Chapter 2, but none of the research describes a framework or pattern on how to integrate other privacy patterns in legacy and/or new software systems. Although it is always recommended to embed privacy services in the early phase of software development (at the time of requirement gathering and modeling), sometimes situations arise when we want to incorporate privacy patterns and their services in existing applications. It is currently problematic for software engineers and testers to incorporate privacy services in later stages of the software development or after go-live. To address that issue, in this thesis, a novel pattern to inject other privacy patterns in the existing system with little or no modification of the existing application is designed. In this section, we explain the architecture of our proposed pattern. For the thesis, we inject de-identification in the existing application using our proposed privacy pattern.

3.1 Design Criteria

The following specify design constraints for a privacy injection pattern.

DR1: As privacy is a cross-cutting concern, privacy integration should be done in a highly modular fashion.

DR2: Integration of privacy patterns into existing code should cause minimal modification to it.

DR3: The pattern should be usable in modern design methodologies e.g. agile design and development and hybrids

DR4: Privacy integration should be automated as completely as possible.

DR5: The learning curve for the software engineer to use the pattern should not be steep

3.2 The Privacy Injection Pattern (PIP)

In recent decades, privacy has become an interesting research topic due to the increase in the sharing of the data between organizations. We identify a key technical challenge as how to inject a privacy pattern and its accompanying service(s) automatically in an existing system. It is always preferable that a legacy system should not be altered, or if it is required, modification should be very minimal and should not affect other existing modules or logic so that a developer inadvertently does not introduce any error in the system. Figure 3.1 shows the architecture of a three tier desktop application. In Figure 3.1, data is transfer from a relational database to the data access layer (DAL) and from the DAL to the business access layer (BAL). From the BAL, data is passed to a presentation layer or a user interface layer. The configuration required for the system is stored in external file such as XML file. Privacy is not implemented in the system.

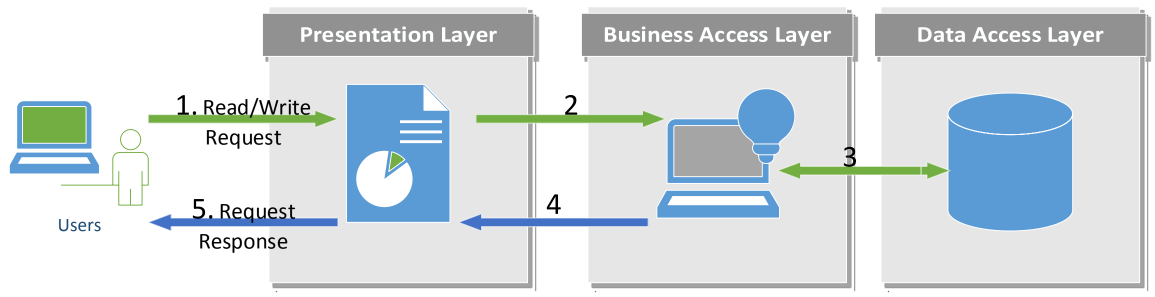


Figure 3.1. Architecture of a 3 tier desktop application

Suppose we need to implement privacy in this system. 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.

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 into a holistic Privacy Injection Pattern (PIP). These three concepts exist independently, but have not been composed into one super-pattern before now for use by software engineers to nimbly embed other privacy patterns and their services in applications. In the PIP, aspects implement known privacy patterns. In the next sections, we will briefly discuss these three concepts, which will help to explain the proposed injection pattern or PIP more clearly. All the examples and sample codes are covered in Microsoft C# language using Microsoft .NET Framework.

3.3 Architecture of the Privacy Injection Pattern (PIP)

Combining the three abstractions (AOP, mocking, and DI), we develop a new privacy injection pattern to insert known privacy patterns or services in new and existing legacy applications. Figure 3.2 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 (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.

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. de-identification, consent, notice) using AOP so that aspects can be used across software implementation classes.

PIP is initialized at the very early stages of the software system. At startup, the software developer loads 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 existing system' DLLs and the privacy program will initialize automatically. When the program loads, a mocked 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 Figure 1), it basically calls the mock BAL object function (3). This mock object fetches data from the business layer as normal. This mock object is similar to the original BAL object layer. It first collects the original information in the similar manner and then applies the privacy pattern over it. The mock object is used to redirect the program flow to the privacy pattern (4). We use the mock object to apply third-party privacy aspects from privacy data patterns (7) and to transfer the modified data to the presentation layer (11). The software engineer can apply privacy patterns implemented as privacy services using aspects that cater for fine-grain privacy attributes such as role, locations, or any other environmental variables. Thus, PIP enables the software engineer to build rich privacy contexts.

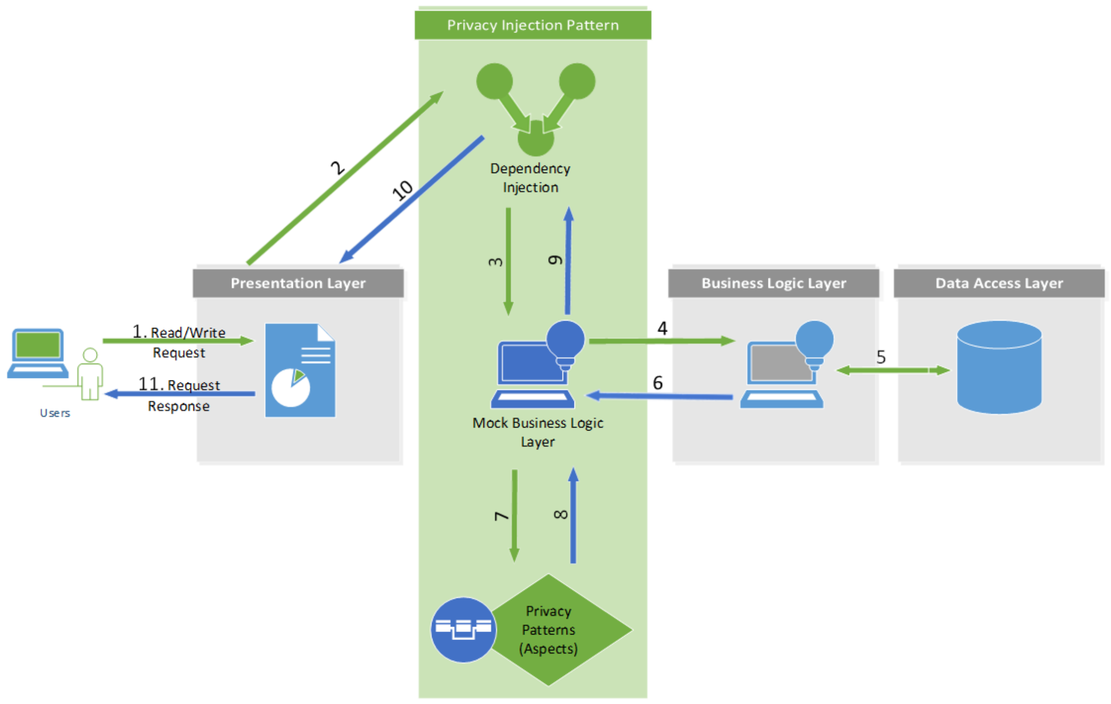


Figure 3.2. Architecture for injecting privacy in legacy application

We can understand the PIP pattern more clearly by using example use cases that are described in next chapter. For the example use cases, we use a de-identification privacy pattern to incorporate in the existing application but indeed a software engineer can use any privacy pattern appropriate to her/his use context.

A mock object or isolation framework is a reusable library, which provides a way to create and configure fake objects at runtime to mimic the behavior of the actual object.

This is a reason, mock objects are extensively used in unit testing and are a widely used approach to replace parts of a program that are not directly relevant to a test case. This thesis proposes to use mock objects in the PIP pattern to replace any code that interacts with data stores, users, or sensors providing data. The use of mocking supports design requirements DR1, DR2, DR3, and DR5.

Dependency injection is proposed for combination in the pattern as it assists the software engineer to automate the injection of privacy patterns in terms of resolving dependencies at runtime. It strongly supports design requirement DR4.

AOP is included in the pattern for its expressive power for cross-cutting concerns, and hence support for DR1 and DR3. It addresses the highly modular design requirement. One of the prominent advantages of AOP is that developers only have to worry about an aspect in one place. The key idea is writing the aspect once and applying it in the solution wherever it is needed. This abstraction helps developers to keep code clean by keeping lots of code out of sight. The downside of AOP is that, a bug in the aspect can take several hours to track and fix (Sonnino, 2014).

Our privacy injection pattern can be used in distributed Service Oriented Architecture (SOA), cloud environment, mobile, as well as in non-web services environments, such as desktop, and many existing client-server and legacy applications. The reason is that this pattern will always reside with business access layer. This makes the pattern to be deployed in any environment setting. For example in cloud environment, organizations hosts their core services which exchanges data with service consumers and our pattern resides on top of business layer which is a part of hosted service. In this way, software engineers do not have to worry about hosting PIP pattern separately.

3.4 Software Engineer’s Learning Curve Perspective

Software engineers use different software pattern to resolve the recurring issues in the software design. Our master privacy injection pattern can be applied to the software applications to incorporate privacy patterns in the existing applications. The software engineer’s understanding of the three main components of the PIP pattern, i.e. dependency injection, mocking, and aspect oriented programming, is necessary for the implementation of our PIP pattern. We will first write an assembly to implement the functionalities required for the privacy pattern to be incorporated in the application example de-identification. Once the assembly is created, we dynamically load the assembly at runtime to avoid modifications in the application. We then create an object of the type mentioned in the assembly and call its function to inject the mock object in the IoC container. We discuss details on the dependency injection in the next section.

3.4.1 Dependency Injection (DI)

Dependency Injection can be implemented to inject object dependencies at runtime. To better understand the Dependency Injection, let us take an example of a banking application where a system will get customer information from the XML file. On the basis of the application user’s role, he/she can view a customer’s complete information or de-identified information. Inversion of Control motivates developers to make higher-level modules dependent on abstraction rather than the concrete implementation of the lower level modules.

We use a banking example to illustrate what the software engineer would need to learn about Dependency Injection. For the implementation of the banking example with DI, the engineer will need to create an interface called *ICustomerManager* with method *GetCustomer*. The dependency injection container is responsible for initialization of the concrete classes and injection of the dependencies to the dependent classes. At runtime, the application decides which concrete implementation of the *ICustomerManager* should be initiated and injected into the application. This task is called registration. Once concrete implementation of the *ICustomerManager* is decided, the application creates that concrete object. This is the resolution task. Once the *ICustomerManager* is available for garbage collection, the application disposes off the *ICustomerManager* instance. This is the disposition task.

There are many dependency injection containers available in implementation platforms such as Microsoft.Net. For further illustration of what a software engineer may need to learn to use PIP, we use the Unity Application Block (Unity) that performs registration, resolution and disposition cycle for dependency injection to work. Let us take the banking example to illustrate how this cycle would work using Unity.

Register

The first task for dependency injection is to register a concrete type in the application. Figure 3.3 shows how we can create a new Unity container and register a type to it. The figure depicts that we are registering a *CustomerManager* object as a concrete implementation of the *ICustomerManager* interface so that when the banking application requires the *ICustomerManager* instance, the application will inject the *CustomerManager* object.

```
if (!Common.Ioc.IocContainer.Instance.IsRegistered(typeof(ICustomerManager)))
{
    Common.Ioc.IocContainer.Instance.Register<ICustomerManager>(
                                             new CustomerManager()
                                             );
}
```

Figure 3.3. Register concrete type to Unity container

Resolve

Resolution is the process of instantiation of the concrete object. In our example, the *CustomerManager* is instantiated using the Resolution method as mentioned in Figure 3.4. In the example, the Unity container instantiates the *CustomerManager* object and its dependent object, if any.

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

Figure 3.4. Instantiation of CustomerManager object

Disposition

As shown in the previous example, the Unity container initializes the *CustomerManager* object that is assigned to the customerInfo variable. When the *CustomerManager* object is eligible for garbage collection, the application disposes off the *CustomerManager* instance. This registration, resolution, and disposition task can be performed manually without dependency injection container but the dependency injection container is the preferred option as it is efficient when dependencies increase and for maintaining a system where requirements change.

3.4.2 Mocking

In the real world, developers write unit tests for a project to test the expected functionality of a class or of a module. Writing unit test cases is a fundamental activity in Extreme programming and Test-Driven Development. Each unit test tests the functionality of a single feature or a method in isolation. If a unit test case fails, it identifies a bug or broken feature in a module (Tillmann & Schulte, 2006). In practice developers dismiss the idea of unit testing the code or a module in isolation because the functionality they want to test is either too complex or too many dependencies are involved. Often configuring and setting up the external dependencies or a system becomes impractical or requires time and effort, which can jeopardize the project's timeline or budget (Seemann, 2004). Configuring such a complex setup for internal/external dependencies is fragile and can break a test, even if written code or the test works perfectly.

In order to avoid such complications, developers rely on mock objects. A mock object or isolation framework is a reusable library, which provides a way to create and configure fake objects at runtime to mimic the behavior of the actual object. This is a reason, mock objects are extensively used in unit testing and form a widely used approach to replace parts of a program that are not directly relevant to a test case. A mock object is a kind of fake object in the system, which decides whether a unit test has passed or failed. It is done by asserting the state of the object being tested to verify if it interacts as expected with the fake object. The mock object saves the communication history, which is used later for verification.

In almost all software systems, objects interact with other objects to complete a task or a part of a task. When writing unit tests we come across similar situations, where an object being tested uses another dependent object over which we do not have control. Examples of such objects are web services, databases, threads, file systems, memory, and time and so on. The key idea is that when a test cannot control what will be the return result from that dependable object and how it will behave during the execution of the test. Let us take an example to understand what the software engineer does to create a mock object in banking example where we want to get customer information from any repository. We have a *CustomerManager* class (Figure 3.5) that is use to get customer information from the XML file and this functionality is provided by a function *GetCustomer*. The function's signature looks like this:

```
public class CustomerManager : SampleBankLibrary.ICustomerManager
{
    /// <summary>
    /// Load customer info from the storage. In this sample application, customer
    /// info is stored in XML file. This method will load customer info from the XML
    /// file and return
    /// </summary>
    /// <returns>
    /// Returns customer information object containing account, transactions etc
    /// </returns>
    public ICustomerInfo GetCustomer()
    {
        var customerInfo = new CustomerInfo();

        var result = LoadFromXmlFile() ;

        return result;
    }
}
```

Figure 3.5. CustomerManager class and GetCustomer function

The *GetCustomer* function will read the XML file stored on the file system to get the information of the customer. The above example is very close to the real world scenario. For testing the above example using a mocking or isolation framework, we have to add a layer that wraps up the calls so that we can mimic the functionality in our tests. Therefore, we have to modify the above example by introducing an interface, *ICustomerManager*, as shown below in Figure 3.6:

```
public interface ICustomerManager
{
    /// <summary>
    /// Return customer information
    /// </summary>
    /// <returns></returns>
    ICustomerInfo GetCustomer();
}
```

Figure 3.6. ICustomerManager interface

The reason for introducing the interface is that we can replace the underlying implementation with any class/object over which we have control. This class/object is an implementation of the interface, *ICustomerManager that* is replaced by a mock object that we can control. The mock object is *customerManagerMock.* Mock implementation is provided in Figure 3.7:

```
var customerManagerMock = new Mock<ICustomerManager>();

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

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

    return new CustomerInfoDeidentifiedImpl(result);
});
```

Figure 3.7. *customerManagerMock* implementation

To implement the pattern, we will not instantiate the *CustomerManager* object, but rather we will hold the instance level reference to the type *ICustomerManager* because now we can hold a reference to any implementation of the *ICustomerManager*. When we want to show complete information of the customer to the application user, we will instantiate *CustomerManager* object and when we want to de-identify the information, we will instantiate *customerManagerMock* object. As shown in the Figure 3.7, we will first create customerManagerMock object using Moq library of the Microsoft .Net Framework. We then setup the implementation of the *customerManagerMock* object in which we first fetch customer information in the similar way as fetch by the original *customerManager* object. The information is then passed to the privacy aspect to de-identify the information whose detail is mentioned in the following section. Once we setup a mock object, we will register the object in the IoC so that whenever any function of the *ICustomerManager* instance is called, it will call the function of the mock object instead of the *GetCustomer* function of the *CustomerManager* class.

3.4.3 Aspect oriented programming

One of the most prominent advantages of AOP is that developers only have to worry to about aspect in one place. The key idea is writing the aspect once and applying it in the solution wherever it is needed. This helps developers to keep code clean by keeping lots of code out of sight, but it’s still there. The downside of AOP is that, a bug in the aspect can take several hours to track it and fix it (Sonnino, 2014). In this section, we will cover an AOP example using our banking application to continue our articulation of what a software engineer must know/learn in order to successfully use the PIP. In this example we will be using PostSharp library for Microsoft .NET Framework. In the Figure 3.8, *LongStringDeidentification* class contains the cross cutting concerns, and this concern will modify the string return from the property that is called. In real life scenarios, these concerns can perform a meaningful task if the called method is successfully executed.

LongStringDeidentification class is inheriting from *LocationInterceptionAspect* class, which is provided by the PostSharp library and it becomes the responsibility of the *LongStringDeidentification* class to override the *OnGetValue* method so that we can de-identify the customer information when getting the value. The reason for overriding the methods is so that we can provide the implementation of our choice for cross cutting concerns. The PostSharp library also makes it mandatory that any class that can be used as an aspect class should be decorated with a *Serializable* attribute.

```
[Serializable]
public class LongStringDeidentification : LocationInterceptionAspect
{
    /// Mask the string from front
    public bool HideFromFront { get; set; }

    /// Character that will be used to mask the string
    public char MaskCharacter { get; set; }

    /// Numbers of characters that will be shown in the original string
    public int VisibleStringLength { get; set; }

    /// Mask the string from the end
    public bool MaskFromBack { get; set; }

    /// Default mask character if no mask character is defined on the attribute
    Private readolny char Default_Mask_Character = '*';

    /// Default number of character that will be visible
    Private readonly int Default_Visible_String_Length = 4;

    /// By Default mask from the end
    Private readonly bool Default_Mask_From_Back = true;

    /// When retrieving the value, it is called to deidentified any long string
    public override void OnGetValue(LocationInterceptionArgs args)
    {
        base.OnGetValue(args);

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

        string value = (string) args.Value;
        if (String.IsNullOrEmpty(value)) return;

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

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

        args.Value = value;
    }
}
```

Figure 3.8. LongStringDeidentification class

OnGetValue method of the *LongStringDeidentification* class first gets the value of the property on which we apply the aspect. If the value is empty then the method will be exited. If we want to apply the masking from the beginning of the value return from the property then we will replace all the characters with mask character except for the last *VisibleStringLength* characters. If we want to apply the masking at the end of the value return from the property then we will show first *VisibleStringLength* characters of the string and then replace the remaining characters of the string with mask character.

LongStringDeidentification is used as shown in the Figure 3.9 below. As we can see that an *AccountInfoDeidentifiedImpl* class has a property *AccountNumber* and this property is decorated with the *LongStringDeidentification* class. Whenever we get *AccountNumber*, the *OnGetValue* method from the *LongStringDeidentification* class will be executed.

```
public class AccountInfoDeidentifiedImpl : SampleBankLibrary.IAccountInfo
{
    /// <summary>
    /// account numbers that customer hold
    /// </summary>
    [LongStringDeidentification(MaskCharacter = '*', VisibleStringLength = 5)]
    public string AccountNumber { get; set; }
}
```

Figure 3.9. Application of LongStringDeidentification aspect class

Chapter 4 : Prototype for Proof of Concept

In the last chapter, we proposed the PIP pattern and explained the software engineer's learning curve around the three main components of the PIP pattern. In this chapter, we illustrate the ease of use and simplicity of implementation of our composite Privacy Injection Pattern (PIP) in the context of two use cases. The first example, a banking use case, injects a well-known character masking de-identification pattern, while the second example, a hospital use case, injects k-anonymity .

4.1 Banking System Use Case

To illustrate ease of use and simplicity of implementation of our composite Privacy Injection Pattern (PIP), we employ PIP in a use case scenario from a banking application that uses de-identification patterns for protecting privacy. 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 (Cavoukian & Khaled El Emam, 2014; Shapiro, 2011; Narayanan & Shmatikov, 2008). Some de-identification patterns are substitution, shuffling, nulling out, character masking and cryptographic techniques. We implement the nulling out and character masking privacy patterns for illustration using aspect-oriented programming (AOP) in our example. We show how to use mocking and dependency injection techniques to automatically inject an 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 implement an example injection of our de-identification service (Cavoukian & Khaled El Emam, 2014) into a banking application. We note that the PIP may be implemented with other technologies, e.g. multi-platform heterogeneous technologies.

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. Figure 4.1 shows the sequence diagram of the Maintain Users' Account Use Case before applying PIP. Once the user is login into the system, system creates customer manager and role manager object and retrieves account and transaction details of all the users and displays it on the screen.

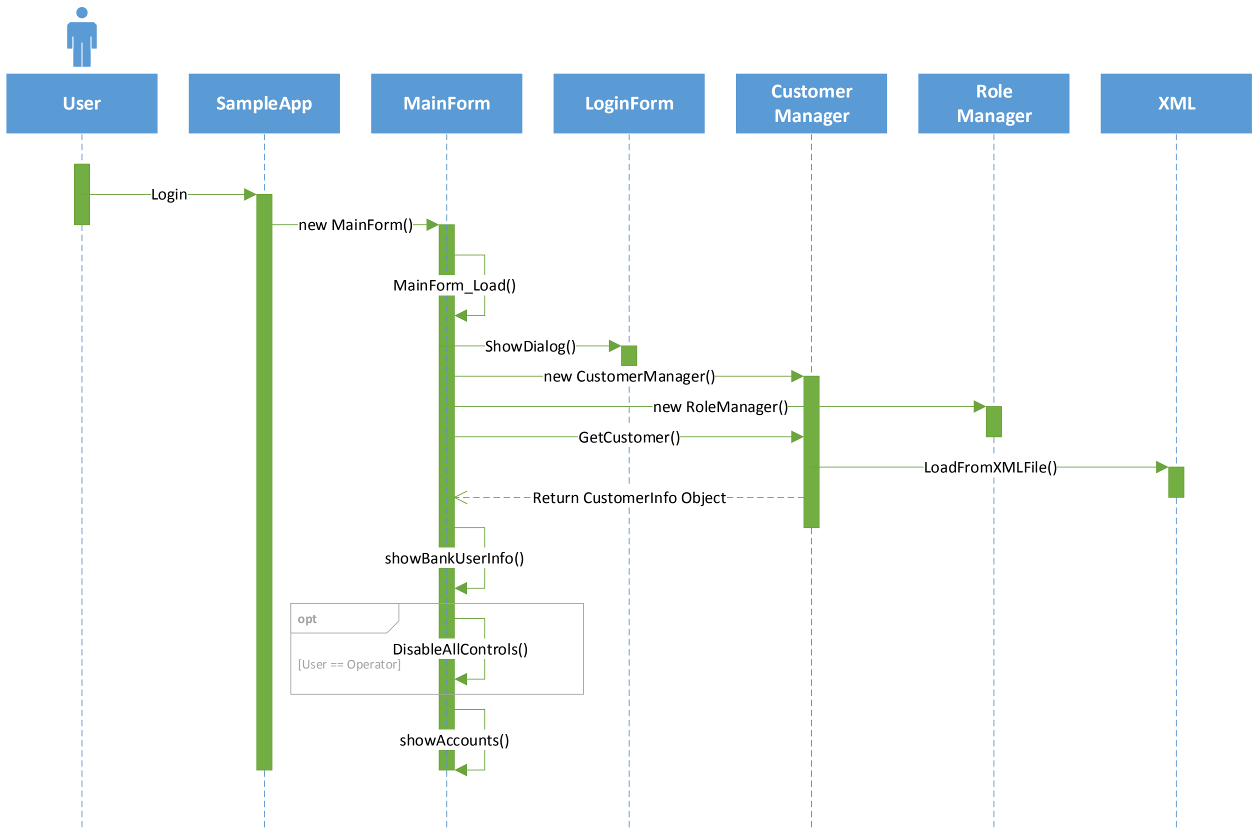


Figure 4.1. Sequence Diagram of User Account before Applying PIP

In this Use Case, we want to 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. De-identification is thus not applied for the manager. The de-identification service DLL is loaded in the main program. Figure 4.2 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 deidentificationServiceLibName = "BankDeidentificationService.dll";

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

    var deidentificationServiceLibCompletePath = Path.Combine(currentPath,
                                                  deidentificationServiceLibName);

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

    Assembly assembly = Assembly.LoadFrom(deidentificationServiceLibCompletePath);
    var deidentificationServiceType = assembly.GetType(
                                "BankDeidentificationService.DeidentificationService");

    var serviceInstance = Activator.CreateInstance(deidentificationServiceType);

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

Figure 4.2. 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. The *CustomerManager* has a method called *GetCustomer* that fetches customer and account detail 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 *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 4.3 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 to resolve the object to access its methods.

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

public static void SetupCustomerManager()
{
    if (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 CustomerInfoDeidentifiedImpl(result);
    });


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

Figure 4.3. Inject mocking object and implementation of the *GetCustomer()* method

As the software developer has registered the mock object in IoC container, when we call Customer Manager object, it will call mock customer manager object. Figure 4.4 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 GetCustomer method of the mock object and applies de-identification on the object. After applying de-identification, the system displays the information on the screen.

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

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

this.customerInfo.Accounts.ForEach(account =>
{
    this.cbAccount.Items.Add(string.Format("{0}-{1}",
                                           account.AccountType,
                                           account.AccountNumber));
});
```

Figure 4.4. Resolve mocking object at runtime to get customer information

We apply the de-identification service by applying 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. As shown in Figure 4.5, we apply de-identification on the account number property, which anonymizes account number by showing only 5 characters from the end of the string and replacing all other characters from asterisk (*). For example from 4455368489645247 to ***********45247. This de-identification technique is called character masking. Since we apply anonymization on the property, when we try to access the property, it applies the de-identification aspect on the field and returns a value.

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

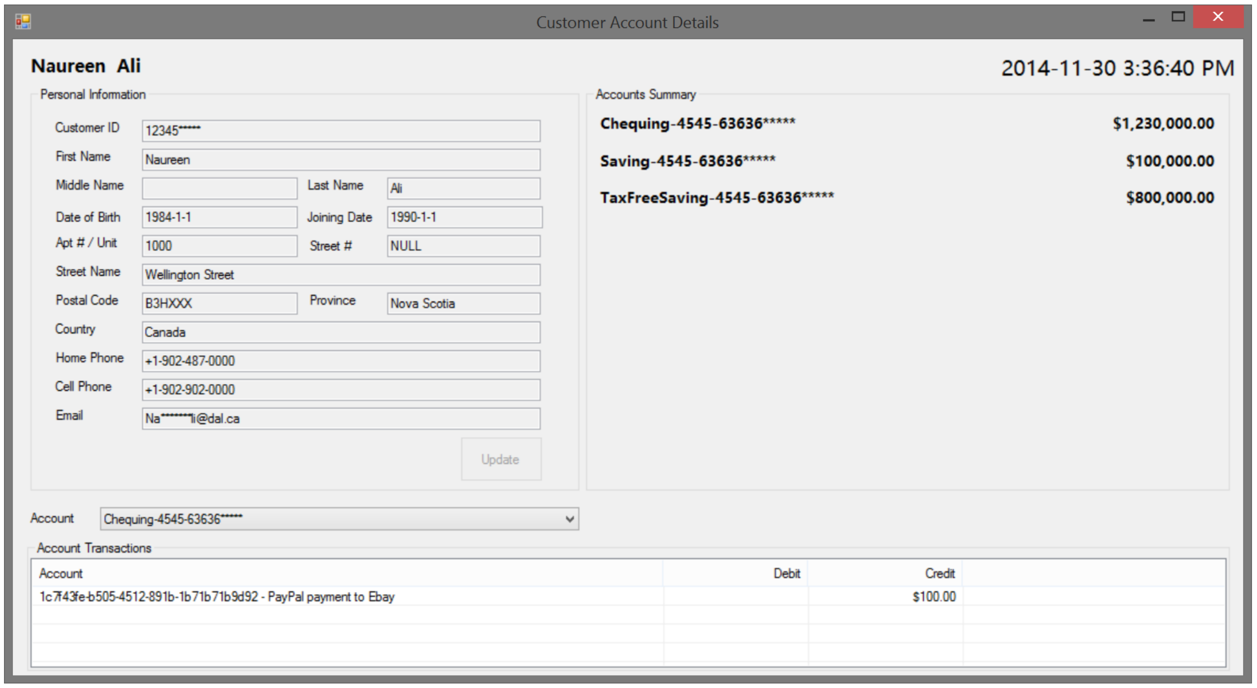
Figure 4.5. Apply *LongStringAnonymization* aspect on *AccountNumber*

We apply *LongStringAnonymization* to *AccountNumber* property. In the *LongStringAnonymization* class, we provide the de-identification logic that will be applied on the field on which we bind this aspect. Figure 4.6 shows the implementation of *OnGetValue* function of *LongStringAnonymization* class. This function is called whenever we try to get a value of some property. In this function, we provide the masking logic by replacing required number of characters with the given character. In this way, we implement the aspect class 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);

    var roleManagerInstance = Common.Ioc.IocContainer.Instance.Resolve<IRoleManager>();

    if (roleManagerInstance.UserRole == Role.Manager)
        return;

    string value = (string) args.Value;

    if (String.IsNullOrEmpty(value))
        return;

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


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

    args.Value = value;
}
```

Figure 4.6. De-identification implementation in LongStringAnonymization class

Figure 4.7 shows an excerpt of the sample bank application. The user is logged in with an operator role. The operator role does not have permission to view all the private information about the customer. The private information in Figure 4.7 is de-identified using the proposed pattern. Different fields' data are de-identified using different de-identification techniques for example for customer id field, we apply character masking; for date of birth we use date variance, and we null out the street number. In Figure 4.8, the user is logged in as an administrator and the de-identification service is not applied. The administrator role has the rights to view all the information and can update them.



Naureen Ali

2014-11-30 3:36:40 PM

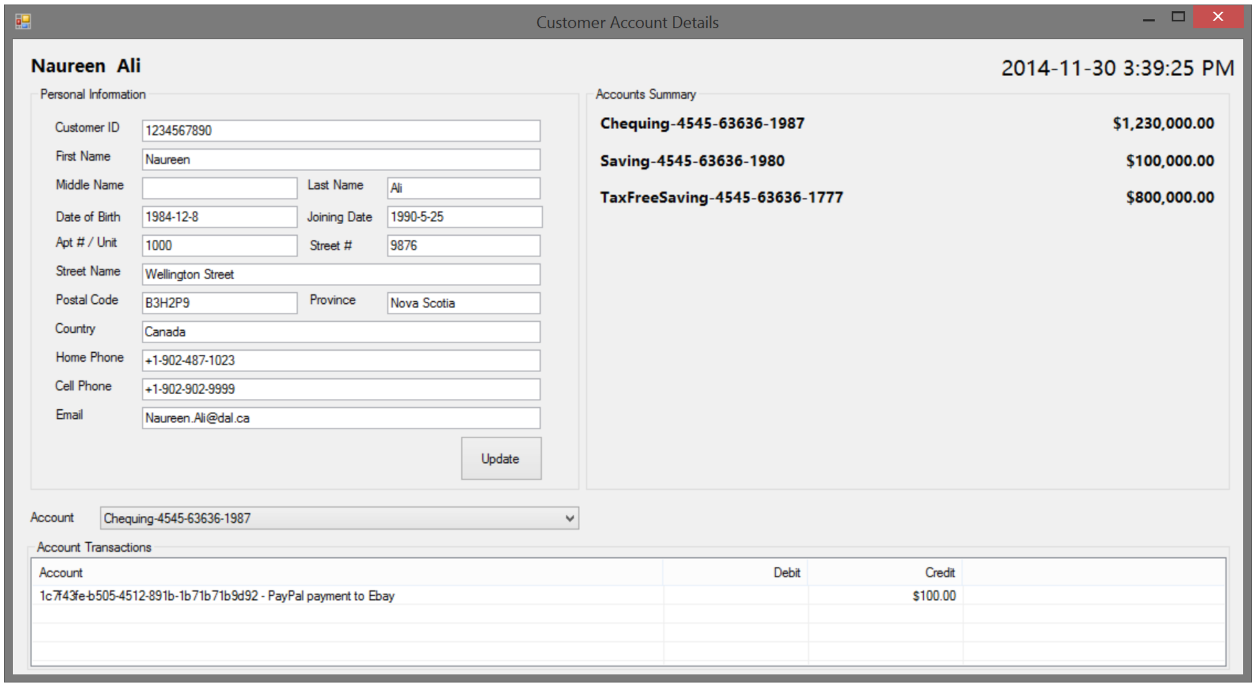
Customer ID	12345*****
First Name	Naureen
Middle Name	
Last Name	Ali
Date of Birth	1984-1-1
Joining Date	1990-1-1
Apt # / Unit	1000
Street #	NULL
Street Name	Wellington Street
Postal Code	B3HXXX
Province	Nova Scotia
Country	Canada
Home Phone	+1-902-487-0000
Cell Phone	+1-902-902-0000
Email	Na*******li@dal.ca

Chequing-4545-63636*****	$1,230,000.00
Saving-4545-63636*****	$100,000.00
TaxFreeSaving-4545-63636*****	$800,000.00

Account: Chequing-4545-63636*****

Account	Debit	Credit
1c7f43fe-b505-4512-891b-1b71b71b9d92 - PayPal payment to Ebay		$100.00

Figure 4.7. Sample Bank Application – User log in with Operator role



Naureen Ali

2014-11-30 3:39:25 PM

Personal Information

Customer ID	1234567890		
First Name	Naureen		
Middle Name		Last Name	Ali
Date of Birth	1984-12-8	Joining Date	1990-5-25
Apt # / Unit	1000	Street #	9876
Street Name	Wellington Street		
Postal Code	B3H2P9	Province	Nova Scotia
Country	Canada		
Home Phone	+1-902-487-1023		
Cell Phone	+1-902-902-9999		
Email	Naureen.Ali@dal.ca		

Accounts Summary

Chequing-4545-63636-1987	**$1,230,000.00**
Saving-4545-63636-1980	**$100,000.00**
TaxFreeSaving-4545-63636-1777	**$800,000.00**

Account: Chequing-4545-63636-1987

Account Transactions

Account	Debit	Credit
1c7f43fe-b505-4512-891b-1b71b71b9d92 - PayPal payment to Ebay		$100.00

Figure 4.8. Sample Bank Application – User log in with Administrator role

Figure 4.9 shows the sequence of objects call when PIP is applied on the maintain users account use case to inject role-based de-identification for access control. The PIP can be applied to inject other privacy patterns in the system.

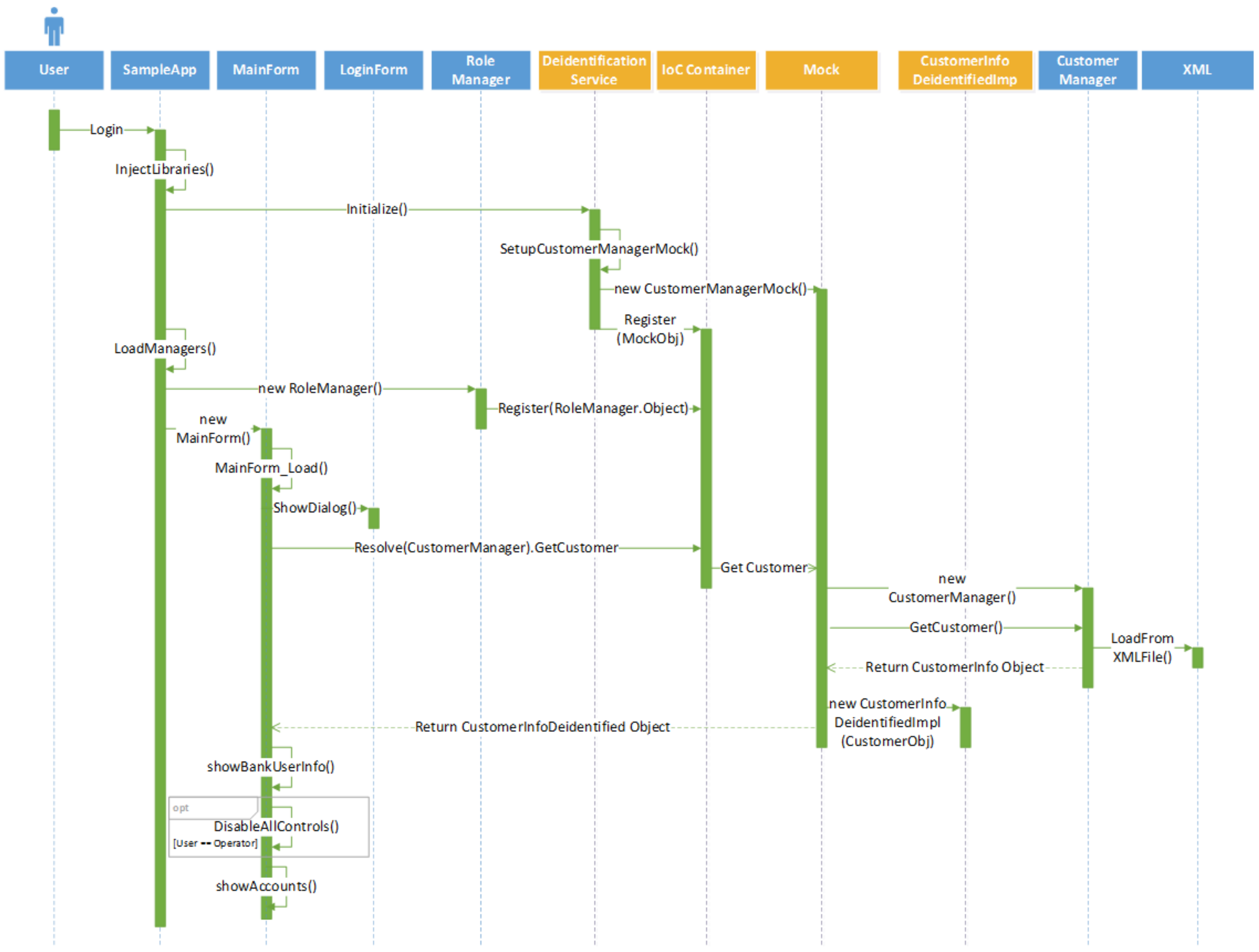


Figure 4.9. Main User Account Sequence Diagram after Applying PIP

We suggest that the PIP pattern can be used repeatedly in many places in a banking application e.g. to also inject a location/time privacy pattern that disallows the operator from viewing even more of customers' fields remotely outside of banking hours.

4.2 Hospital System Use Case

Implementing privacy in healthcare applications has become a leading concern of many researchers in the last few decades. According to William J. Clinton, "As more of our medical data are stored electronically, the threats to all our privacy increase" (Sharma et al., 2014). To determine whether it is easy and simple to implement PIP in a more complex use case, we employ PIP in a use case scenario from a hospital management application that uses k-anonymity as the de-identification pattern while sharing data with other organizations. We implement the k-anonymity privacy method using the ARX (ARX, n.d.) DLL for illustration using aspect-oriented programming (AOP) in our example. ARX is the Java-based open source graphical tool for anonymizing personal data. The tool supports different data import and cleansing techniques. We can use many data transformation techniques such as generalization, suppression, and micro aggregation and different privacy models such as k-anonymity, ℓ-diversity, and t-closeness using the ARX tool. ARX also provide visualizations of data utility and re-identification risks. ARX is also available as a fully featured software library that delivers data anonymization capabilities to any Java program. We are using Microsoft .Net for the thesis; we can transform the ARX Java-based library to the .Net library using a Java and .NET interoperability tool. We are using (IKVM.NET, 2015) which is the Java Virtual Machine (JVM) for the .NET and Mono (Java) runtimes. It can convert Java jars into .NET assemblies. We used IKVM to transform the ARX jar file into .Net DLL.

We show that the mocking and the dependency injection techniques automatically inject the AOP instance of the de-identification service. The k-anonymity is a de-identification method and it helps to preserve sensitive information. The idea behind k-anonymity is to reduce the granularity of the representation of the data in such a way that a given record cannot be distinguished from at least (k-1) other records (Aggarwal & Yu, 2008). This granularity is reduced using techniques such as generalization and suppression. In generalization, we replace the attribute value with a generalized value. Suppression is the technique where one or more attribute values are removed completely.

Our technical implementation uses Visual Studio .Net (IDE), PostSharp (AOP), the Unity Container (Dependency Injection), and the Mock library to implement an example injection of our de-identification service into a hospital application. We note that the PIP may be implemented with other technologies, e.g. multi-platform heterogeneous technologies.

The hospital use case scenario is a search screen to retrieve information on the basis of criteria, such as country, city etc. The k-anonymity algorithm is applied to the data before sharing the data with other organizations. There is an option given to the user to apply k-anonymity to the data. This option helps us to study the behavior of the system before and after applying the proposed pattern. Figure 4.10 shows the sequence diagram of the search patient information use case before applying PIP.

In this case study, we want to inject the option-based de-identification pattern such that user can apply k-anonymity on the patient information to share the information with other organizations or can view all information for the organizations' needs. The de-identification service DLL is loaded in the main program.

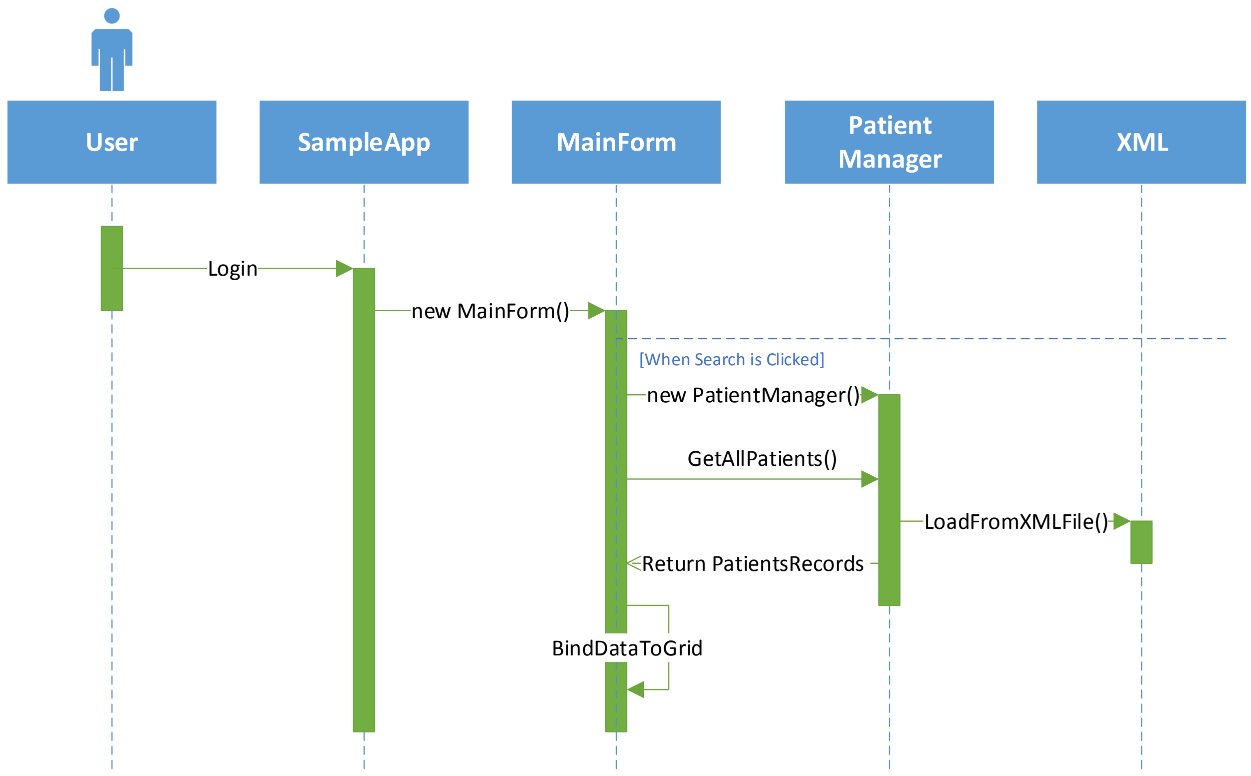


Figure 4.10. Sequence Diagram of Patient Search before Applying PIP

Figure 4.11 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 deidentificationServiceLibName = "DeIdentificationService.dll";

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

    var deidentificationServiceLibCompletePath = Path.Combine(
                                                  currentPath,
                                                  deidentificationServiceLibName);

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

    Assembly assembly = Assembly.LoadFrom(deidentificationServiceLibCompletePath);

    var deidentificationServiceType = assembly.GetType(
                        "DeIdentificationService.DeIdentificationServiceInitializer");

    var serviceInstance = Activator.CreateInstance(deidentificationServiceType);

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

Figure 4.11. 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 *PatientManager*, which is an implementation of the *IPatientManager* interface. *PatientManager* has multiple methods that fetch patients' information from the XML (or database) on the basis of certain criteria. The software engineer creates a mock object of the *IPatientManager* type and then registers it. The engineer also setups the updated implementation of all the methods to fetch patients' details in the same way as the originating object method, and then applies the de-identification aspect on this object.

Figure 4.12 shows the implementation of methods to de-identify the information. Subsequently, when the developer calls the original *PatientManager* method, the updated method of the mock object is invoked. In the Unity Container, for dependency injection the software engineer first registers the object at the beginning of the program to resolve the object to access its methods.

```
private static void SetupManagerObjects()
{
    if (Common.Ioc.IocContainer.Instance.IsRegistered(typeof(IPatientManager)))
          return;

    var patientManagerMock = new Mock<IPatientManager>();

    patientManagerMock.Setup(x => x.GetAllPatients()).Returns(() =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.GetAllPatients();

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();
        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });

    patientManagerMock.Setup(x => x.SearchPatientByCity(It.IsAny<string>()))
       .Returns((string cityName) =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.SearchPatientByCity(cityName);

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();
        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });

    patientManagerMock.Setup(x => x.SearchPatientByCountry(It.IsAny<string>()))
       .Returns((string countryName) =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.SearchPatientByCountry(countryName);

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();
        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });

    Common.Ioc.IocContainer.Instance.Register<IPatientManager>(
                                            patientManagerMock.Object);
}
```

Figure 4.12. Inject mocking object and invoke IOC

Figure 4.13 shows how the software developer resolves the *IPatientManager* object to fetch patients' information. The developer will call the original function, for example *GetAllPatients*, to fetch the required information. This action calls the GetAllPatients method of the mock object and applies de-identification on the object. After applying de-identification, the system displays the information on the screen.

```
IEnumerable<IPatientInfo> searchedPatientRecords = new List<IPatientInfo>();

var patientManager = Common.Ioc.IocContainer.Instance.Resolve<IPatientManager>();

switch (this.cmbSearchBy.SelectedItem.ToString())
{
    case  "Country":
        searchedPatientRecords = patientManager.SearchPatientByCountry(
                                                this.txtSearchTerm.Text);
        this.BindDataToGrid(searchedPatientRecords);
        break;

    case "City":
        searchedPatientRecords = patientManager.SearchPatientByCity(
                                                this.txtSearchTerm.Text);
        this.BindDataToGrid(searchedPatientRecords);
        break;

    case "All Records":
        searchedPatientRecords = patientManager.GetAllPatients();
        this.BindDataToGrid(searchedPatientRecords);
        break;
}

this.lblRecordsCount.Text = string.Format(
                              "{0} {1} found",
                              searchedPatientRecords.Count(),
                              searchedPatientRecords.Count() > 1 ? "Records":"Record");
```

Figure 4.13. Resolve mocking object at runtime to get patient information

The software engineer applies the de-identification service by applying a de-identification aspect with properties or methods. In this example, the developer applies de-identification on the *PatientRecords* property. When the software fetches patient records through this property, it applies the de-identification aspect on the field and returns a value.

```
public class PatientInfoRecordCollection
{
    [DeIdentificationAspect(AnonymizationFactor = 3)]
    public List<PatientInfoDeIdentified> PatientRecords { get; set; }
}
```

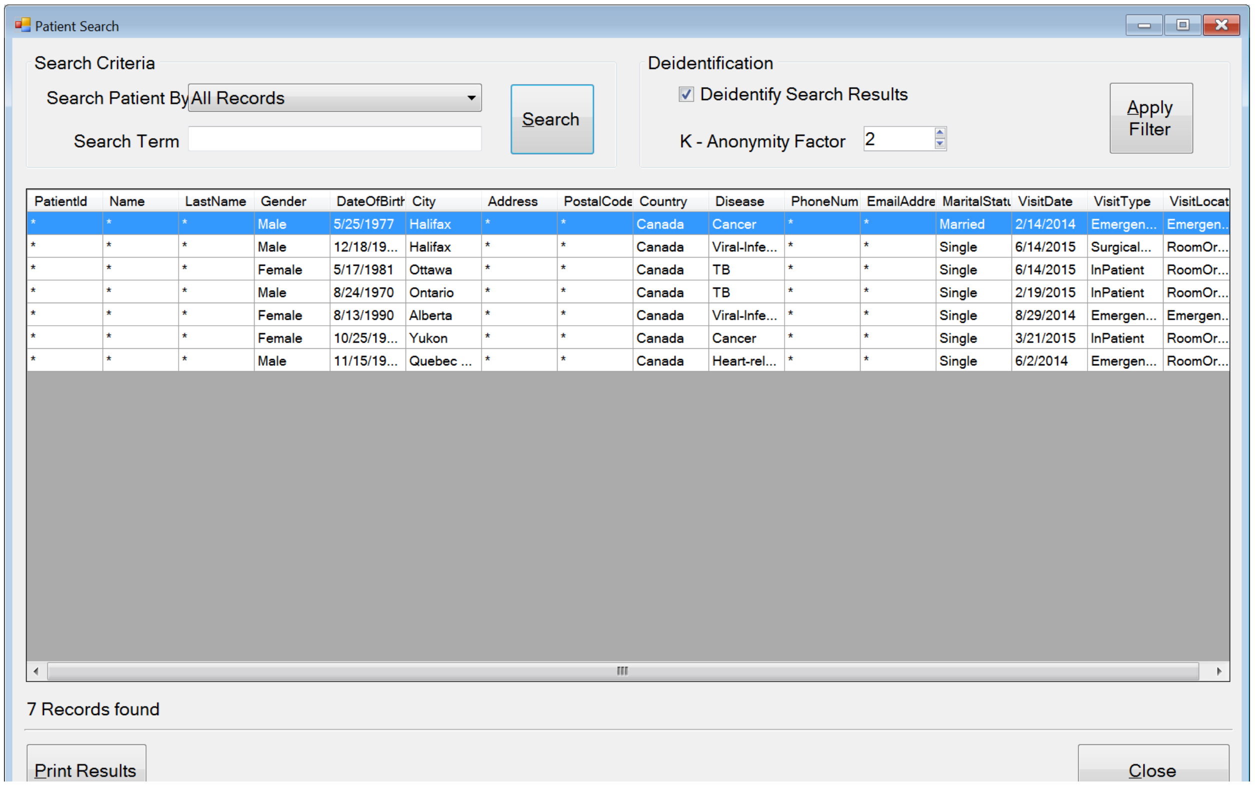
Figure 4.14. Apply Deidentification aspect on *PatientRecords*

The software developer applies the *DeidentificationAspect* to the *PatientRecords* property (Figure 4.14). In the *DeidentificationAspec*t class, s/he provides the de-identification logic that will be applied on the bind field as in Figure 4.15. *DeidentificationAspect* then calls *DeidentifyRecords* of the *DeIdentification* class to use the ARX anonymize function to de-identify the data and return it to the calling method.

```
[Serializable]
public class DeIdentificationAspect : LocationInterceptionAspect
{
    public int AnonymizationFactor { get; set; }

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

        // DeidentifyRecords uses ARX anonimize function to anonimize the list
        // provided as argument
        var newValue = new DeIdentification<PatientInfoDeIdentified>()
                    .DeIdentifyRecords(
                           (List<PatientInfoDeIdentified>) args.Value,
                           this.AnonymizationFactor);

        args.Value = newValue.ToList();
    }
}
```

Figure 4.15. De-identification implementation in *DeidentificationAspect* class

Figure 4.16 shows a patient search screen of the hospital application that results from the use of the PIP for injection of the k-anonymity method. The patients' information is searched by different criteria and the k factor for k-anonymity is also provided by the user on the screen. For k-anonymization of the records, we provide the attribute type (identifying, quasi-identifying, insensitive) of each attribute in the input list. For the quasi-identifying attribute, an attribute hierarchy is required.


Patient Search

Search Criteria

Search Patient By: All Records

Search Term:

Search

Deidentification

☑ Deidentify Search Results

K - Anonymity Factor: 2

Apply Filter

PatientId	Name	LastName	Gender	DateOfBirth	City	Address	PostalCode	Country	Disease	PhoneNum	EmailAddre	MaritalStatu	VisitDate	VisitType	VisitLocat
*	*	*	Male	5/25/1977	Halifax	*	*	Canada	Cancer	*	*	Married	2/14/2014	Emergen...	Emergen...
*	*	*	Male	12/18/19...	Halifax	*	*	Canada	Viral-Infe...	*	*	Single	6/14/2015	Surgical...	RoomOr...
*	*	*	Female	5/17/1981	Ottawa	*	*	Canada	TB	*	*	Single	6/14/2015	InPatient	RoomOr...
*	*	*	Male	8/24/1970	Ontario	*	*	Canada	TB	*	*	Single	2/19/2015	InPatient	RoomOr...
*	*	*	Female	8/13/1990	Alberta	*	*	Canada	Viral-Infe...	*	*	Single	8/29/2014	Emergen...	Emergen...
*	*	*	Female	10/25/19...	Yukon	*	*	Canada	Cancer	*	*	Single	3/21/2015	InPatient	RoomOr...
*	*	*	Male	11/15/19...	Quebec ...	*	*	Canada	Heart-rel...	*	*	Single	6/2/2014	Emergen...	RoomOr...

7 Records found

Print Results

Close

Figure 4.16. Patient Search Screen of Hospital Application

Figure 4.17 shows the sequence of objects calls when PIP is applied on the Patient Search use case to inject privacy by using k-anonymity in the system.

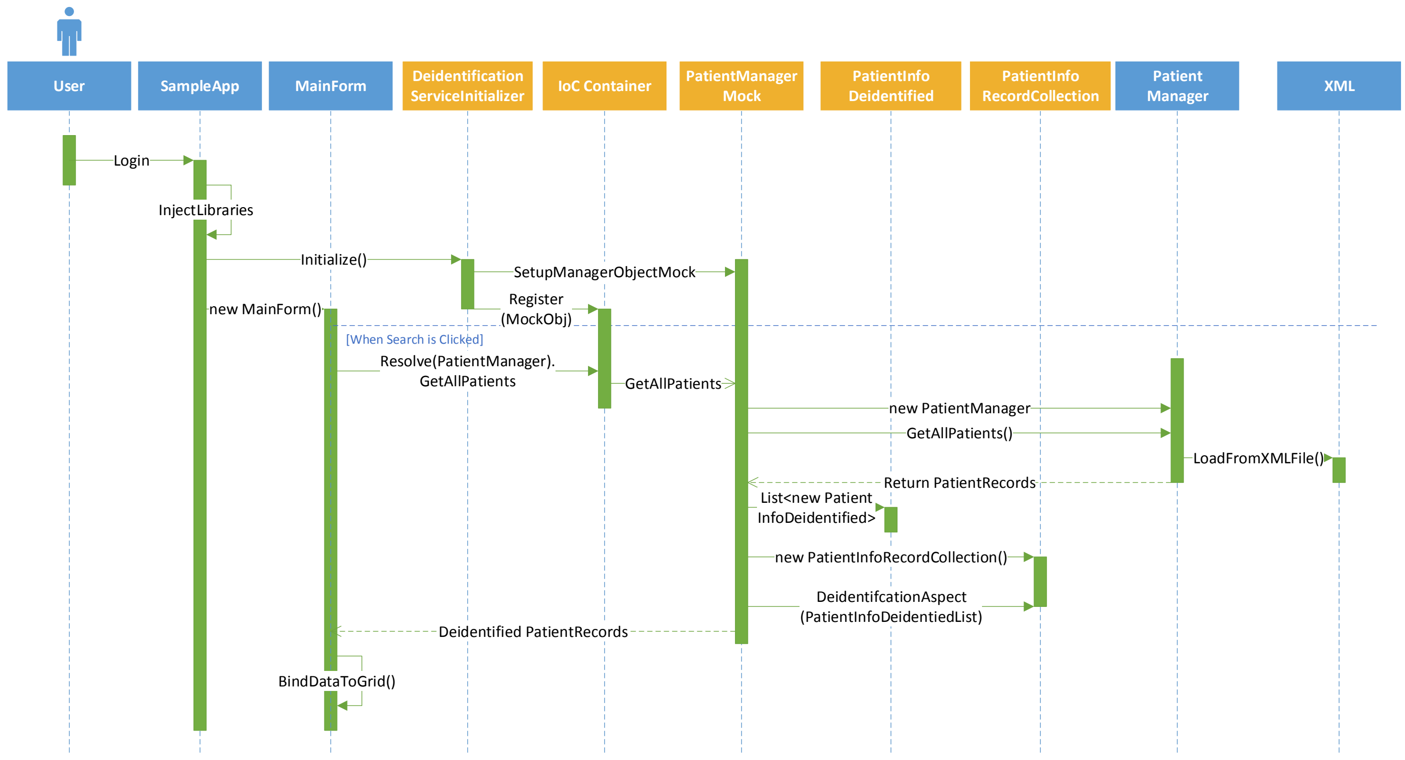


Figure 4.17. Patient Search Sequence Diagram after Applying PIP

Chapter 5 : Results and Evaluation

In this chapter, we provide findings from the online survey. In section 5.1, an instrument for evaluating the usefulness and ease of use of PIP is created based on the TAM (Technology Acceptance Model) model. Section 5.2 discusses the participants, including their demographics and distribution throughout the survey. In section 5.3 we focus on the perceived usefulness and perceived ease of use of the participants. In section 5.4 we present the participants' opinion to improve the PIP pattern. In section 5.6, we present the motivation of the participants to use the pattern when they encounter a situation where they need to incorporate privacy in their application without modifying the underlying application or changing it to some extent. At the end of the chapter, we will discuss the limitations of the user study.

5.1 Evaluation Methodology

Researchers have used aspect oriented programming (AOP) for privacy as mentioned in section 2.2.4.3, but to the best of our knowledge, none of the researchers have conducted a survey to evaluate AOP's usefulness. Many researchers have suggested that adoption of technologies is highly dependent on the user acceptance (Davis, Bagozzi, & Warshaw, 1989; Hartwick & Barki, April 1994). The Technology Acceptance Model (TAM) by Davis (1989) is one of the most widely used models to measure technology acceptance (King & He, 2006). According to the TAM model, the reaction of an individual towards a technology influences one's intention to use the technology, which ultimately affects the actual use.

TAM is based on theory of reasoned action (TRA) (Fishbein & Ajzen, 1975). TRA propose that an individual attitude to perform a behavior and the subjective norm about that behavior influences one's intention to perform the specific behavior and the behavior itself (Figure 5.1). The individual attitude can be determined by the perceived consequences of performing that behavior multiplied by the evaluation of the consequences. Subjective norm can be defined as "*The person's perception that most* *people who are important to him think he should or should not perform the behavior in question*" (Fishbein & Ajzen, 1975). Thus, subjective norm is the perception of the surrounding people about the individual's intention to perform the behavior. According to the TRA, the individual's attitude and subjective norm influence his/her behavioral intention, which is the probability of an individual to perform a behavior. Ultimately, behavioral intention impacts the actual performance of the behavior. TRA provides general beliefs that will be important in a context for adoption of information technologies. TRA does not focus on specific beliefs. Davis (1989) proposed and validated a comprehensive approach to identify the critical beliefs related to technology adoption in organizations. Davis (1989) identified two common beliefs that influence IT adoption: perceived usefulness (PU) and perceived ease of use (PEOU). These two beliefs are influenced by external variables such as design features of the IT system and organizational training.

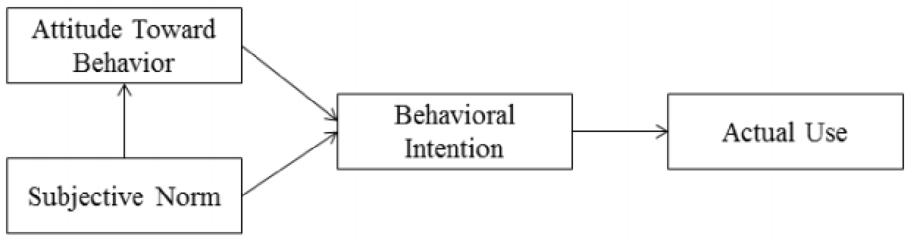


Figure 5.1. Theory of Reasoned Action (Fishbein & Ajzen, 1975)

According to Davis, user perceived usefulness and perceived ease of use influenced his/her attitude and behavior of using the technology. In this research, we are using the Technology Acceptance Model (TAM) to evaluate the usefulness and ease of use of the PIP pattern by conducting a survey with the IT professionals. In this section, we will discuss our survey objectives (section 5.1.1); survey questions (section 5.1.2); survey approach (section 5.1.3) including study protocol (section 5.1.3.1), study instrument (section 5.1.3.2), survey design (section 5.1.3.3), intended data analysis (section 5.1.3.4), recruitment (section 5.1.3.5), and participants (section 5.1.3.6).

5.1.1 Survey Objectives

The purpose of this survey is to understand how useful and beneficial our proposed pattern is to inject privacy in the existing and new applications and whether it is easy to adopt this pattern by the developers. We use Davis's Technology Acceptance Model (TAM) model (Davis, 1989). In this survey, we use the TAM model to evaluate the usefulness and acceptance of the proposed pattern. In our context, the degree to which PIP pattern is easy to use and is useful, as perceived by IT professionals, will affects their attitude toward using the pattern. We conducted an online survey from software developer, software designer, privacy and security engineers. We also analyze the motivation and intention of engineers in using this pattern and what improvements can be incorporated in the PIP pattern, if any. From our findings, we can provide guidelines and recommendations to the developers and designers on how to incorporate privacy in the application.

5.1.2 Survey Questions

For the survey we conducted, we used questions which are validated by Davis (1989) for perceived usefulness and perceived ease of use of the PIP pattern. Our high-level survey questions are as follows:

1) Will the PIP pattern be useful to the engineers in their job if they encounter to incorporate privacy in their existing application?
2) Will the PIP pattern be easy to use to the engineers to incorporate privacy in the application?
3) Will engineers intended to use this pattern when required?
4) What improvements do engineers think that could be done in this pattern, if any?

5.1.3 Survey Approach

We conducted an online questionnaire using Opinio software where the participants may have different levels of expertise with software development and design. Their expertise ranges from that found in any new junior software developers to senior software architects and privacy engineers. The questions were designed to analyze the perceived usefulness and perceived ease of use of the PIP pattern and to examine the improvements and motivation of using the pattern by the engineers. We chose an online survey because we wanted to reach a wide variety of participants with different level of exposure with software development and design. We wanted to have at least 30 participants for the survey.

5.1.3.1 Survey protocol

We submitted an ethics application to Dalhousie research board for approval before starting our study. After the approval (see Appendix A), the recruitment script (see Appendix B) was sent to various engineers through personal contacts (explained in section 5.1.3.5). The recruitment script had a link to the survey. A click on the link (or if typed in any web browser) directed participants to an informed consent form (see Appendix C). This was the first page of the online survey. After reading the survey, if they wish to take part, they clicked on the "agree" button to proceed. The next page (see Appendix E) asks for their permission to quote their responses. They may click "*yes*" or "*no*" to "*I agree that the researchers may quote my responses to free form questions*", if they wish their responses to be quoted or not.

5.1.3.2 Survey instrument

We explored various available survey tools. Given the restrictions by Dalhousie University research ethics board regarding the hosting of the survey outside of Canada, we used Opinio for this research.

5.1.3.3 Survey Design

The key challenge for our survey was to evaluate our proposed privacy injection pattern from the point of view of usefulness and ease of use and obtain feedback on it. We customized the questions in (Davis, 1989) where possible for PIP while keeping their core essence. We have divided our questionnaire in five parts i.e. Part A, B, C, D, and E. Figure 5.2 shows the overall flow of the survey.

All participants completed section A. It consisted of demographic questions and questions to access their experience and type of work. Once part A is completed, participants were directed to Part B. Part B consists of details of the Privacy Injection Pattern (PIP) with examples. In part B, details of the proposed pattern was shown to the participants along with a simple and a complex example to understand how the PIP can be implemented in different use cases. Both the examples' code were provided as a downloadable link and in written form so that participants can view whatever format in which he/she is comfortable. Once, participants read the pattern code materials, they are asked about the usefulness and ease of use of the PIP on the basis of Davis's TAM model. At the end of the part B, the participants are asked whether they have executed the examples or tried to modify it. If participants selected "yes", they were routed to Part D else they were routed to Part C. Part C and Part D consists of 27 survey questions for usefulness and ease of use of PIP based on the TAM model.

After part C or part D, participants were directed to part E. In part E, participants were asked if they can improve the proposed pattern. If participants selected "yes", they were asked what changes they would make to improve the pattern. This feedback could provide us with some future directions. Then participants were asked if they were to face a situation where they need to incorporate privacy service or patterns in existing applications, would they prefer to use this pattern? If participants selected "no", then they were asked what would be their preference in implementing privacy in existing application considering that we don't want to modify the existing application or if required, can modify it to a small extent only.

After completing the survey, participants were asked to provide their E-mail IDs if they wished to receive a copy of this study findings when the survey period is complete.

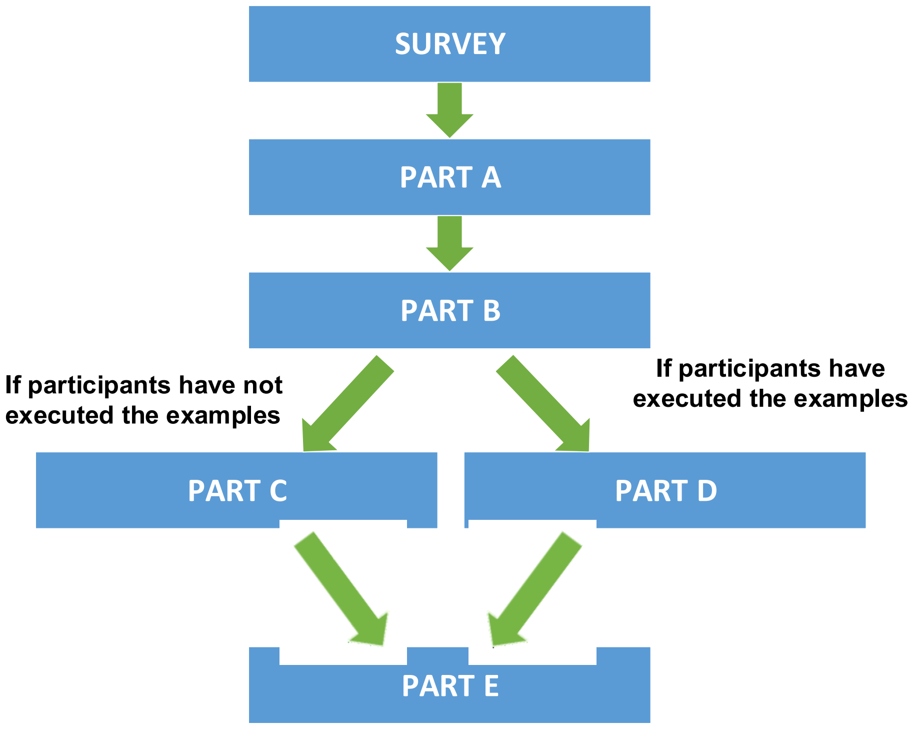

SURVEY
PART A
PART B
If participants have not executed the examples
If participants have executed the examples
PART C
PART D
PART E

Figure 5.2. Survey Design

5.1.3.4 Intended Data Analysis

Our data were both quantitative and qualitative in nature. For multiple choice questions, the option lists were created from the survey of Davis (1989). There were two free form questions to elicit the perspective of participants in more detail. For better explanation of the PIP pattern, we provided participants with the definition of the technological abstractions used in the PIP pattern that is, aspect oriented programming, dependency injection, and mocking. Two examples and their descriptions with attached code were also given to the participants to understand how PIP can be applied to any use case to implement privacy in the existing application. We designed the survey to compare the perceived usefulness and perceived ease of use of those participants who downloaded the examples and tried to execute it with those who read the description and gave the responses. We would like to compare what people perceived about the pattern whether it is easy to use or whether it is useful or both. We will use MS Excel and SPSS methods to analyze our quantitative data. Which analysis method is used depends on the sample size we obtain.

5.1.3.5 Recruitment

We try to recruit participants with a broad spectrum of demographic characteristics and experiences in software development and designing. Participants are recruited through posting recruitment notices (See Appendix B) to using personal contacts such as management personnel and other employees in organizations that develop software e.g. Intel and CA technologies.

5.1.3.6 Participants

In total, 26 participants responded to our survey. Out of which, 6 responses were filtered out as these were people who started the survey but did not finish it due to unknown reasons. We have discussed about incomplete survey responses too in our next chapter. We did not collect any demographic information about participants other than age and gender. There were 15 males and 5 females (Section 5.2) and the majority of them fall between 21 – 40 years of age.

5.2 Demographics of the Participants

In total 21 participants responded to our survey in meaningful way. Participants did not all continue to the end of the survey or answered a subset of questions, so different sections/questions of our survey were answered by different number of participants. Table 5.1 shows the demographics of all participants who responded to each of the sections in our survey.

Table 5.1. Demographics of the participants

Attribute	Variable	Total	Frequency
Age	21 – 25	3	14.29%
	26 – 30	8	38.1%
	31 – 40	8	38.1%
	41 - 50	2	9.52%
Gender	Male	16	76.19%
	Female	5	23.81%
Industry Experience	< a year	1	4.76%
	1 – 2	3	14.29%
	3 – 4	3	14.29%
	5 – 6	4	19.05%
	7 - 8	4	19.05%
	9 -10	4	19.05%
	11 -15	2	9.52%
Occupation	Software Engineer/developer	19	90.48%
	Software Engineer/designer	2	9.52%

Most of the participants fall between the age group 26 - 40. The majority of the population was software engineer/developer (90.48%) except for 2 participants (9.52%) who were software engineer/designer. The table shows that around 57.15% of the participants have 5 – 10 years of experience in the industry. There were 2 participants who had experience of 11-15 years. There were no participants that had experience greater than 15 years. Table 5.1 indicates that 76.19% of the total participants were males and 23.81% were female.

5.3 Perceived Usefulness and Perceived Ease of Use

In this section, we will discuss participants' evaluation of the PIP pattern in terms of its perceived usefulness and perceived ease of use. To understand the responses better and for reporting purpose, we divide the results sample into two: participants who executed the examples given in the survey and participants who did not execute the example. Another reason for the division is that it will help us to understand whether there is any change in the opinion about the usefulness of the pattern after executing the example.

For the survey, we provided explanations, diagrams, and code for two use cases to the users. The first example is a simple maintain user account use case of the banking application and the second example consists of a hospital use case: search patient information. Our first example uses masking for de-identification as a privacy service, and our second example uses k-anonymity for de-identification. In total 13 participants executed the examples. Out of these 13 participants, 12 participants claimed that they "agree" that they understood the pattern while 1 participant said that he/she 'strongly agrees' he/she understands the PIP pattern. While 7 participants did not execute the examples and most of them 'Somewhat' understand the pattern (2 = 'Strongly understand', 1 = 'Understand', 3 = 'Somewhat understand, 1 = 'Strongly do not understand'). This result indicates that executing the examples helped participants to understand the PIP pattern more clearly. Figure 5.3 shows the comparison of the between the understanding level of the participants who executed the examples and those who did not.

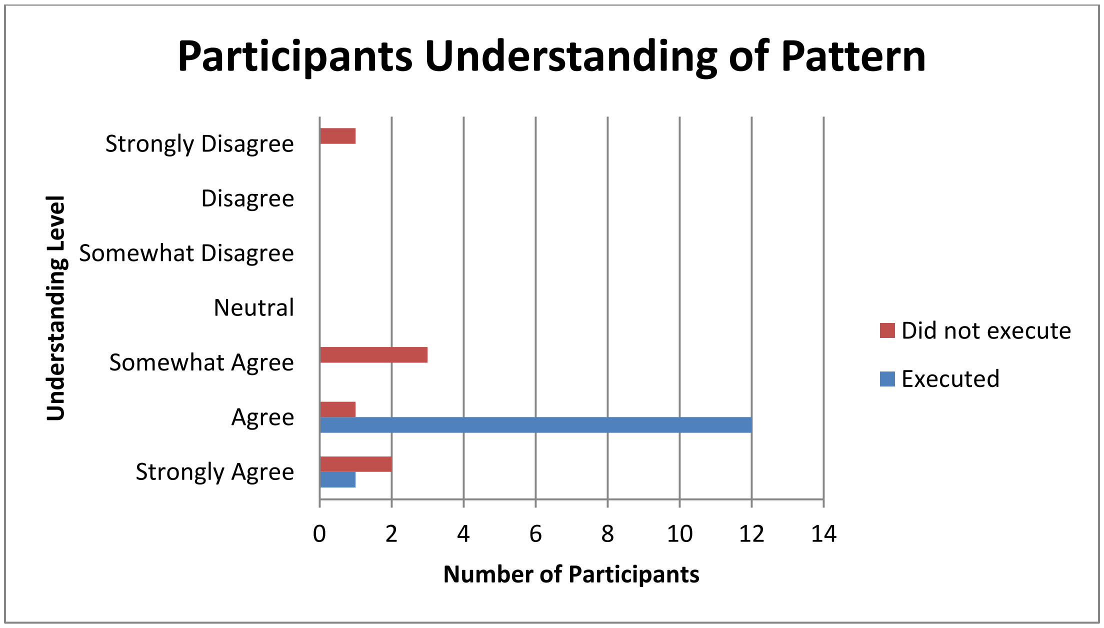


Figure 5.3. Participants understanding of the PIP pattern

On the basis of their understanding, participants fill out the survey to indicate whether the PIP pattern is useful and easy to use or if it is difficult and require lot of mental effort. There were 13 7-point Likert scale questions related to perceived usefulness and 14 questions for perceived ease of use.

5.3.1 Qualitative Analysis

Although the number of survey responses are not sufficient to provide scientifically significant statistical results, this section describes what would be done if there is more data, In the next sections, the thesis show how the researcher would analyze the responses from the perspective of reliability and confirming instrument validity for our context.

5.3.1.1 Reliability

Reliability is the degree to which measurements yield consistent results and are free from errors. In other words, it assess the internal consistency of the data that is, to what extent items are homogeneous to each other (Armentano, Christensen, & Schiaffino, 2015). One way to measure the reliability is Cronbach's alpha (α). Using our 7 response-survey data, Cronbach's alpha of perceived usefulness is 0.92, which shows good internal reliability in the sample data. Perceived ease of use was measured with fourteen items from Davis (Davis, 1989). Perceived ease of use showed good internal reliability with Cronbach's alpha of .85 as shown in the Table 5.2. The Cronbach's alpha (α) ranges from 0 to 1. The internal consistency of any item is said to be maximum when it is closer to 1. According to Masrom and Teknologi (2007), the criteria for acceptable internal consistency is 0.70 and above. According to the criteria, internal consistency of factors involved in the study has good internal consistency. The Cronbach's alpha of each group that is those who executed the examples and those who did not is mentioned in Table 5.3. The table shows that internal consistency of the data for those who executed the example applications is better than those who did not.

Table 5.2. Mean, Standard Deviation and Cronbach's Alpha for Variables

Variable	**Mean**	**Std. deviation**	**Cronbach's Alpha**
PU1	1.950	0.759	0.92
PU2	2.000	0.918	
PU3	1.800	0.894	
PU4	2.000	0.562	
PU5	2.050	1.276	
PU6	2.150	0.988	
PU7	2.200	0.768	
PU8	2.150	0.587	
PU9	2.400	1.142	
PU10	1.950	1.317	
PU11	2.150	1.348	
PU12	2.050	0.945	
PU13	2.150	1.387	
PEOU1	2.600	1.501	0.85
PEOU2	2.350	1.496	
PEOU3	2.900	1.553	
PEOU4	3.450	1.638	
PEOU5	3.350	2.033	
PEOU6	2.950	1.669	
PEOU7	2.350	1.137	
PEOU8	2.250	1.118	
PEOU9	2.650	1.599	
PEOU10	3.200	1.852	
PEOU11	2.400	1.231	
PEOU12	2.200	1.361	
PEOU13	2.050	1.050	
PEOU14	2.100	1.165	
Intention to Use	0.850	0.366	NA

Table 5.3. Cronbach's Alpha of Perceived Usefulness and Perceived Ease of Use

	All participants	**Participants who executed the examples**	**Participants who did not execute the examples**
Perceived Usefulness	0.92	0.93	0.90
Perceived Ease of Use	0.85	0.82	0.71

5.3.1.2 Validity

Validity is the degree to which variables within a single factor are correlated. We can analyze the validity by examining the factor loading. According to Hair, Black, Babin, and Anderson (2009), the recommended minimum threshold for samples size of 100 is 0.55. Although our sample size is limited but we will analyze the validity of the data. Before we analyze the correlation between variables and factors, we will perform the inter correlation between the items within a factor. Principal Component Analysis (PCA) is a statistical approach to analyze the correlation among the variables in the dataset. This will help us to group variables that are strongly related to a factor. We can eliminate the problematic questions in the survey that do not fit well with the variables they try to describe. We performed Principal Component Analysis with Varimax rotation to extract the factors from the survey questions that we need to analyze in Microsoft Excel using XLSTAT. Table 5.4 and Table 5.5 show the correlation matrix for the individual questions included in the experiments. According to the tables, it shows that there are some questions are not strongly correlated to other questions asked for the same factor. For example perceived usefulness survey question 3 (PU3) is not strongly correlated with perceived usefulness survey question 7 (PU7). This weak correlation can be because of the number of the responses received from the participants. So in order to improve our overall qualitative results, we remove a question from weakly correlated questions; for example, remove PU7 from relation PU3 and PU7. Red marked questions are removed from the experiment to improve the overall result of the experiment as these questions might affect our qualitative analysis result.

Table 5.4. Correlation Matrix for Perceived Usefulness

Variables	PU1	PU2	PU3	PU4	PU5	PU6	PU7	PU8	PU9	PU10	PU11	PU12	PU13
PU1	1												
PU2	0.438	1											
PU3	0.363	0.220	1										
PU4	0.307	0.275	0.119	1									
PU5	0.196	0.267	0.527	0.529	1								
PU6	0.380	0.241	0.382	0.467	0.734	1							
PU7	0.518	0.473	-0.006	0.193	0.103	0.161	1						
PU8	0.293	0.291	0.101	0.789	0.330	0.304	0.104	1					
PU9	0.355	0.219	0.290	0.781	0.583	0.569	0.136	0.658	1				
PU10	0.224	0.506	-0.036	0.285	-0.043	0.044	0.593	0.222	0.128	1			
PU11	0.615	0.357	0.459	0.595	0.533	0.545	0.459	0.576	0.601	0.477	1		
PU12	0.283	0.413	0.541	0.491	0.606	0.610	0.128	0.534	0.663	0.218	0.718	1	
PU13	0.445	0.522	0.474	0.595	0.614	0.482	0.248	0.506	0.636	0.390	0.778	0.743	1

Table 5.5. Correlation Matrix for Perceived Ease of Use

Variables	PEOU1	PEOU2	PEOU3	PEOU4	PEOU5	PEOU6	PEOU7	PEOU8	PEOU9	PEOU10	PEOU11	PEOU12	PEOU13	PEOU14
PEOU1	1													
PEOU2	0.768	1												
PEOU3	0.422	0.274	1											
PEOU4	0.266	0.399	0.456	1										
PEOU5	0.477	0.743	0.428	0.632	1									
PEOU6	-0.026	-0.088	-0.144	-0.151	0.058	1								
PEOU7	-0.087	0.073	-0.027	-0.076	0.152	0.556	1							
PEOU8	0.001	0.084	-0.277	-0.252	-0.054	0.480	0.524	1						
PEOU9	0.371	0.622	0.461	0.458	0.751	-0.346	-0.229	-0.238	1					
PEOU10	0.665	0.545	0.813	0.490	0.557	0.034	-0.094	-0.031	0.515	1				
PEOU11	0.184	0.201	-0.015	-0.368	0.180	0.587	0.486	0.324	0.020	0.170	1			
PEOU12	0.176	0.220	-0.075	-0.004	0.157	0.608	0.543	0.496	-0.101	0.118	0.580	1		
PEOU13	0.165	0.213	-0.262	-0.278	0.015	0.470	0.543	0.348	-0.237	-0.144	0.581	0.812	1	
PEOU14	0.251	0.050	-0.085	-0.323	-0.058	0.526	0.594	0.183	-0.304	-0.047	0.628	0.555	0.718	1

Factor loading represent how much a factor explains a variable or a question. Varimax (Kaiser, The varimax criterion for analytic rotation in factor analysis, 1958) rotation is a rotation method to change the coordinates of the individual items in such a way that each variable associate with at most one factor. They are split into disjoint sets as much as possible through Varimax rotation. Table 5.6 shows the factor loading of each individual item after performing Varimax rotation and Figure 5.4 shows it corresponding graph. According to the data, PU4, PU8, and PEOU4 are not strongly associated with their corresponding factor. Again the reason can be the limited number of responses involved in the experiment. To improve the further qualitative analysis result, we remove these three questions from the qualitative analysis.

Table 5.6. Factor loadings after Varimax rotation

	D1	D2	D3
PU1	0.627	-0.151	0.043
PU2	0.585	0.128	-0.125
PU3	0.719	-0.334	-0.140
PU4	**0.482**	**0.295**	**0.715**
PU5	0.704	0.392	0.133
PU6	0.714	0.314	0.072
PU8	**0.410**	**0.030**	**0.787**
PU9	0.640	0.134	0.559
PU11	0.768	0.036	0.381
PU12	0.816	-0.074	0.287
PU13	0.785	0.251	0.288
PEOU1	0.108	0.746	0.044
PEOU2	0.010	0.732	0.233
PEOU3	0.268	0.750	0.113
PEOU4	**-0.262**	**0.469**	**0.654**
PEOU5	-0.223	0.707	0.483
PEOU9	-0.207	0.767	0.108
PEOU10	0.169	0.820	0.257
Intention to Use	-0.216	-0.705	0.183

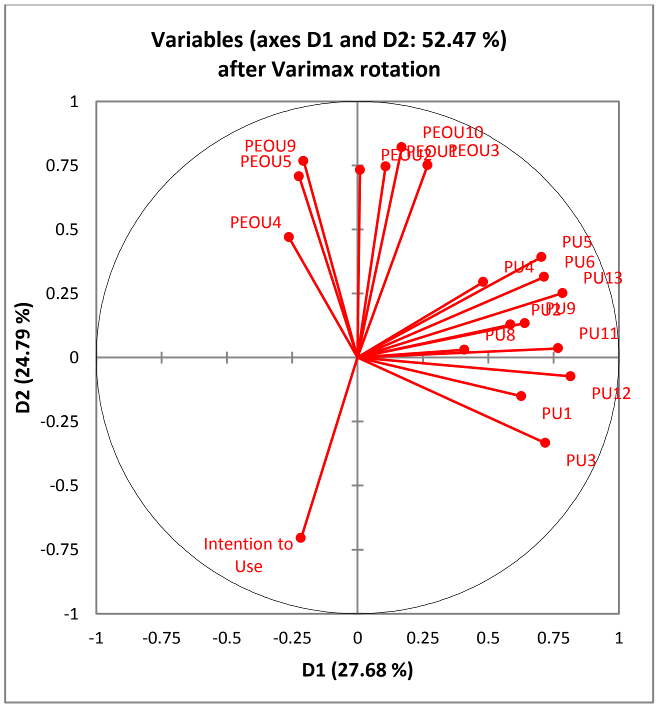


Figure 5.4. Correlation between Factors and Variables after Varimax

5.4 Results for Perceived Usefulness and Ease of Use

We consider all 27 questions (13 for perceived usefulness and 14 for perceived ease of use). In this section, we discuss how many responses we received for each factor. We will also compare the responses of the participants who executed the application and who did not execute the examples.

Table 5.7 shows the perception of the participants regarding usefulness and ease of use of the PIP pattern. Overall, the table indicates that participants think the PIP pattern is more useful than its ease of use. Besides, participants who executed the examples considered it more easy to use and useful comparatively to the participants who did not execute the examples.

Table 5.7. Average of Scale of Perceived Usefulness and Perceived Ease of Use

	All participants	**Participants who executed the examples**	**Participants who did not execute the examples**
Perceived Usefulness	2.077	1.956	2.142
Perceived Ease of Use	2.629	2.429	2.505

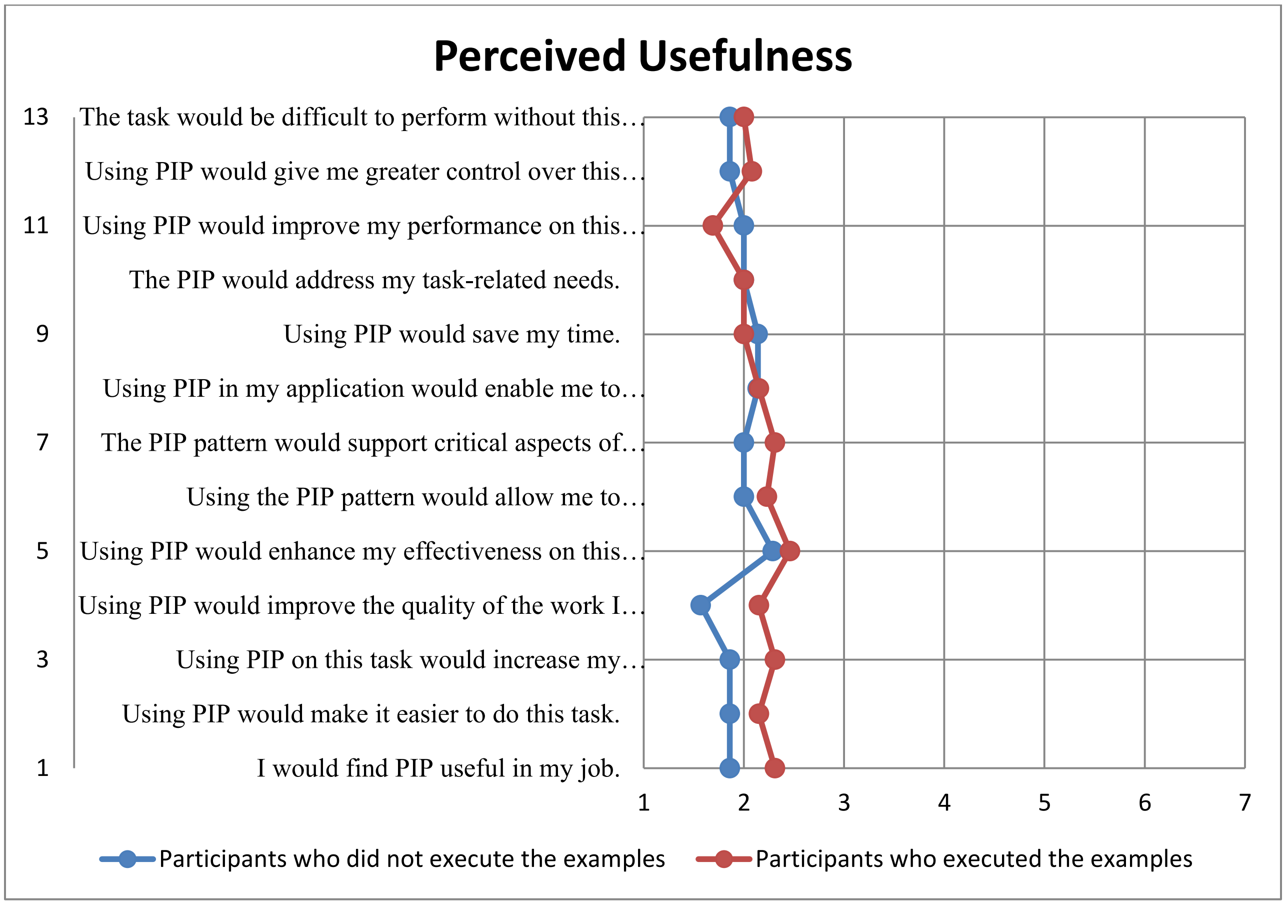


Figure 5.5. Participants Groups' Perceived Usefulness

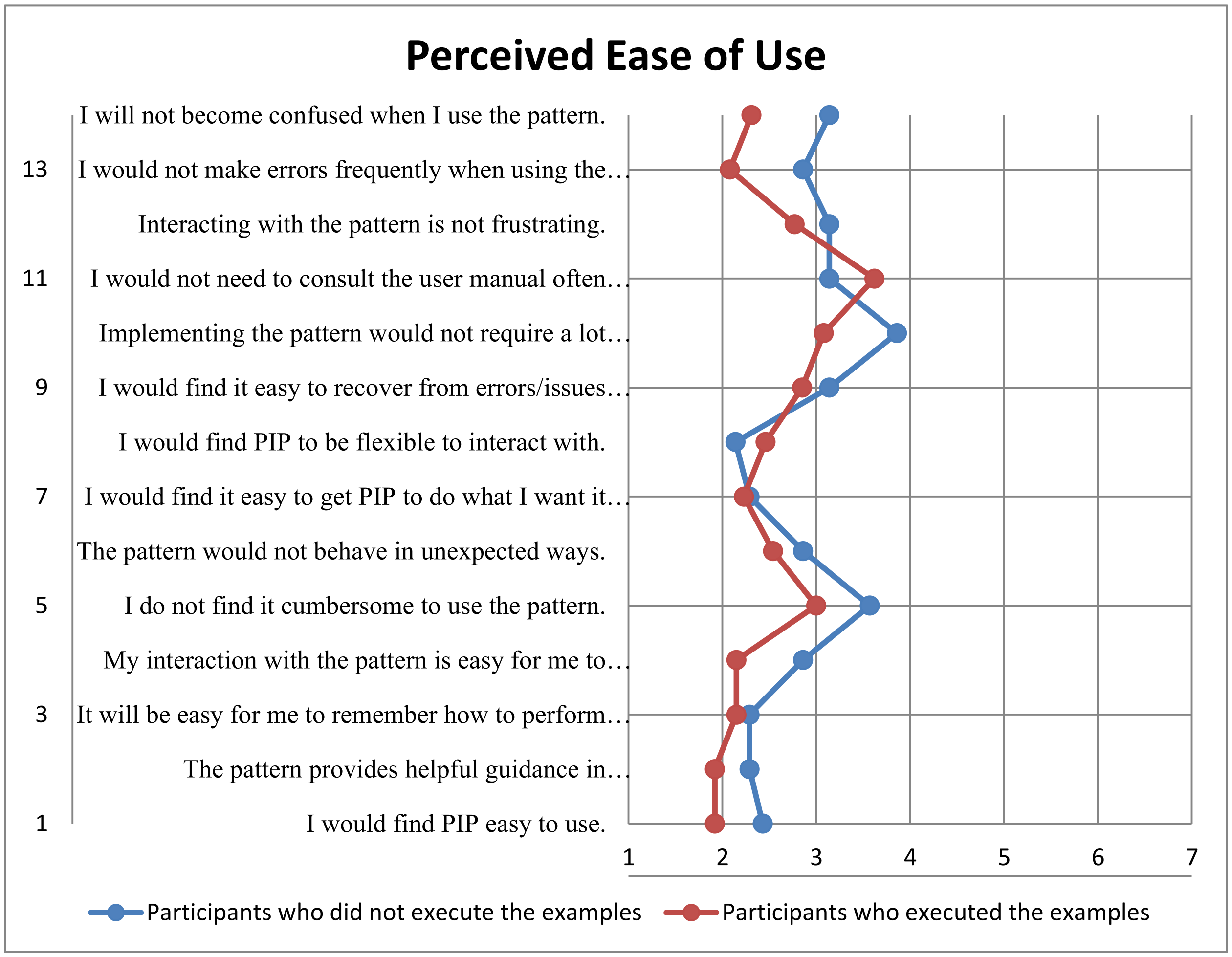


Figure 5.6. Participants Groups' Perceived Ease of Use

Figure 5.5 and Figure 5.6 show the comparison of the perceived usefulness and perceived ease of use of the two groups: those who executed the application and those who did not. The perceived usefulness of both the groups is somewhat same as indicated by Figure 5.5. Their variations are comparatively less than the variations in the responses of the group participants for perceived ease of use as shown in Figure 5.6. Overall, participants believe that the PIP pattern is more useful.

Figure 5.7 and Figure 5.8 shows the overall responses of the participants with considering w they have executed the applications or not. The figures show that overall more people (blue and red) consider the PIP pattern as useful and easy to use. Figure 5.7 shows that around 83% of the responses (strongly agree and agree) are in favor of usefulness of the PIP pattern. While 70% of the responses for perceived ease of use fall in strongly agree and agree category. In Figure 5.8, except for the item "*I would not need to consult the user manual often when using the pattern*." all items show that the PIP pattern is easy to use. Overall, again the PIP pattern is more useful than the efforts required to use it.

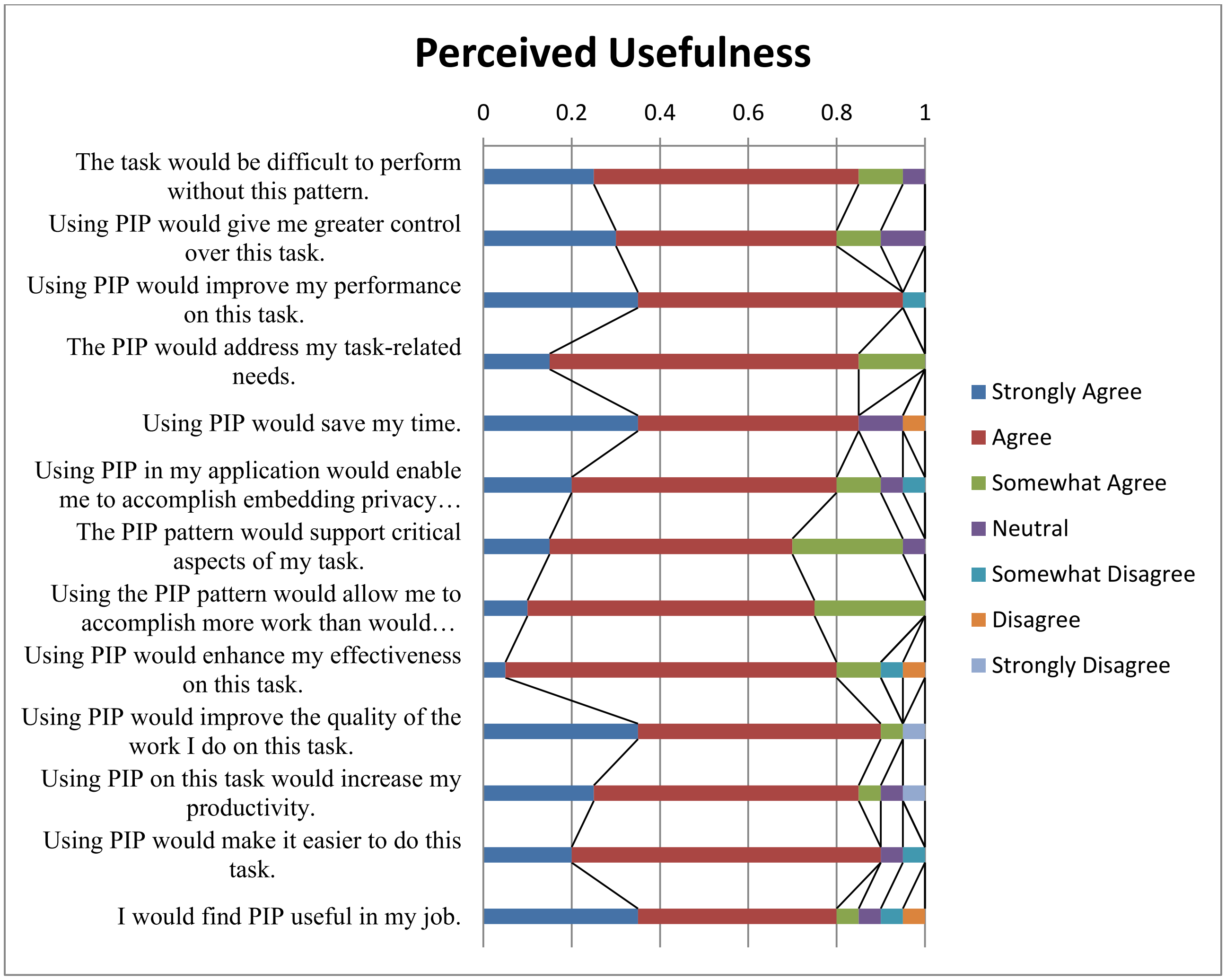


Figure 5.7. Overall Stacked Bar Chart of the Perceived Usefulness

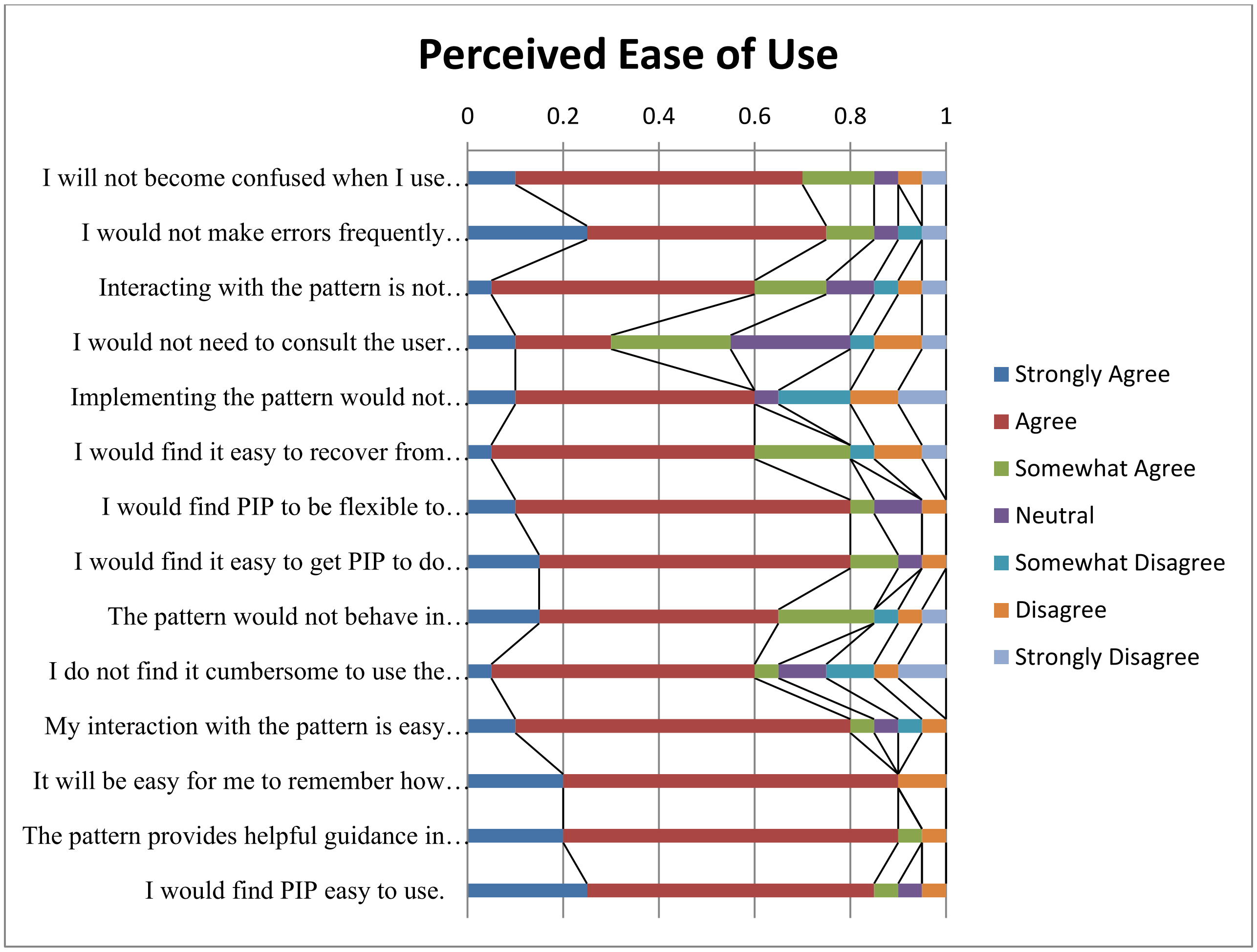


Figure 5.8. Overall Stacked Bar Chart of the Perceived Ease of Use

5.5 Improvements in PIP Pattern

In total 21 participants responded to our survey in a meaningful way. Out of 21 participants, 6 (28%) participants believe that there is a margin of improvement in the PIP pattern. Except for one participant we have not received any description regarding what improvement can be made in the PIP pattern to improvise it. According to the participant who responded to the question "*how we can improve this pattern*?" he mentioned that "*In the mock object, do privacy check before calling business functions. If the privacy checks fail, the mock object can fail fast that way*". This suggestion is a good suggestion; we can fail fast by checking the requesting user's permissions. But, in our examples privacy services are operating on actual data sets and these permissions cannot be verified until we fetch data from the data stores.

5.6 Overall Motivation to Use the Pattern

Out of 21 participants, 18 (85%) participants responded 'yes' to "*If you have to implement privacy in your application, would you use this pattern?*". Out of these 18 participants, 5 participants mentioned that we can improvise the PIP pattern even though they showed their intention to use the pattern.

	Intention to Use		
No	3	2	No Improvement and no intention to Use
		1	Improvement and no intention to Use
Yes	17	12	No Improvement and Intends to Use
		5	Improvement but Intends to Use

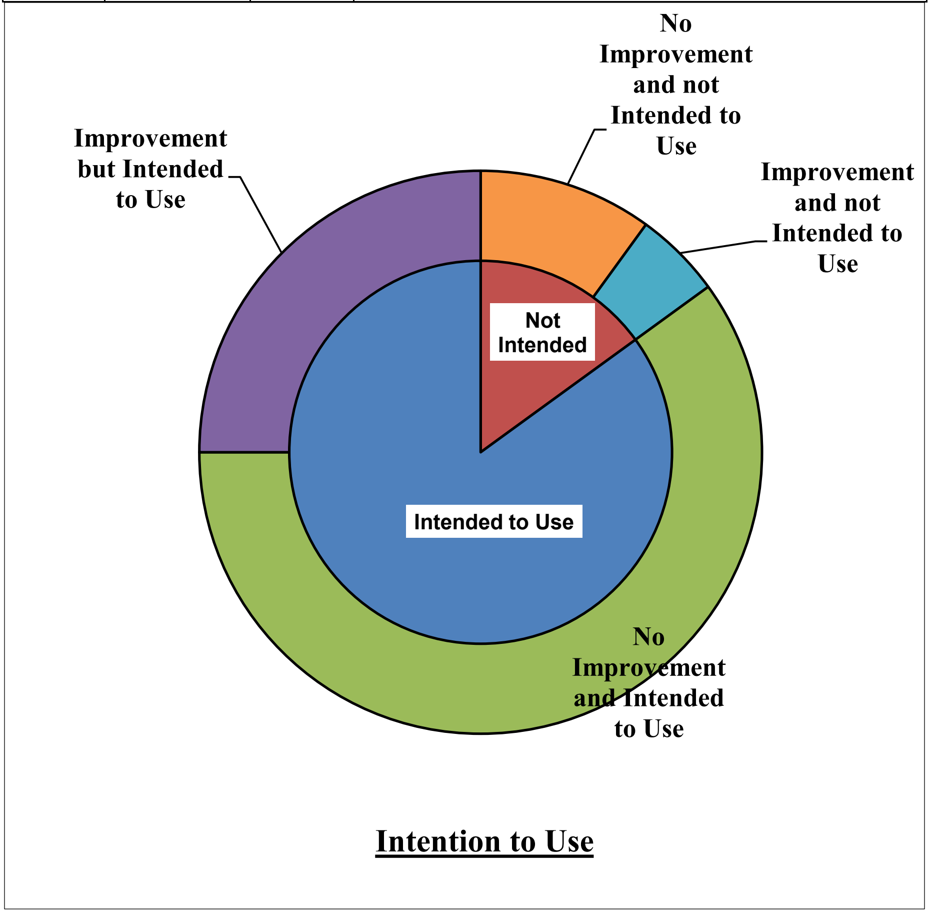


Figure 5.9. Participants' Intention to Use the PIP Pattern

5.7 Limitations

This study suffered from a number of limitations. The most profound limitation to this study was the size of the sample. Only 27 software professionals participated in the survey out of which only 21 surveys were completed. Generally, a small sample size makes the study less generalizable to the population of interest. Another limitation is that nearly all of them were software engineers. It would be really helpful if population will be from diverse IT background. Not only IT background, but geographic locations of the participants can also be a limitation of the study as most of the participants were from Canada. As other countries have their own distinctive policies regarding privacy, a more equal distribution of geographic location of participants may have had an impact on the findings and results. The effort of participants is another limitation. As there was no incentive to complete the survey, participants might have not given the online survey their full efforts and interest. The years of experience of the professionals are also another limitation of the study as on average participants had 6 – 7 years of experience. It would be helpful to conduct a survey from professionals who have experience in privacy engineering and are working in the industry for various amounts of years. Although, the findings of the small study favor our proposed PIP pattern, further studies are needed with more population and with different specialties. However, while the study provides preliminary results, these results provide a foundation for future studies.

Chapter 6 : Conclusions and Future Work

To improve software engineers' productivity, we describe a novel pattern for privacy pattern injection. To the best of our knowledge, a privacy super-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 Service Oriented Architecture (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 thesis, privacy can be incorporated in an existing system without modifying its code, or in some cases modifying the code to a very small extent. We evaluated our privacy injection pattern using de-identification in the thesis. For future work, we can evaluate our PIP pattern by injecting other privacy patterns.

We demonstrate our privacy injection pattern in a banking use case and a health care use case. We conducted an online survey and recruited broadly with the only criteria being that participants should have experience in software development and design and were interested in using and learning about our proposed master Privacy Injection Pattern (PIP). We used Davis's Technology Acceptance Model (TAM) model (Davis, 1989) in the survey to understand the acceptance of the PIP on the basis of the perceived usefulness (PU) and perceived ease of use (PEOU) of the software engineers. Our solution is based on three existing technologies; aspect oriented programming (AOP), dependency injection (DI), and mocking. This work can substantially reduce the designing and programming time for all those who will encounter such issue of injecting privacy in the existing system or even to those who want to use existing privacy services for new application. Furthermore, this work also allows software developers to quickly test the implemented privacy services by writing unit tests because the frameworks we have used for our patterns allows great flexibility for unit testing.

We illustrated the simplicity of the PIP implementation, which we believe will enhance its chances of adoption by software engineers. We chose 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". (Ali et al., 2015)

We conducted a user study with software engineers from large to small participating software organizations. Software engineers, internationally, were provided with guidance on using PIP and asked to evaluate the PIP using a validated Technology Acceptance Model (TAM) survey instrument. Our overall finding is that the PIP pattern has perceived benefits and is worth the efforts required to use it. According to our initial survey, 85% of the population agreed to use the PIP pattern when they encounter a situation where they need to incorporate privacy in existing application without modifying the underlying application. It is noteworthy that practicing, software engineers who took the time to execute the example application believed the PIP pattern to be more easy and useful comparatively to those who did not execute the examples.

There are numerous areas in which to conduct future research on incorporation of privacy patterns in the existing application. As indicated, in the previous chapter, our preliminary study on the PIP pattern can be taken forward to analyze the result with larger sample sizes and populations. Researchers can also include other factors (e.g. training by the organization, IT experience etc.) in analyzing the adoption of the PIP pattern. Proper quantitative analysis can be done in future research.

Researchers may hire software and privacy engineers from industry and ask them to implement the PIP pattern in existing applications. Researchers can then record engineers concerns and their inputs. This will provide a complete and fresh analysis on the proposed PIP pattern.

Bibliography

Aggarwal, C. C., & Yu, P. S. (2008). Privacy-Preserving Data Mining: Models and Algorithms. Springer.

Agrawal, R., & Srikant, R. (2000). Privacy-Preserving Data Mining. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (págs. 439-450). ACM.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Constructions.* Oxford University Press (Aug. 1 1977).

Ali, N., Jutla, D., & Bodorik, P. (2015). PIP: A (Privacy) Injection Pattern for Inserting Privacy Patterns and Services in Software. *Annual Privacy Forum*.

Almorsy, M., Grundy, J., & Ibrahim, A. S. (2012). VAM-aaS: Online Cloud Services Security Vulnerability Analysis and Mitigation-as-a-Service. *WISE'12 Proceedings of the 13th international conference on Web Information Systems Engineering* (págs. 411-425). Paphos, Cyprus: Springer-Verlag Berlin Heidelberg.

Armentano, M. G., Christensen, I., & Schiaffino, a. S. (2015). Applying the Technology Acceptance Model to Evaluation of Recommender Systems. *Polibits (51)*, 73 - 79.

ARX. (s.f.). *ARX - Powerful Data Anonymization*. Obtenido de http://arx.deidentifier.org/

Ashley, P., Hada, S., Karjoth, G., Powers, C., & Schunter, M. (2003). *Enterprise Privacy Authorization Language (EPAL 1.2).* IBM Research.

Barth, A., Mitchell, J. C., & Rosenstein, J. (2004). Conflict and Combination in Privacy Policy Languages. *WPES '04 Proceedings of the 2004 Workshop on Privacy in the Electronic Society* (págs. 45-46). ACM Press.

Basin, D., Doser, J., & Lodderstedt, T. (2006). Model Driven Security: from UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15 Issue 1*, 39-91.

Bender, J., & McWherter, J. (2011). *Professional test driven development with C#: developing real world applications with TDD.* Wrox Press Ltd.

Benenson, Z., Fort, M., Freiling, F., Kesdogan, D., & Penso, L. D. (2006). TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. En *Computer Security – ESORICS 2006* (págs. 34-48). Springer Berlin Heidelberg.

Beresford, A. R., Rice, A., Skehin, N., & Sohan, R. (2011). MockDroid: trading privacy for application functionality on smartphones. *HotMobile '11 Proceedings of the*

12th Workshop on Mobile Computing Systems and Applications (págs. 49-54). ACM New York, NY, USA.

Beresford, A., & Stajano, F. (2003). Location Privacy in Pervasive Computing. *Pervasive Computing, IEEE, Volume:2 ,Issue: 1*, 46-53.

Berghe, C. V., & Schunter, M. (2006). Privacy Injector — Automated Privacy Enforcement Through Aspects. *PET'06 Proceedings of the 6th international conference on Privacy Enhancing Technologies* (págs. 99-117). Springer-Verlag Berlin, Heidelberg.

Bier, C., & Krempel, E. (2012). Common privacy patterns in video surveillance and smart energy. *Computing and Convergence Technology (ICCCT), 2012 7th International Conference (page 610-615).*

Bodorik, P., & Jutla, D. (2008). Privacy with Web Serivces: Intelligence Gathering and Enforcement. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 546-549.

Bodorik, P., Jutla, D. N., & Brynn, A. (2014). Privacy Engineering with PAWS: Injecting RESTful Privacy Web Services. Submitted to IEEE Software, November 30, 2014.

Bodorik, P., Jutla, D. N., & Dhillon, I. (2009). Privacy compliance with Web Service. *Journal of Information Assurance and Security*, 412-421.

Bowen, W. (1986). The Puny Payoff from Office Computers. *Fortune*, 20-24.

Brown, P. F., Janssen, G., Jutla, D. N., Sabo, J., & Willett, M. (3 de July de 2013). Privacy Management Reference Model and Methodology (PMRM) Version 1.0. OASIS Committee Specification 01.

Cavoukian, A. (2013). *About PbD*. Recuperado el 05 de 01 de 2013, de Privacy by Design: http://www.privacybydesign.ca/index.php/about-pbd/

Cavoukian, A., & Khaled El Emam. (25 de June de 2014). *De-identification Protocols: Essential for Protecting Privacy.* Recuperado el 30 de November de 2014, de http://www.privacybydesign.ca/content/uploads/2014/09/pbd-de-identifcation-essential.pdf

Cavoukian, A., Carter, F., Jutla, D., Sabo, J., Dawson, F., Fieten, S., . . . Finneran, T. (25 de June de 2014). *Annex Guide to Privacy by Design Documentation for Software Engineers Version 1.0 Committee Note Draft 01.* Recuperado el 30 de November de 2014, de http://docs.oasis-open.org/pbd-se/pbd-se-annex/v1.0/cnd01/pbd-se-annex-v1.0-cnd01.pdf

Cavoukian, A., Shapiro, S., & Cronk, R. J. (January de 2014). *Privacy Engineering: Proactively Embedding Privacy, by Design.* Obtenido de https://www.privacybydesign.ca/content/uploads/2014/01/pbd-priv-engineering.pdf

Ceccato, M., & Tonella, P. (2004). Measuring the Effects of Software Aspectization. *1st Workshop on Aspect Reverse Engineering*.

Champatiray, C. (6 de June de 2014). *Dependency Inversion Principle, IoC Container, and Dependency Injection (Part - 1)*. Obtenido de CodeProject: http://www.codeproject.com/Articles/465173/Dependency-Inversion-Principle-IoC-Container-and-D

Chen, K., & Wang, D.-W. (2007). An Aspect-Oriented Approach to Privacy-Aware Access Control. *Proceedings of the Sixth International Conference on Machine Learning and Cybernetics, Volume 5* (págs. 3016 - 3021). Hong Kong: IEEE.

Cranor, L. F. (2002). *Web Privacy with P3P.* O'Reilly Media.

Cranor, L., Langheinrich, M., Marchiori, M., & Reagle, J. (April de 2002). *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, W3C Recommendation*. Obtenido de http://www.w3.org/TR/P3P/

Culp, A. (4 de May de 2015). *The Dependency Injection Design Pattern*. Obtenido de MSDN:https://msdn.microsoft.com/en-us/library/vstudio/hh323705(v=vs.100).aspx

Davis, F. D. (1989). Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly Vol. 13, No. 3, Management Information Systems Research Center, University of Minnesota*, 319-340.

Davis, F. D., Bagozzi, R. P., & Warshaw, P. R. (1989). User acceptance of computer technology: a comparison of two theoretical models. *Management Science, Volume 35 Issue 8*, 982 - 1003.

Deiters, M. (September de 2005). *Aspect-Oriented Programming*. Obtenido de MSDN: https://msdn.microsoft.com/en-us/library/aa288717%28v=vs.71%29.aspx

Doty, N., & Gupta, M. (2013). Privacy Design Patterns and Anti-Patterns. Patterns Misapplied and Unintended Consequences. *Trustbusters Workshop at the Symposium on Usable Privacy and Security.*

Federal Trade Commission. (s.f.). *About the FTC*. Obtenido de Federal Trade Commission: https://www.ftc.gov/about-ftc

Finn, R. L., Wright, D., & Friedewald, M. (2013). *Seven Types of Privacy*. Obtenido de SelectedWorks of Michael Friedewald: http://works.bepress.com/michael_friedewald/60

Fishbein, M., & Ajzen, I. (1975). *Belief, Attitude, Intention and Behavior: An Introduction to Theory and Research.* Addison- Wesley.

Fowler, M. (23 de January de 2004). *Inversion of Control Containers and the Dependency Injection pattern*. Obtenido de Martin Fowler: http://martinfowler.com/articles/injection.html

Fowler, M. (2 de January de 2007). *Mocks Aren't Stubs*. Obtenido de Martin Fowler: http://martinfowler.com/articles/mocksArentStubs.html

Ghazinour, K., & Barker, K. (2009). A Lattice-based Privacy Aware Access Control Model. *International Conference on Computational Science and Engineering, 2009. CSE '09, Volume:3* (págs. 154-159). Vancouver, BC: IEEE.

Gong, L., Mueller, M., Prafullchandra, H., & Schemers, R. (2007). Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit. *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (págs. 103-112). Monterey, California.

Goold, B. J. (2009). Surveillance and the Polictical Value of Privacy. *Amsterdam Law Forum - VU University Amsterdam, Vol. 1, No. 4*.

Groves, M. D. (2013). *AOP in .NET Practical Aspect-Oriented Programming.* Manning Publications Co.

Guarda, P., & Zannone, N. (2009). Towards the Development of Privacy-Aware Systems. *Information and Software Technology Volume 51 Issue 2*, 337-350.

Gutwirth, S., Leenes, R., & de Hert, P. (2015). *Reforming European Data Protection Law.* Springer.

Hafiz, M. (2006). A Collection of Privacy Design Patterns. *PLoP '06 Proceedings of the 2006 conference on Pattern languages of programs, Article No. 7.* ACM New York, NY, USA.

Hair, J. F., Black, W. C., Babin, B. J., & Anderson, R. E. (2009). *Multivariate Data Analysis.* Prentice Hall; 7 edition (Feb. 13 2009).

Haque, H. (12 de March de 2013). *A curry of Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) and IoC Container*. Obtenido de Code Project: http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve

Hartwick, J., & Barki, H. (April 1994). Explaining the role of user participation in information. *Management Science*, 440-465.

He, Q., & Antón, A. I. (2003). A Framework for Modeling Privacy Requirements in Role Engineering. *International Workshop on Requirements Engineering for Software Quality (REFSQ 2003).* Klagenfurt / Velden, Austria.

Hoepman, J.-H. (2014). Privacy Design Strategies. *Proceedings in IFIP Advances in Information and Communication Technology. 9th IFIP TC 11 International Conference, Volume 428*, (págs. 446-459). Marrakech, Morocco.

Holmes, N. (25 de September de 2008). *Canada's Federal Privacy Laws.* Obtenido de Library of Parliament: http://www.parl.gc.ca/Content/LOP/researchpublications/prb0744-e.pdf

Hornyack, P., Han, S., Jung, J., Schechter, S., & Wetherall, D. (2011). "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. *CCS '11 Proceedings of the 18th ACM Conference on Computer and Communications Security* (págs. 639-652). Chicago, IL: ACM, New York, NY, USA.

IKVM.NET. (August de 2015). Obtenido de http://www.ikvm.net/

Jeroen van Rest, Daniel Boonstra, Maarten Everts, Martin van Rijn, & Ron van Paassen. (2014). Designing Privacy-by-Design. En *Privacy Technologies and Policy, LNCS 8319* (págs. 55-72). Springer Berlin Heidelberg.

Ježek, K., Holý, L., & Brada, P. (2012). Dependency Injection Refined by Extra-functional Properties. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2012* (págs. 255-256). Innsbruck: IEEE.

Johnson, R. E., & Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 22-35.

Jürjens, J. (2005). *Secure Systems Development with UML.* Springer-Verlag Berlin Heidelberg.

Jutla, D. N., Bodorik, P., & Ali, S. (2013). Engineering Privacy for Big Data Apps with the Unified Modeling Language. *IEEE International Congress on Big Data*, 38-45.

Kaindl, H. (2000). A Design Process Based on a Model Combining Scenarios with Goals and Functions. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on (Volume:30 ,Issue: 5)*, 537-551.

Kaiser, H. F. (1958). The varimax criterion for analytic rotation in factor analysis. *Psychometrika, vol. 23 issue 3*, 187 - 200.

Kalloniatis, C., Kavakli, E., & Gritzalis, S. (2007). Using Privacy Process Patterns for Incorporating Privacy Requirements into the System Design Process. *The Second International Conference on Availability, Reliability and Security, ARES 2007 (page 1009--1017).*

Karjoth, G., & Schunter, M. (2002). A Privacy Policy Model for Enterprises. *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE* (págs. 271-281). IEEE.

Karjoth, G., Schunter, M., & Waidner, M. (2002). The platform for enterprise privacy practices: privacy-enabled management of customer data. *PET'02 Proceedings of the 2nd international conference on Privacy enhancing technologies* (págs. 69-84). Springer-Verlag Berlin, Heidelberg.

Kenny, S., & Borking, J. (2002). The Value of Privacy Engineering. *Journal of Information, Law and Technology*.

King, W. R., & He, J. (2006). A meta-analysis of the technology acceptance model. *Information & Management, Volume 43, Issue 6*, 740-755.

Laddad., R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co.

Li, N., Li, T., & Venkatasubramanian, S. (2007). t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. *IEEE 23rd International Conference on Data Engineering, ICDE.* (págs. 106 - 115). Istanbul: IEEE.

Livne, O. E., Schultz, N. D., & Narus, S. P. (2011). Federated Querying Architecture with Clinical & Translational Health IT Application. *Journal of Medical Systems, Volume 35, Issue 5*, 1211-24.

Ma, Q., & Liu, L. (2004). The Technology Acceptance Model: A Meta-Analysis of Empirical Findings. *Journal of Organizational and End User Computing*, 59 - 72.

Machanavajjhala, A., Gehrke, J., Kifer, D., & Venkitasubramaniam, M. (2007). l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD), Volume 1 Issue 1*.

Mackinnon, T., Freeman, S., & Craig, P. (2001). Endo-testing: unit testing with mock objects. *Extreme programming examined*, 287-301.

Masrom, M., & Teknologi, U. (2007). Technology Acceptance Model and E-learning. *12th International Conference on Education, Sultan Hassanal Bolkiah Institute of Education , Universiti Brunei Darussalam*, (págs. 1 - 10).

Masuda, G., Sakamoto, N., & Ushijima, K. (1999). Evaluation and Analysis of Applying Design Patterns. *International Workshop on Principles of Software Evolution - IWPSE.*

Md. Moniruzzaman, Ferdous, M., & Hossain, R. (2010). A Study of privacy policy enforcement in access control models. *13th International Conference on Computer and Information Technology (ICCIT), 2010* (págs. 352-357). Dhaka: IEEE.

Mourad, A., Laverdière, M.-A., & Debbabi, M. (2008). An aspect-oriented approach for the systematic security hardening of code. *Computers & Security, Volume 27, Issues 3–4*, 101–114.

Narayanan, A., & Shmatikov, V. (2008). Robust de-anonymization of large sparse datasets. *Proceedings of the 2008 IEEE Symposium on Security and Privacy.*

Pietraszek, T., & Berghe, C. V. (2005). Defending against injection attacks through context-sensitive string evaluation. *In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID2005)* (págs. 124–145). Springer-Verlag Berlin, Heidelberg.

Pitofsky, R., Anthony, S., Thompson, M., Swindle, O., & Leary, T. (2000). *Privacy Online: Fair Information Practices in the Electronic Marketplace, A Report to Congress.* US Federal Trade Commission.

Porekar, J., Jerman-Blazic, A., & Klobucar, T. (2008). Towards Organizational Privacy Patterns. *Digital Society, 2008 Second International Conference (page 15-19).*

Prasanna, D. (2009). *Dependency Injection.* Manning Publications Co.

Raghunathan, B. (2013). *Complete Book of Data Anonymization From Planning to Implementation.* CRC Press Taylor & Francis Group.

Rainie, L., Kiesler, S., Kang, R., & Madden, M. (5 de September de 2013). *Anonymity, Privacy, and Security Online*. Obtenido de Pew Research Center Internet, Science & Tech: http://www.pewinternet.org/files/old-media//Files/Reports/2013/PIP_AnonymityOnline_090513.pdf

Rizvi, S. J., & Haritsa, J. R. (2002). Maintaining Data Privacy in Association Rule Mining. *Proceedings of the 28th international conference on Very Large Data Bases, VLDB*, (págs. 682-693).

Rodríguez, A., Piattini, M., & Fernández-Medina, E. (2006). Security Requirement with a UML 2.0 Profile. *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on* (pág. 8). IEEE.

Romanosky, S., Acquisti, A., Hong, J., Cranor, L. F., & Friedman, B. (2006). Privacy patterns for online interactions. *Proceedings of the 2006 conference on Pattern languages of programs (page 1--9).* Portland, Oregon.

Roy, O. (2014). *The art of unit testing, 2nd edition.* Shelter Island, NY: Manning.

Rubner, Y., Tomasi, C., & Guibas, L. J. (2000). The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision, Volume 40, Issue 2*, 99-121.

Sabo, J., Willett, M., Brown, P., & Jutla, D. (26 de March de 2012). Privacy Management Reference Model and Methodology, OASIS PMRM TC Standards Track Committee Draft.

Sabo, J., Willett, M., Brown, P., & Jutla, D. (25 de March de 2013). Privacy Management Reference Model and Methodology, OASIS PMRM TC Standards Track Committee Specification.

Samarati, P., & Sweeney, L. (1998). Protecting Privacy when Disclosing Information: k-Anonymity and Its Enforcement through Generalization and Suppression. *IEEE Symp. on Security and Privacy*.

Scheffler, T., Geiß, S., & Schnor, B. (2008). An implementation of a privacy enforcement scheme based on the Java security framework using XACML policies. *Proceedings of the IFIP TC 11, 23rd International Information Security Conference, Volume 278* (págs. 157–171). Boston: Springer.

Scheffler, T., Schindler, S., & Schnor, B. (2012). Enforcing Location Privacy Policies through an AOP-based Reference Monitor. *World Congress on Internet Security (WorldCIS-2012)* (págs. 51 - 56). IEEE.

Schepers, J., & Wetzels, M. (2007). A meta-analysis of the technology acceptance model: Investigating subjective norm and moderation effects. *Information and Management 44* (págs. 90 - 103). Elsevier.

Seemann. (October de 2004). Mock Objects to the Rescue! Test Your .NET Code with NMock. *MSDN Magazine*. Obtenido de https://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S2

Seemann, M. (2012). *Dependency injection in. NET.* Manning.

Shapiro, S. S. (2011). Separating the Baby from the Bathwater - Toward a Generic and Practical Framework for Anonymization. *IEEE*.

Sharma, N., Batra, U., & Mukherjee, S. (2014). Enhancing Security in Service Oriented Architecture driven EAI using Aspect Oriented Programming in healthcare IT. *INTERNATIONAL JOURNAL OF SCIENTIFIC & ENGINEERING RESEARCH, VOLUME 5, ISSUE 3*, 50-53.

Sommerville, I. (2011). *Software Engineering 9*. Boston: Addison-Wesley.

Sonnino, B. (February de 2014). *Aspect-Oriented Programming with the RealProxy Class*. Obtenido de MSDN Magazine: https://msdn.microsoft.com/en-us/magazine/dn574804.aspx

Spiekermann, S., & Cranor, L. F. (2009). Engineering Privacy. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 67-82.

Sun, W., France, R., & Ray, I. (2011). Rigorous Analysis of UML Access Control Policy Models. *Policies for Distributed Systems and Networks (POLICY)*, 9-16.

Sweeney, L. (2002). k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10(5)*, 557-570.

Tillmann, N., & Schulte, W. (2006). Mock-Object generation with behavior. *ASE '06 Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (págs. 365-368). IEEE Computer Society Washington, DC, USA.

Tumer, A., Dogac, A., & Toroslu, I. H. (2003). A semantic-based user privacy protection framework for web services. *ITWP'03 Proceedings of the 2003 international conference on Intelligent Techniques for Web Personalization* (págs. 289-305). Springer-Verlag Berlin, Heidelberg.

Warren, S. D., & Brandeis, L. D. (15 de December de 1890). The Right to Privacy. *Harvard Law Review 4 (5)*, págs. 193-220.

Wenning, R. (2007). *Platform for Privacy Preferences (P3P) Project*. Recuperado el 15 de 10 de 2012, de http://www.w3.org/P3P/

Win, B. D., Joosen, W., & Piessens, F. (2002). Developing secure applications through aspect-oriented programming. En *Aspect-Oriented Software Development* (págs. 633-650). Addison-Wesley.

Yu, W., & Murthy, S. (2007). PPMLP: A Special Modeling Language Processor for Privacy Policies. *Computers and Communications 12th IEEE Symposium*, 851-858.

Zhong, S., Yang, Z., & Wright, R. N. (2005). Privacy-Enhancing k-Anonymization of Customer Data. *PODS '05 Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (págs. 139-147). ACM New York, NY, USA.

Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011). Taming Information-Stealing Smartphone Applications (on Android). *TRUST'11 Proceedings of the 4th international conference on Trust and trustworthy computing* (págs. 93-107). Pittsburgh, PA: Springer-Verlag Berlin, Heidelberg.

Zhu, Z. J., & Zulkernine, M. (2009). A model-based aspect-oriented framework for building intrusion-aware software systems. *Information and Software Technology, Volume 51, Issue 5*, 865–875.

Appendix A – Letter of Approval

Social Sciences & Humanities Research Ethics Board

Letter of Approval

July 23, 2015

Ms Naureen Ali

Computer Science\Computer Science

Dear Naureen,

REB #: 2015-3610
Project Title: PIP: A (Privacy) Injection Pattern for Inserting Privacy Patterns and Services in Software

Effective Date: July 23, 2015
Expiry Date: July 23, 2016

The Social Sciences & Humanities Research Ethics Board has reviewed your application for research involving humans and found the proposed research to be in accordance with the Tri-Council Policy Statement on *Ethical Conduct for Research Involving Humans.* This approval will be in effect for 12 months as indicated above. This approval is subject to the conditions listed below which constitute your on-going responsibilities with respect to the ethical conduct of this research.

Sincerely,

[redacted]

Dr. Karen Beazley, Chair

Appendix B – Recruitment Script

We invite you to take part in an on-line survey that will ask you to evaluate a new software pattern, called PIP or Privacy Injection Pattern. The purpose of the PIP is to inject privacy services or privacy patterns in existing systems without modifying it or if required, modify it to small extent. We want to learn your opinions of the benefits and ease of use of the PIP, and to understand how we may improve this pattern. The survey is conducted under the supervision of Dr. Dawn Jutla and Dr. Peter Bodorik.

We invite all software professionals who are interested in using and learning about the privacy injection pattern in systems to take the survey. We particularly want feedback from those participants who have experience with software development, software design, or privacy engineering.

Before starting with the survey you will be presented with a consent form. Once you click on the `Agree` button, you will be directed to the survey questions. The survey should take about 30-45 minutes.

There is no compensation for taking part in this research. As it is an online survey, a researcher will always be available through e-mail or phone to answer any questions you may have or address any problems that you may experience while performing the survey. If you have any questions, please contact Naureen Ali by email: Naureen@cs.dal.ca.

The survey is located at: (URL: https://surveys.dal.ca/opinio/s?s=29098)

Thank You

Naureen Ali
Student, Faculty of Computer Science, Dalhousie University,
Halifax, NS

Appendix C – Informed Consent

INSTRUCTION: Please read the following consent form carefully before clicking on the 'Agree' button.

PIP: A (PRIVACY) INJECTION PATTERN FOR INSERTING PRIVACY PATTERNS AND SERVICES IN SOFTWARE

Principal Investigators: Naureen Ali, MCSc Student, Faculty of Computer Science
Other Researchers (Supervisors): Dawn Jutla, Professor, Dept. of Finance, Information Systems, and Mgmt. Science, Saint Mary's University
Peter Bodorik, Professor, Faculty of Computer Science, Dalhousie University.

Contact Person: Naureen Ali, MCSc Student, Faculty of Computer Science, naureen@cs.dal.ca, 902-412-4980

Introduction

We invite you to take part in a survey of a software-based Privacy Injection Pattern conducted by Naureen Ali who is a graduate MCSc student at Dalhousie University. Your participation in this survey is voluntary; there is no compensation for participating in this survey. Neither your academic nor your employment performance evaluation will be affected by whether or not you participate. The survey is described below. This description tells you about the risks, inconvenience, or discomfort which you might experience. Participating in the study might not benefit you directly, but you may learn things that will benefit you. You may discuss any questions you have about this study with Naureen Ali at any time through e-mail or phone (before, during or after the study).

Purpose

The purpose of this survey is to understand the perceived benefits of our proposed privacy injection pattern (PIP). Specifically, is the pattern useful and/or easy to use to inject privacy in existing and new software applications? The knowledge gained from our survey will be in the results of the independent evaluation of our proposed pattern by practitioners in the software engineering field. The results may also provide new insights for better designing patterns to inject privacy after deployment.

Study Design

Before starting with the survey you will be shown a consent form online. Once you click on the 'agreed' button, you will be directed to the survey. The survey should take about 30-45 minutes. At the end of the survey, you will be asked to provide your e-mail id (optional), if you wish to receive the copy of study findings. As it is an online survey, a researcher will always be available through e-mail or phone to answer any questions you may have or address any problems that you may experience while performing the survey.

Who can Participate in the Survey?

For the study, the target population will be software engineers in general, as well as any technical personnel in charge of privacy. The potential participants must have basic knowledge of software development. It doesn’t matter if recruited participants have implemented privacy in their applications before or not.

Possible Risks and Discomforts

Some participants may get bored while answering the survey questions especially for those who have no information about the technologies being used in the proposed pattern such as aspect-oriented programming, dependency injection and mocking framework. To maintain participants’ interest we have provided the basic definitions of these technologies in the survey. The only identifiable information is the optional email addresses at the end of the survey for receiving a copy of study findings, and those will not be linked to the survey responses and will be stored separately. Since it is an online survey, the researchers will not be physically available to monitor participants; however, participants can contact researchers by email or phone if they have questions or technical difficulties.

Possible Benefits

There are no direct benefits for participants taking part in this survey, aside from the opportunity of becoming aware of the technologies used in the proposed pattern and also its use in example use cases provided with the survey. Indirect benefits include provision of help in advancing software design patterns to incorporate privacy in existing applications, an opportunity to be exposed to new research questions, and to contribute to research that may benefit others.

Anonymity and Confidentiality

The only personally identifiable information we are collecting are the **optional** email addresses for respondents to receive a copy of the study's findings. Email addresses will not be linked to the survey responses. All research data will be kept confidential and in a secure location. The data will be retained in encrypted format using best practices under Dalhousie University's data management guidelines. After five years the data and documents will be destroyed.

Use of Quotations

Your responses to free-form questions may be quoted in the final report. There will be no attribution of the quote beyond descriptive characteristics (e.g., one participant who does not currently use such systems stated "____").

Provision of Results

If you would like to receive a copy of study findings when published, please provide your email address at the end of the questionnaire or email Dr. Dawn Jutla (dawn.jutla@gmail.com) with your contact information.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: catherine.connors@dal.ca.

Appendix D – Signature Page

Signature Page

Project Title: PIP: A (Privacy) Injection Pattern for Inserting Privacy Patterns and Services in Software

Lead Researcher: Naureen Ali, Faculty of Computer Science, naureen@cs.dal.ca, 902-412-4980

I have read the explanation about this survey. I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time.

- Agree
- Disagree

I agree that the researchers may quote my responses to free-form questions.

- Agree
- Disagree

Appendix E – Survey Questions

PART –A

Survey: PIP - Privacy Injection Pattern

This is an online-survey which has questions on our proposed Privacy Injection Pattern (PIP). Our proposed pattern may help in injecting privacy into existing applications without modifying the existing modules or modifying it to a small extent. The purpose for conducting this survey is to obtain your evaluation of the proposed software pattern.

1. How old are you?

   ☐ below 21
   ☐ 21 - 25
   ☐ 26 - 30
   ☐ 31 - 40
   ☐ 41 - 50
   ☐ 51 - 60
   ☐ above 60

2. What is your gender?

   ☐ Male
   ☐ Female

3. How long have you been in software (development and designing) industry?

   ☐ less than a year
   ☐ 1 – 2 years
   ☐ 3 – 4 years
   ☐ 5 – 6 years
   ☐ 7 – 8 years
   ☐ 9 – 10 years
   ☐ 11 – 15 years

☐ 16 – 20 years

☐ more than 20 years

4. What is your current designation?

☐ Software engineer/developer

☐ Software designer

☐ Project manager

☐ Privacy manager

☐ Quality assurance

☐ Software support

☐ Other (please specify)________________

PART – B

Increasingly, software engineers are looking for repeatable ways to embed privacy in their code. We propose the concept of a master privacy injection pattern (PIP) for software engineers to use to automate dynamically "injecting" existing privacy patterns in existing or new code. PIP is composed of a novel tri-abstraction combination of aspect-oriented programming (AOP), dependency injection (DI), and mocking.

Software Engineering Technique	Terminology and Traditional Uses
Aspect-Oriented Programming (AOP)	Aspect-oriented programming (AOP) is a programming technique to separate crosscutting concerns, such as privacy, in units 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 a reusable library which 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 objects 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 the flow of control of an application to a container or a class (Prasanna, 2009) Dependency injection is mostly used for loosely coupled designs. It is commonly used for unit testing and validation/exception management (Culp, 2015).

Combining the three abstractions, we develop a new privacy injection pattern to insert known privacy patterns or services in new and existing legacy applications. Figure 0.1 shows our proposed Privacy Injection Pattern (PIP) 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 11) through one pattern instance. The concepts intrinsic to the PIP (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.

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 so that aspects can be used across software implementation classes. When using the 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 Figure 0.1), it basically calls the mock BAL object function (3). This mock object fetches data from the business layer as normal (4). We use the mock object to apply third-party privacy aspects from privacy data patterns (7) and to transfer the modified data to the presentation layer (11).

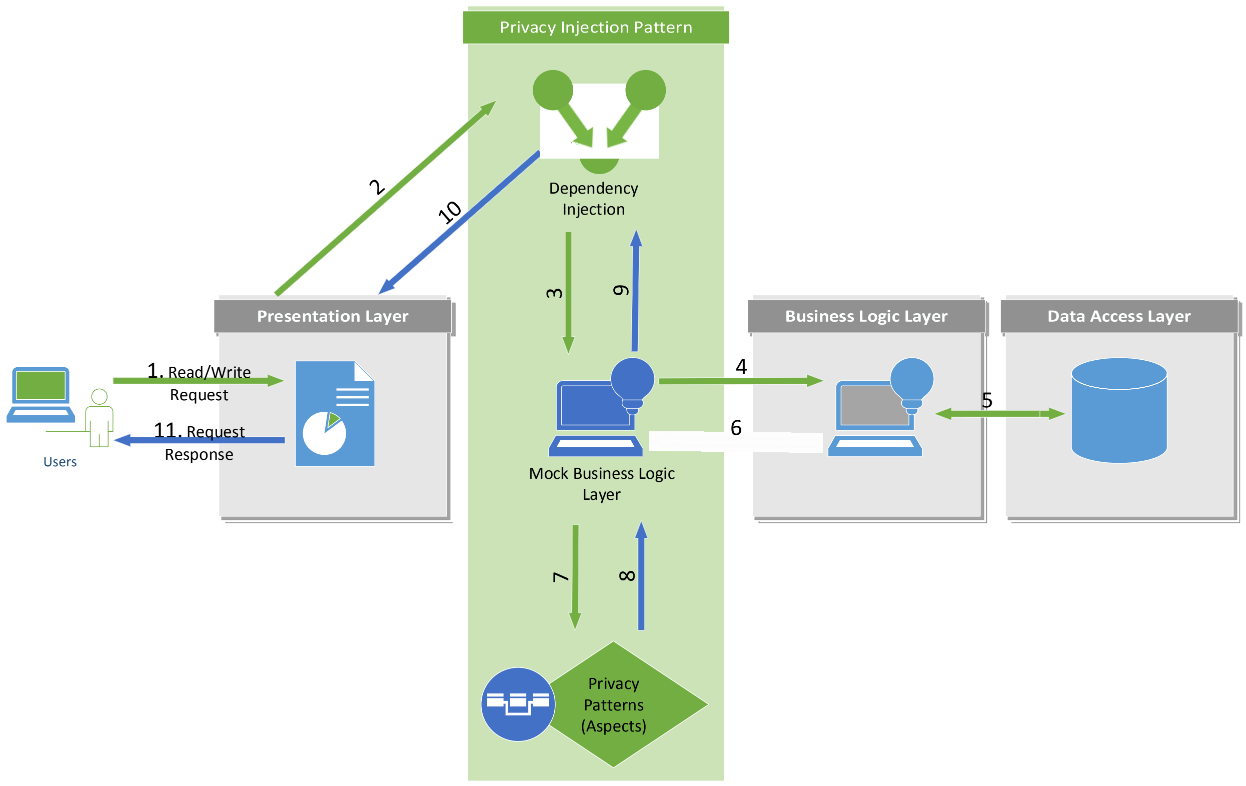


Figure 0.1. Privacy Injection Pattern (PIP) Architecture

References:

Culp, A. (2015, May 4). The Dependency Injection Design Pattern. Retrieved from MSDN: https://msdn.microsoft.com/en-us/library/vstudio/hh323705(v=vs.100).aspx

Prasanna, D. (2009). Dependency Injection: Design Patterns Using Spring and Guice, O"Reilly Media, 352 pages.

We illustrate our new Privacy Injection Pattern and the simplicity of its implementation with two use cases on the next pages. The first example injects well-known de-identification patterns in a banking use case, while a more complex example injects k-anonymity in a hospital use case. You can download the project code from the Dalhousie University's secure servers here: banking example and hospital example. The use cases and the code description are followed by survey questions that depend on your understanding of the PIP pattern and its illustration in the two use cases. Although you can answer the questions based on the understanding of the use cases, we encourage you to actually download the code and try it out.

BANKING USE CASE

To illustrate the use and 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 for protecting privacy. 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 (Cavoukian & Khaled El Emam, 2014; Shapiro, 2011; Narayanan & Shmatikov, 2008). Some de-identification patterns are substitution, shuffling, nulling out, character masking and cryptographic techniques. We implement the nulling out and character masking privacy patterns for illustration using aspect-oriented programming (AOP) in our example. We show how to use mocking and dependency injection techniques to automatically inject an 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 implement 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. You can download the project code from here also.

The banking application's use case scenario contains information about a customer account which is shown to a user who can be either a manager or an operator; the operator can view only some information while the manager can view all information. Figure 0.2 shows the sequence diagram of the maintain users' account use case as prepared by the software developer/engineer without including privacy controls (i.e., before applying PIP). Once the user is logged into the system, the system creates the CustomerManager and RoleManager objects, retrieves account and transaction details, and displays them on the screen.

In this use case, we want to 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. Thus, first the de-identification service DLL is loaded in the main program.

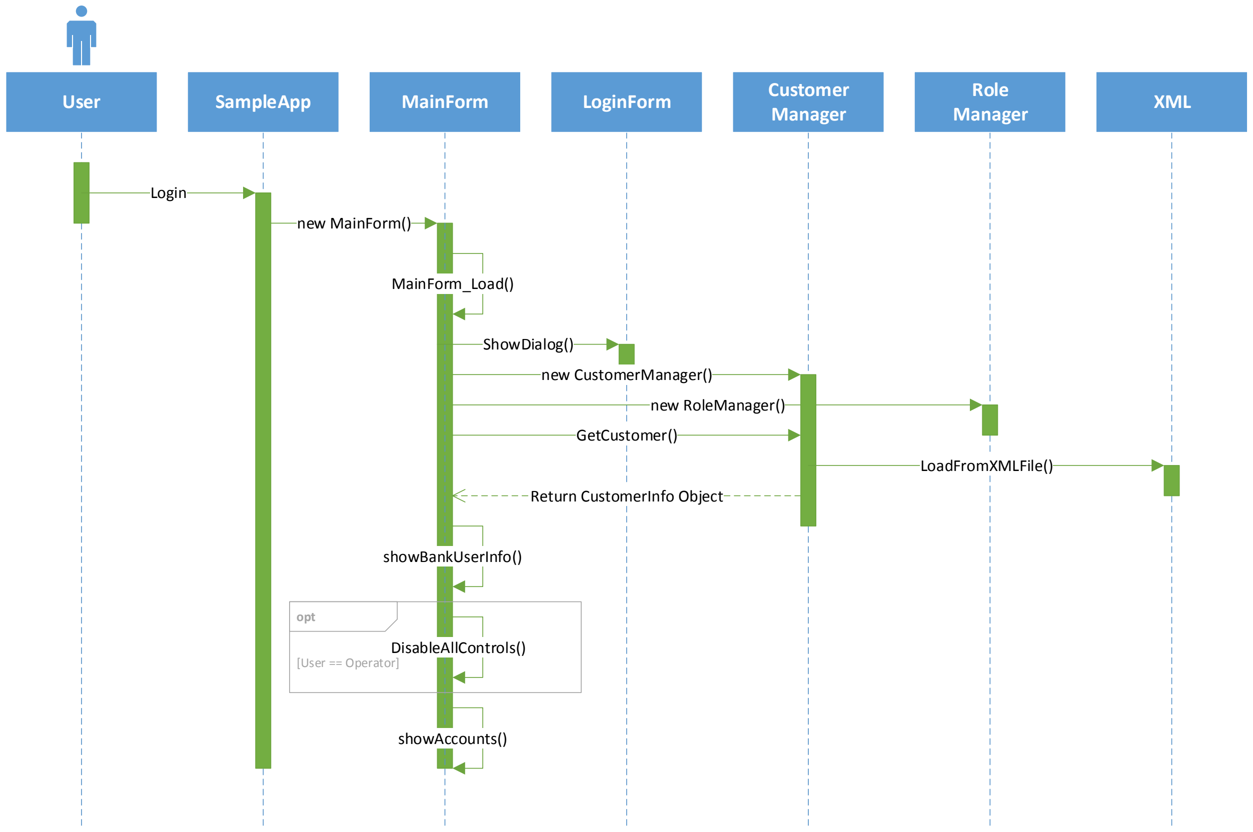


Figure 0.2. Sequence Diagram of User Account Before Applying PIP

Figure 0.3 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 deidentificationServiceLibName = "BankDeidentificationService.dll";

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

    var deidentificationServiceLibCompletePath = Path.Combine(currentPath, deidentifica-
tionServiceLibName);

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

    Assembly assembly = Assembly.LoadFrom(deidentificationServiceLibCompletePath);
    var deidentificationServiceType = assem-
bly.GetType("BankDeidentificationService.DeidentificationService");

    var serviceInstance = Activator.CreateInstance(deidentificationServiceType);

    deidentificationServiceType.InvokeMember("Initialize", BindingFlags.Default | Bind-
ingFlags.InvokeMethod, null, serviceInstance, null);
}
```

Figure 0.3. 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 called 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 0.4 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 to resolve the object to access its methods.

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

public static void SetupCustomerManager()
{
    if (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 CustomerInfoDeidentifiedImpl(result);
    });


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

Figure 0.4. Inject mocking object and invoke IOC

As the software developer has registered the mock object in the IoC container, when we call the customer manager object, it will call the mock customer manager object. Figure 0.5 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 GetCustomer method of the mock object and applies the de-identification service on the object. After applying de-identification, the system displays the information on the screen.

We apply the de-identification service by applying 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.

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

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

this.customerInfo.Accounts.ForEach(account =>
{
    this.cbAccount.Items.Add(string.Format("{0}-{1}", account.AccountType,
account.AccountNumber));
});
```

Figure 0.5. Resolve mocking object at runtime to get customer information

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

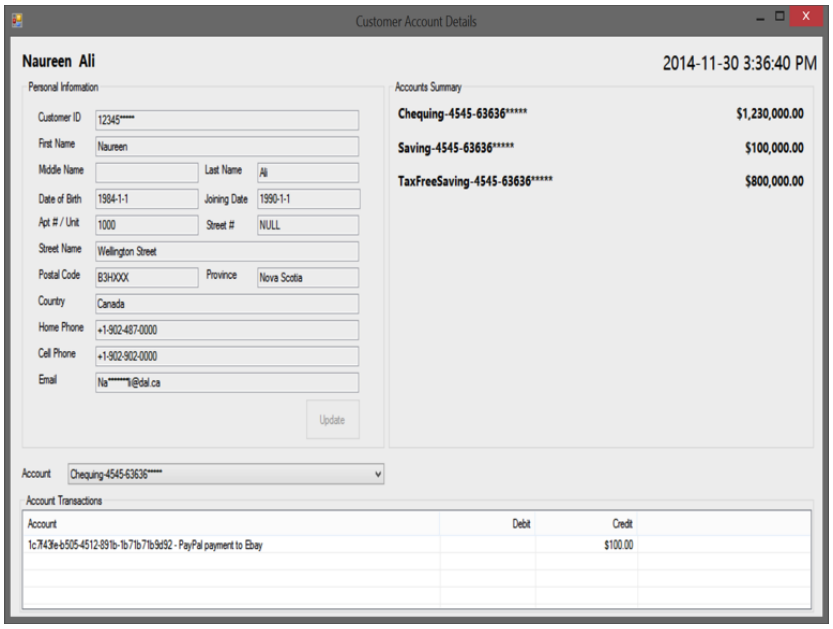
Figure 0.6. Apply LongStringDeidentification aspect on AccountNumber

We apply LongStringDeidentification to the AccountNumber property (Figure 0.6). In the LongStringDeidentification class, we provide the de-identification logic that will be applied on the field on which we bind as in Figure 0.7. 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 (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;
}
```

Figure 0.7. De-identification implementation in LongStringDeidentification class

Figure 0.8 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. We null out the street number.



Naureen Ali

2014-11-30 3:36:40 PM

Personal Information

Customer ID	12345*****		
First Name	Naureen		
Middle Name		Last Name	Ali
Date of Birth	1984-1-1	Joining Date	1990-1-1
Apt # / Unit	1000	Street #	NULL
Street Name	Wellington Street		
Postal Code	B3HXXX	Province	Nova Scotia
Country	Canada		
Home Phone	+1-902-487-0000		
Cell Phone	+1-902-902-0000		
Email	Na*******i@dal.ca		

Accounts Summary

Chequing-4545-63636*****	**$1,230,000.00**
Saving-4545-63636*****	**$100,000.00**
TaxFreeSaving-4545-63636*****	**$800,000.00**

Account: Chequing-4545-63636*****

Account Transactions

Account	Debit	Credit
1c7f43fe-b505-4512-891b-1b71b71b9d92 - PayPal payment to Ebay		$100.00

Figure 0.8. Operator Screen of Sample Bank Application

Figure 0.9 shows the sequence of object calls when the PIP is applied on the maintain users account use case to inject role-based de-identification for access control. The PIP can be applied to inject other privacy patterns in the system.

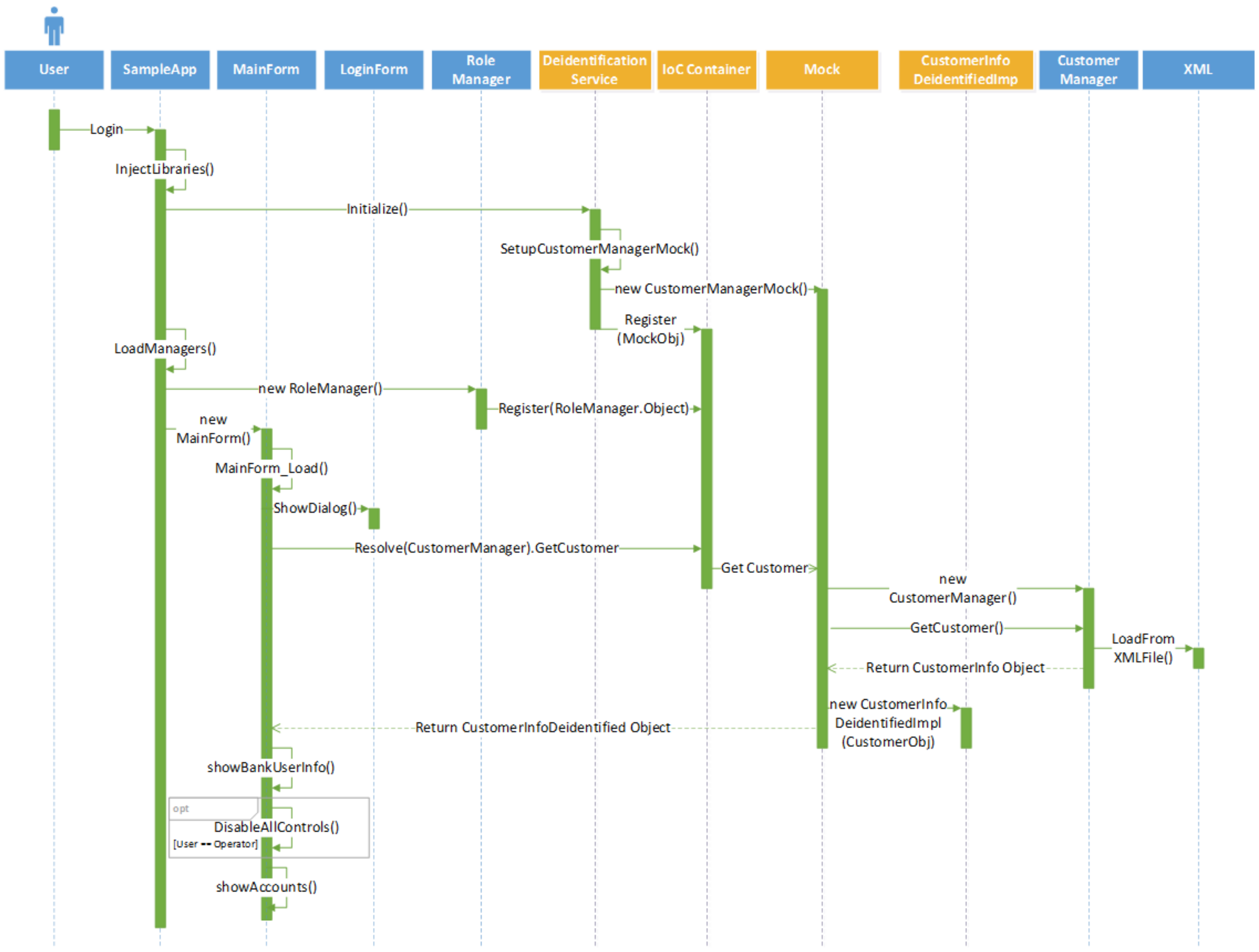


Figure 0.9. Main User Account Sequence Diagram after Applying PIP

We suggest that the PIP pattern can be used repeatedly in many places in a banking application e.g. to also inject a location/time privacy pattern that disallows the operator from viewing even more of customers' fields remotely outside of banking hours.

Bibliography

Cavoukian, A., & Khaled El Emam. (2014, June 25). De-identification Protocols: Essential for Protecting Privacy. Retrieved November 30, 2014, from http://www.privacybydesign.ca/content/uploads/2014/09/pbd-de-identifcation-essential.pdf

Narayanan, A., & Shmatikov, V. (2008). Robust de-anonymization of large sparse datasets. Proceedings of the 2008 IEEE Symposium on Security and Privacy.

Shapiro, S. S. (2011). Separating the Baby from the Bathwater - Toward a Generic and Practical Framework for Anonymization. IEEE.

HOSPITAL USE CASE

Implementing privacy in healthcare applications has become a leading concern of some researchers in the last few decades. To determine the use and implementation of the Privacy Injection Pattern (PIP) in a more complex use case, we employ the PIP in a use case scenario from a hospital management application that uses k-anonymity as the de-identification pattern while sharing data with other organizations. We implement the k-anonymity privacy method using ARX (ARX, n.d.) DLL for illustration using aspect-oriented programming (AOP) in our example. We show that the mocking and the dependency injection techniques automatically inject the AOP instance of the de-identification service. The k-anonymity is a de-identification method and it helps to preserve sensitive information. The idea behind k-anonymity is to reduce the granularity of the representation of the data in such a way that a given record cannot be distinguished from at least (k-1) other records (Sweeney, 2002; Aggarwal & Yu, 2008). This granularity is reduced using techniques such as generalization and suppression. In generalization, we replace the attribute value with a generalized value. Suppression is the technique where the attribute's value is removed completely.

Our technical implementation uses Visual Studio .Net (IDE), PostSharp (AOP), the Unity Container (Dependency Injection) and the Mock library to implement an example injection of our de-identification service into a hospital application. We note that the PIP may be implemented with other technologies, e.g. multi-platform heterogeneous technologies.

The hospital use case scenario is a search screen to retrieve information on the basis of criteria, such as country, city etc. The k-anonymity algorithm is applied to the data before sharing the data with other organizations. There is an option given to the user to apply k-anonymity to the data. This option helps us to study the behavior of the system before and after applying the proposed pattern. Figure 0.1 shows the sequence diagram of the search patient information use case before applying PIP.

In this case study, we want to inject the option-based de-identification pattern such that a power user can apply k-anonymity on the patient information to share the information with other organizations. The de-identification service DLL is loaded in the main program.

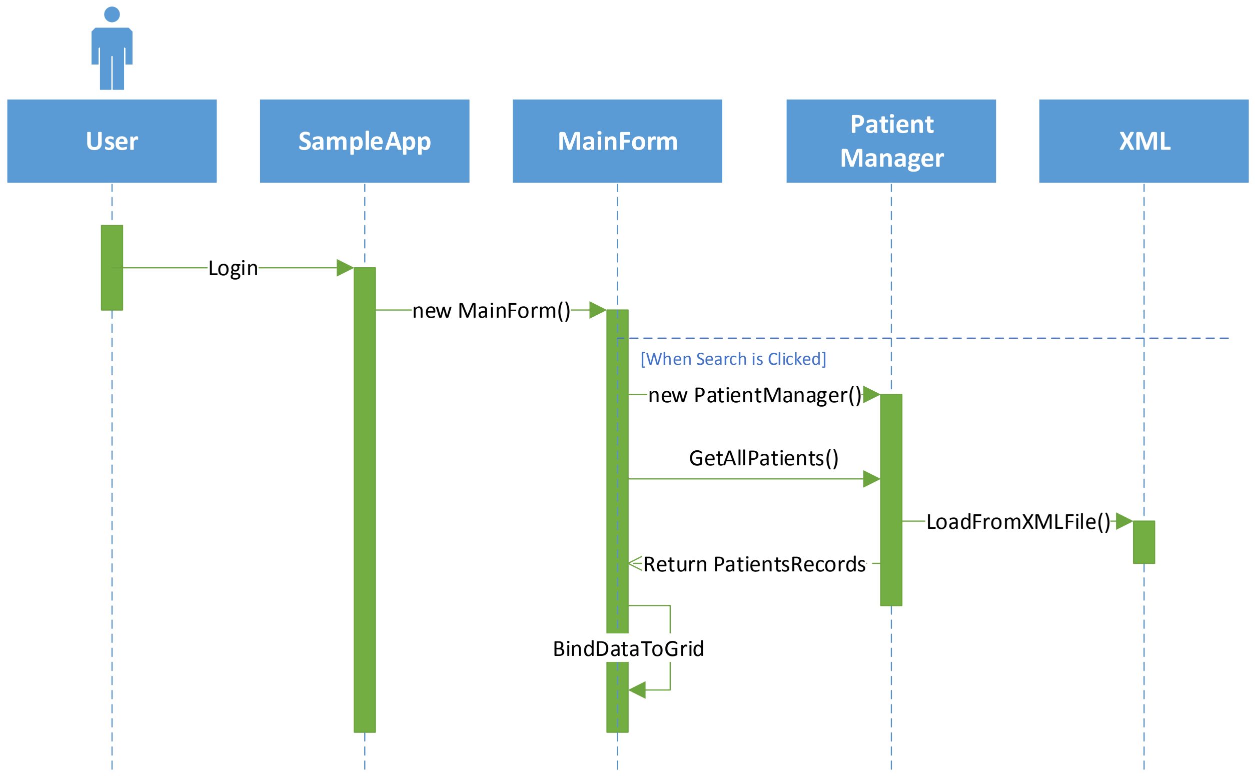


Figure 0.1. Sequence Diagram of Patient Search Before Applying PIP

Figure 0.2 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 applications, the software developer simply places the privacy DLL with other DLLs.

```
private static void InjectLibraries()
{
    var deidentificationServiceLibName = "DeIdentificationService.dll";

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

    var deidentificationServiceLibCompletePath = Path.Combine(currentPath, deidentifica-
tionServiceLibName);

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

    Assembly assembly = Assembly.LoadFrom(deidentificationServiceLibCompletePath);
    var deidentificationServiceType = assem-
bly.GetType("DeIdentificationService.DeIdentificationServiceInitializer");

    var serviceInstance = Activator.CreateInstance(deidentificationServiceType);

    deidentificationServiceType.InvokeMember("Initialize", BindingFlags.Default | Bind-
ingFlags.InvokeMethod, null, serviceInstance, null);
}
```

Figure 0.2. 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 PatientManager, which is an implementation of the IPatientManager interface. PatientManager has multiple methods that fetch patients' information from the XML (or database) on the basis of certain criteria. The software engineer creates a mock object of the IPatientManager type and then registers it. The engineer also setups the updated implementation of all the methods to fetch patients' details in the same way as the originating object method, and then applies the de-identification aspect on this object.

Figure 0.3 shows the implementation of methods to de-identify the information. Subsequently, when the developer calls the original PatientManager method, the updated method of the mock object is invoked. In the Unity Container, for dependency injection the software engineer first registers the object at the beginning of the program to resolve the object to access its methods.

```
private static void SetupManagerObjects()
{
    if (Common.Ioc.IocContainer.Instance.IsRegistered(typeof(IPatientManager)))
    {
        return;
    }
    var patientManagerMock = new Mock<IPatientManager>();

    patientManagerMock.Setup(x => x.GetAllPatients()).Returns(() =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.GetAllPatients();

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();

        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });

    patientManagerMock.Setup(x =>
x.SearchPatientByCity(It.IsAny<string>())).Returns((string cityName) =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.SearchPatientByCity(cityName);

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();

        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });

    patientManagerMock.Setup(x =>
x.SearchPatientByCountry(It.IsAny<string>())).Returns((string countryName) =>
    {
        var patientManager = new PatientManager();
        var result = patientManager.SearchPatientByCountry(countryName);

        var deIndentifiablePatients = from patient in result
                                      select new PatientInfoDeIdentified(patient);

        var patientRecordCollection = new PatientInfoRecordCollection();

        patientRecordCollection.PatientRecords = deIndentifiablePatients.ToList();

        return patientRecordCollection.PatientRecords;
    });
    Common.Ioc.IocContainer.Instance.Register<IPatientManager>(patientManagerMock.Object);
}
```

Figure 0.3. Inject mocking object and invoke IOC

Figure 0.4 shows how the software developer resolves the IPatientManager object to fetch patients' information. The developer will call the original function, for example GetAllPatients, to fetch the required information. This action calls the GetAllPatients method of the mock object and applies the de-identification service on the object. After applying de-identification, the system displays the information on the screen.

```
IEnumerable<IPatientInfo> searchedPatientRecords = new List<IPatientInfo>();

var patientManager = Common.Ioc.IocContainer.Instance.Resolve<IPatientManager>();

switch (this.cmbSearchBy.SelectedItem.ToString())
{
    case  "Country":
        searchedPatientRecords = patientManag-
er.SearchPatientByCountry(this.txtSearchTerm.Text);
        this.BindDataToGrid(searchedPatientRecords);
        break;

    case "City":
        searchedPatientRecords = patientManag-
er.SearchPatientByCity(this.txtSearchTerm.Text);
        this.BindDataToGrid(searchedPatientRecords);
        break;

    case "All Records":
        searchedPatientRecords = patientManager.GetAllPatients();
        this.BindDataToGrid(searchedPatientRecords);
        break;
}

this.lblRecordsCount.Text = string.Format("{0} {1} found",
                                          searchedPatientRecords.Count(),
                                          searchedPatientRecords.Count() > 1 ?
"Records" : "Record");
```

Figure 0.4. Resolve mocking object at runtime to get patient information

The software engineer applies the de-identification service by applying a de-identification aspect with properties or methods. In this example, the developer applies de-identification on the PatientRecords property. When the software fetches patient records through this property, it applies the de-identification aspect on the field and returns a value.

```
public class PatientInfoRecordCollection
{
    [DeIdentificationAspect(AnonymizationFactor = 3)]
    public List<PatientInfoDeIdentified> PatientRecords { get; set; }
}
```

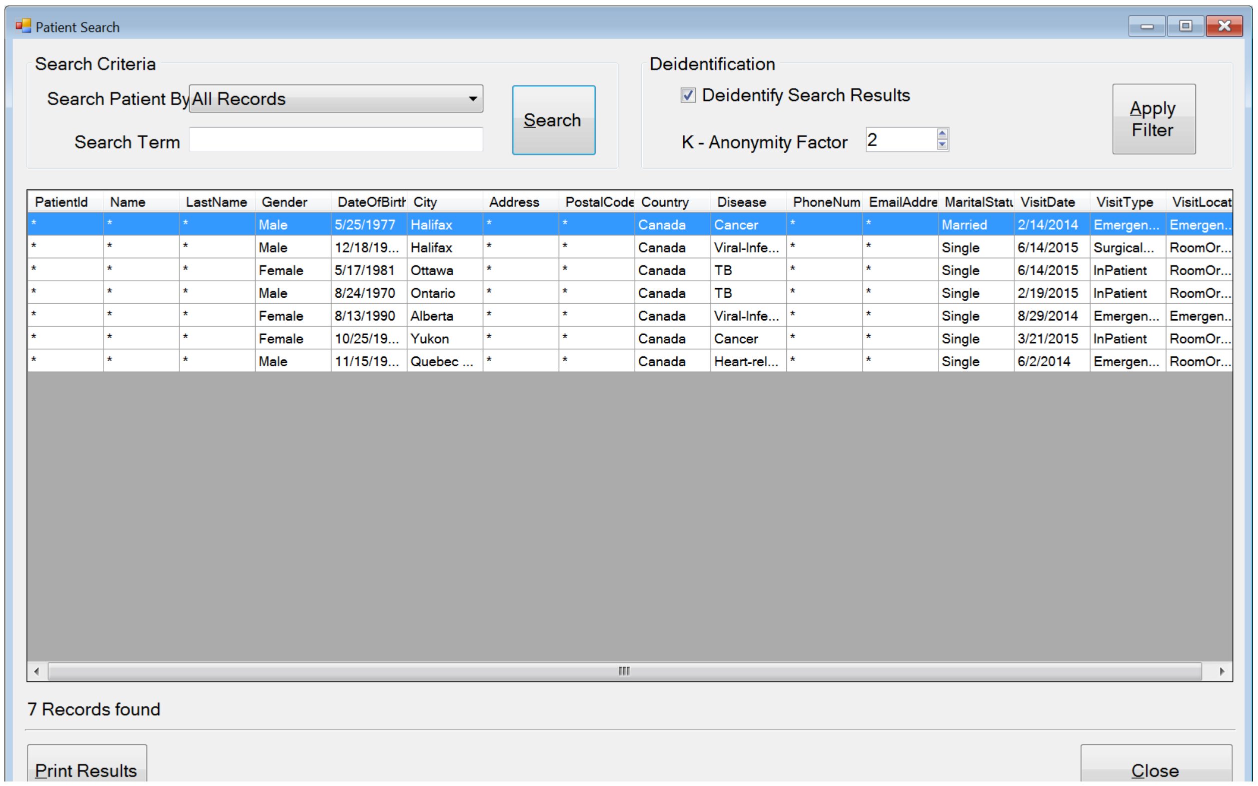
Figure 0.5. Apply Deidentification aspect on PatientRecords

The software developer applies the DeidentificationAspect to the PatientRecords property (Figure 0.5). In the DeidentificationAspect class, s/he provides the de-identification logic that will be applied on the bind field as in Figure 0.6. DeidentificationAspect then calls DeidentifyRecords of the DeIdentification class to use the ARX anonimize function to de-identify the data and return it to the calling method.

```
[Serializable]
public class DeIdentificationAspect : LocationInterceptionAspect
{
    public int AnonymizationFactor { get; set; }

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

        //DeidentifyRecords uses ARX anonimize function to anonimize the list provided as 
argument
        var newValue = new DeIdentifica-
tion<PatientInfoDeIdentified>().DeIdentifyRecords((List<PatientInfoDeIdentified>)
args.Value, this.AnonymizationFactor);

        args.Value = newValue.ToList();
    }
}
```

Figure 0.6. De-identification implementation in DeidentificationAspect class

Figure 0.7 shows a patient search screen of the hospital application that results from the use of the PIP for injection of the k-anonymity method. The patients' information is searched by different criteria and the k factor for k-anonymity is also provided by the user on the screen. For k-anonymization of the records, we provide the attribute type (identifying, quasi-identifying, insensitive) of each attribute in the input list. For the quasi-identifying attribute, an attribute hierarchy is required.



Search Patient By: All Records

Search Term:

☑ Deidentify Search Results

K - Anonymity Factor: 2

PatientId	Name	LastName	Gender	DateOfBirt	City	Address	PostalCode	Country	Disease	PhoneNum	EmailAddre	MaritalStatu	VisitDate	VisitType	VisitLocat
*	*	*	Male	5/25/1977	Halifax	*	*	Canada	Cancer	*	*	Married	2/14/2014	Emergen...	Emergen..
*	*	*	Male	12/18/19...	Halifax	*	*	Canada	Viral-Infe...	*	*	Single	6/14/2015	Surgical...	RoomOr...
*	*	*	Female	5/17/1981	Ottawa	*	*	Canada	TB	*	*	Single	6/14/2015	InPatient	RoomOr...
*	*	*	Male	8/24/1970	Ontario	*	*	Canada	TB	*	*	Single	2/19/2015	InPatient	RoomOr...
*	*	*	Female	8/13/1990	Alberta	*	*	Canada	Viral-Infe...	*	*	Single	8/29/2014	Emergen...	Emergen..
*	*	*	Female	10/25/19...	Yukon	*	*	Canada	Cancer	*	*	Single	3/21/2015	InPatient	RoomOr...
*	*	*	Male	11/15/19...	Quebec ...	*	*	Canada	Heart-rel...	*	*	Single	6/2/2014	Emergen...	RoomOr...

7 Records found

Figure 0.7. Patient Search Screen of Hospital Application

Figure 0.8 shows the sequence of object calls when PIP is applied on the patient search use case to inject privacy by using k-anonymity in the system.

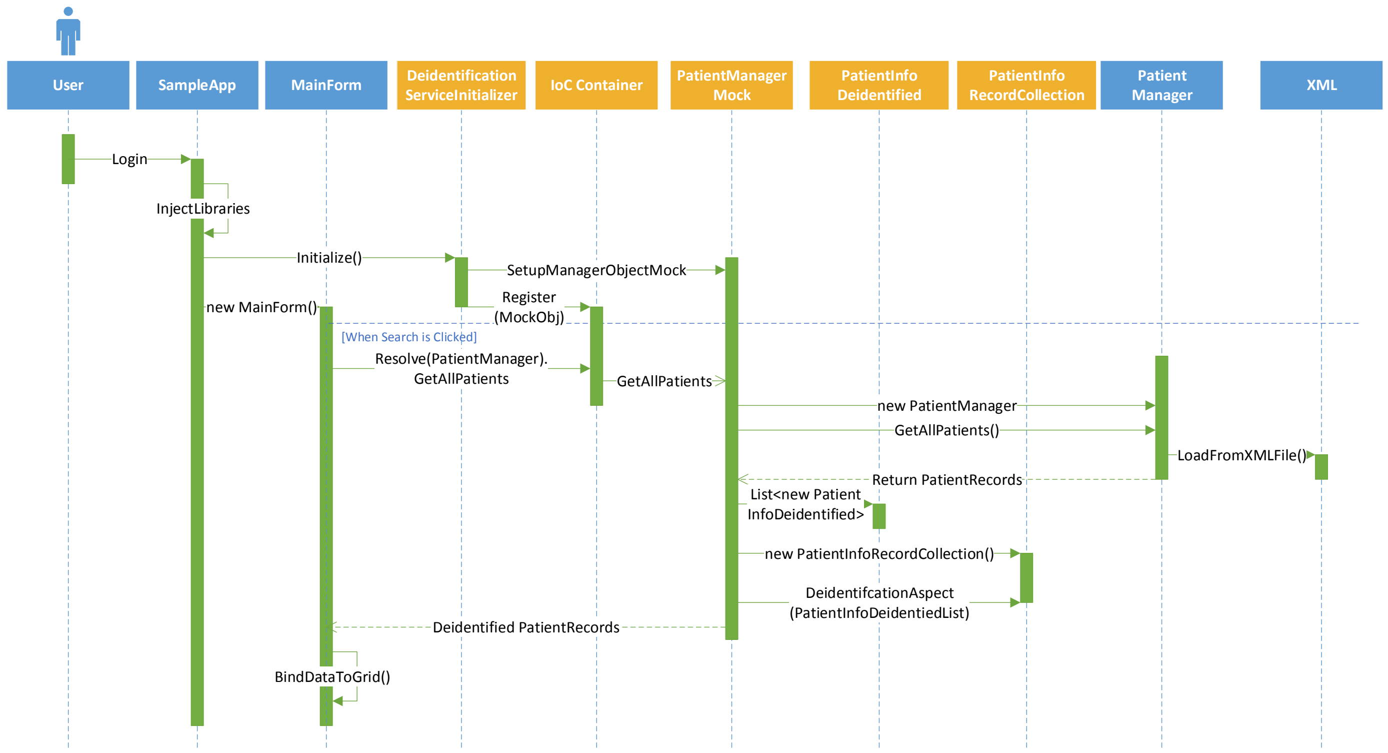


Figure 0.8. Patient Search Sequence Diagram after Applying PIP

Bibliography

Aggarwal, C. C., & Yu, P. S. (2008). Privacy-Preserving Data Mining: Models and Algorithms. Springer.

ARX. (n.d.). ARX - Powerful Data Anonymization. Retrieved from http://arx.deidentifier.org/

Cavoukian, A., & Khaled El Emam. (2014, June 25). De-identification Protocols: Essential for Protecting Privacy. Retrieved November 30, 2014, from http://www.privacybydesign.ca/content/uploads/2014/09/pbd-de-identifcation-essential.pdf

Latanya Sweeney, k-Anonymity: A Model for Protecting Privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10(5): 557-570(2002).

The rest of the survey questions will depend on your understanding of the PIP pattern and the Hospital and Banking use cases. If you have not as yet, please download the code and try it out.

5. Did you download and try running the example applications or try extending the examples?

☐ Yes ☐ No

6. I understood the PIP concepts, examples, and/or code.

☐ Strongly Agree ☐ Agree ☐ Somewhat Agree ☐ Neutral ☐ Somewhat Disagree ☐ Disagree ☐ Strongly Disagree

PART – C

7. Suppose you have developed an application and it is in beta testing phase. Suddenly it has been pointed out to you that your company has promised to embed privacy in all software applications a few months back. You realize that you have not implemented privacy in your application and if you implement it, it requires changes in many modules of the application. Furthermore, assume the appropriate privacy controls, such as de-identification, that you wish to insert in your code have already been coded and verified. Your challenge is to easily modify your existing application to use one or more of these privacy controls. You have read about and studied the privacy injection pattern (PIP) and its example uses. The following questions target to what extent you think that PIP may help you to resolve your issue.

Perceived Usefulness

	Strongly Agree	Agree	Somewhat Agree	Neutral	Somewhat Disagree	Disagree	Strongly Disagree
The task would be difficult to perform without this pattern.							
Using PIP would give me greater control over this task.							
Using PIP would improve my performance on this task.							
The PIP would address my task-related needs.							

Using PIP would save my time.							
Using PIP in my application would enable me to accomplish embedding privacy more quickly.							
The PIP pattern would support critical aspects of my task.							
Using the PIP pattern would allow me to accomplish more work than would otherwise be possible.							
Using PIP would enhance my effectiveness on this task.							
Using PIP would improve the quality of the work I do on this task.							
Using PIP on this task would increase my productivity.							
Using PIP would make it easier to do							

this task.							
I would find PIP useful in my job.							

Perceived Ease of Use

	Strongly Agree	Agree	Somewhat Agree	Neutral	Somewhat Disagree	Disagree	Strongly Disagree
I will become confused when I use the pattern.							
I would make errors frequently when using the pattern.							
Interacting with the pattern is often frustrating.							
I would need to consult the user manual often when using the pattern.							
Implementing the pattern would require a lot of my mental effort.							
I would find it easy to recover from errors/issues encountered while using the pattern.							
I would find PIP to							

be flexible to interact with.							
I would find it easy to get PIP to do what I want it to do.							
The pattern would often behave in unexpected ways.							
I find it cumbersome to use the pattern.							
My interaction with the pattern is easy for me to understand.							
It will be easy for me to remember how to perform tasks using the pattern.							
The pattern provides helpful guidance in performing tasks.							
I would find PIP easy to use.							

PART – D

7. Suppose you have developed an application and it is in beta testing phase. Suddenly it has been pointed out to you that your company has promised to embed privacy in all software applications a few months back. You realize that you have not implemented privacy in your application and if you implement it, it requires changes in many modules of the application. Furthermore, assume the appropriate privacy controls, such as de-identification, that you wish to insert in your code have already been coded and verified. Your challenge is to easily modify your existing application to use one or more of these privacy controls. You have read about and studied the privacy injection pattern (PIP) and its example uses. The following questions target to what extent you think that PIP may help you to resolve your issue.

Perceived Usefulness

	Strongly Agree	Agree	Somewhat Agree	Neutral	Somewhat Disagree	Disagree	Strongly Disagree
The task would be difficult to perform without this pattern.							
Using PIP would give me greater control over this task.							
Using PIP would improve my performance on this task.							
The PIP would address my task-related needs.							

Using PIP would save my time.							
Using PIP in my application would enable me to accomplish embedding privacy more quickly.							
The PIP pattern would support critical aspects of my task.							
Using the PIP pattern would allow me to accomplish more work than would otherwise be possible.							
Using PIP would enhance my effectiveness on this task.							
Using PIP would improve the quality of the work I do on this task.							
Using PIP on this task would increase my productivity.							
Using PIP would make it easier to do							

this task.							
I would find PIP useful in my job.							

Perceived Ease of Use

	Strongly Agree	Agree	Somewhat Agree	Neutral	Somewhat Disagree	Disagree	Strongly Disagree
I will become confused when I use the pattern.							
I would make errors frequently when using the pattern.							
Interacting with the pattern is often frustrating.							
I would need to consult the user manual often when using the pattern.							
Implementing the pattern would require a lot of my mental effort.							
I would find it easy to recover from errors/issues encountered while using the pattern.							
I would find PIP to							

be flexible to interact with.							
I would find it easy to get PIP to do what I want it to do.							
The pattern would often behave in unexpected ways.							
I find it cumbersome to use the pattern.							
My interaction with the pattern is easy for me to understand.							
It will be easy for me to remember how to perform tasks using the pattern.							
The pattern provides helpful guidance in performing tasks.							
I would find PIP easy to use.							

PART – E

7. Do you think we can improve PIP?

☐ Yes ☐ No

8. If yes, how we can improve this pattern?

9. If you have to implement privacy in your application, would you use this pattern?

☐ Yes ☐ No

10. If No, then how would you implement privacy in your application without modifying your existing application or if required, modifying it to a small extent?

Would you like to receive a copy of study findings when published?

☐ Yes (please specify your email address) --------------------------

☐ No thanks

-----------------------*Thanks for participating in the Survey***------------------------**

****************Have a great day***************