School of Electrical and Information Engineering
University of the Witwatersrand, Johannesburg
ELEN 7039 – Information Engineering Design

ELEN 7039 – Information Engineering Design

Student Name: Adrian Freemantle

Student Number: 1558184

Date: 2016/00/00

Abstract

Debit Orders provide a cost effective and convenient way for South African consumers to pay for services such as long-term insurance policies, with 48 million Debit Orders Debit Orders to the value of R61 billion being processed each month in South Africa. The current Debit Order process is open to abuse as written and voice recorded Debit Order mandates do not verify that the individual granting the mandate is the legal owner of the bank account, resulting in 8% of transactions being disputed. The South African Reserve Bank and the Payments Association of South Africa have launched the Authenticated Collections process to replace the Debit Order process. Transitioning to Authenticated Collections poses a challenge for many companies whose existing Debit Order processes are implemented on legacy platforms. This paper presents the design of a Billings System for a long-term insurer with an established legacy position. The Domain-Driven Design concept of a Bounded Context is used to extract an explicit Billings Domain Model from a legacy system, allowing the new model to evolve over time without affecting legacy functionality. The design is shown to be extensible, flexible, and maintainable and advantages and disadvantages of applying Domain-Driven Design techniques to replacing legacy systems functionality is discussed.

Table of Contents

List of Figures

1 Introduction

Debit Orders [1] provide a cost effective and convenient way for South African customers to pay *monthly premiums on life and investment policies, mortgage and hire purchase payments, medical aid subscriptions* [1] etc. According to the Payments Association of South Africa (PASA), 48 million *Debit Orders Debit Orders to the value of R61 billion are processed each month in South Africa* [2].

The current Debit Order process is open to abuse as written and voice recorded Debit Order mandates do not verify that the individual granting the mandate is the legal owner of the bank account, resulting in 8% of transactions being disputed [2]. In response to this problem, the South African Reserve Bank (SARB) and PASA have launched the Authenticated Collections [2] process, requiring all mandates to be electronically authenticated with the customer's bank. The SARB has approved a transition phase to allow companies to migrate to Authenticated Collections by no later than 30 June 2019 [3].

Transitioning to Authenticated Collections poses a challenge for many companies whose existing Debit Order processes are implemented on legacy platforms [4]. Modification of existing systems poses a risk if these systems are undocumented or have no automated tests [5].

Section two provides an overview of Debit Orders and Authenticated Collections in the context of premium collections within the South African long-term insurance industry. Section three discusses the challenges faced by an established long-term insurance company in migrating their existing legacy systems to the Authenticated Collections process, defining the problem scope, constraints, requirements and assumptions.

Section four presents the methodology used to design a new billings system and section five presents a high-level design. Section six presents a detailed design with the project implementation discussed in section seven. The design is critically reviewed in section eight.

2 Context

Long-term insurance is an arrangement between an insurer and the insured, in which the insurer commits to paying a sum of money to the insured, or their nominated beneficiaries, if the insured experiences a life-changing event such as death, disability or terminal illness. In return for this service, the insured pays the insurer a monthly insurance premium. The agreement is formalised through an insurance policy, a contract which specifies the terms and conditions agreed upon, with the insured being referred to as the policy holder [6].

Monthly premiums may be paid directly by the policy holder; however, this can be inconvenient as the policy holder must ensure that the payment is made each month. A more convenient option is to permit the insurer to directly debit the policy holder's account through the Debit Order process.

> *Debit Orders are payment instructions directed by a third party to the account of a paying customer of a bank in terms of authority (a mandate) granted by the paying customer to such third party [1]*

The insurer may only submit a Debit Order after obtaining a written, electronic or recorded voice mandate from the customer. The mandate specifies the bank account, debit amount and a fixed day-of-the-month on which each Debit Order will be processed [1].

Written and voice recorded mandates are currently open to abuse as they do not verify that the individual granting the mandate is the legal owner of the bank account, resulting in 8% of transactions being disputed [2]. In response to this problem, the South African Reserve Bank (SARB) and the Payments Association of South Africa(PASA) launched the Authenticated Collections process [2].

Authenticated Collections requires billing institutions to electronically submit all debit mandates to the client's bank. The bank is required to verify the mandate with the client via an electronic medium such as an SMS message. A successful authorisation results in the bank providing an authorisation code to the billing institution which must be submitted with all future debit requests [2].

The Authenticated Collections process has been in operational since 1 October 2016, however the SARB has approved a transition phase to allow companies to migrate their existing Debit Order processes to Authenticated Collections by no later than 30 June 2019 [3].

3 Problem Definition

MegaSure[1] is a South African financial-services [7] company and distributor of long-term insurance products. The company has over one million active long-term insurance policies, with approximately 95% of customer's premiums being collected via the Debit Order process. MegaSure must switch to the Authenticated Collections process by 30 June 2019, however implementing this on MegaSure's existing legacy platforms poses a challenge.

MegaSure's current systems are hosted on a mainframe [8] computer which runs the Pick Operating System [9], with policy information stored in the D3 Multidimensional Database Management System [10]. The main components of this system are shown in Figure 1.

D3 was first released in 1973 and is *a terminal-oriented, multi-user system, designed for interactive use to facilitate decision making from a shared data base* [9]. The Pick file structure *is considered mathematically "three dimensional"* [9] and data files are queried using the ACCESS query language. Menus, macros and scripts are programmed using the Pick BASIC programming language, *specifically tailored for data base management in a multi-user environment* [9].

[1] A fictional company.

The legacy systems which run on this platform have evolved over twenty years and perform their current functions adequately, making a rewrite impractical. Modification of existing systems would be risky as they are undocumented, have no automated tests, and most of the developers who created these systems have retired or left the company.

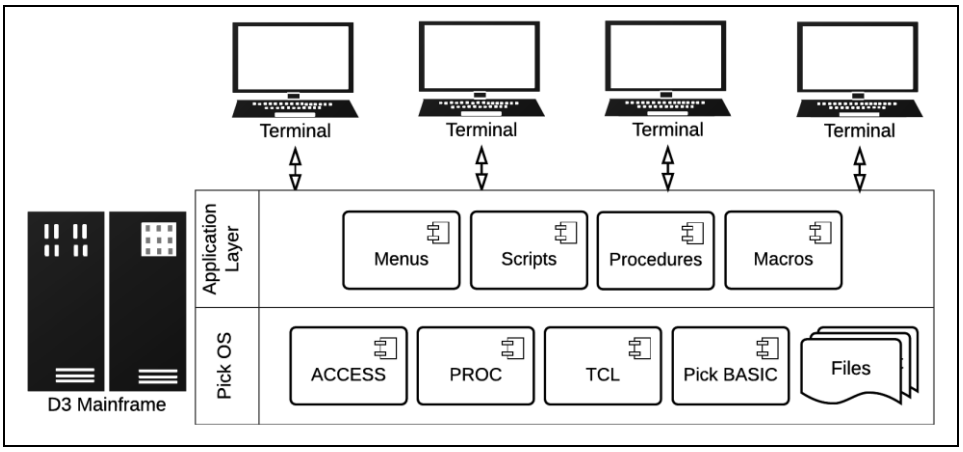


Figure 1: The Pick Operating System

The existing D3 data-model poses a further challenge as it was created when MegaSure provided a single life insurance product. The creators of the model assumed that a policy holder would only have a single policy and stored policy holder and Debit Order mandate information directly in the policy record. As MegaSure expanded its product range to cover disability and terminal illness, existing customers purchased additional insurance policies, resulting in a fragmented and inconsistent view of the customer.

3.1 Scope

MegaSure launched the Billings System project to transition to Authenticated Collections and address problems inherent in the D3 data-model. The project is broken into four incremental [11] phases:

- Phase 1 – Implement a prototype Billings System.
- Phase 2 – Stabilise the Billings System prototype.
- Phase 3 – Integrate with an authorised Authenticated Collections service provider.
- Phase 4 – Transition to the Billings System.

The design presented in this paper is limited to Phase 1 of the project.

3.2 Requirements

Requirements for the prototype phase were elicited by using wireframe screen-designs and captured in the User Story [12] format. The four most important user stories are:

3.2.1 Separate the Policy Payer from the Policy Holder

As a client, I want to be able to pay the premium of a long-term insurance policy for which I am not the policy holder, so that I can insure my immediate and extended family members against life changing events.

3.2.2 Consolidated View of Policy Premium Collection Statuses

As a Client Services consultant, I want to see a consolidated view of the premium collection status for all policies paid for by a client, so that I can provide a more efficient service to our clients.

3.2.3 Compliance with Authenticated Collections Regulations

As a Compliance auditor, I want all premium debits made against a bank account to be associated with a valid electronic mandate, so that I can prove that MegaSure is compliant with Authenticated Collections regulations.

3.2.4 Delays in the Electronic Mandate Process Must Not Affect Policy Sales

As a Sales agent, I want to be able to complete the sale of a policy without being affected by delays in the electronic mandate process, so that I can maximise my commission by continuing to sell additional policies.

3.3 Constraints

The Billings System must integrate with D3 without affecting or requiring changes to existing D3 functionality.

3.4 Assumptions

1. The ability to support multiple currencies is not required.
2. Pro rata billing of premiums is not required.
3. Credit Card and other forms of payment are not considered for the prototype.

4 Design Methodology

Domain-Driven Design (DDD) [13] is an approach to modelling and designing complex software systems which are resilient to changing business rules and requirements. Developers and domain experts collaborate to create a shared Domain Model [14]; a system of abstractions which express the concepts, data and behaviour of the problem domain through a shared Ubiquitous Language [13].

Multiple models will exist within a complex problem domain, frequently using the same term to convey different concepts. Each model exists in a specific context within which its terms have explicit and unambiguous meaning, however when the models do not have explicit boundaries, they may become mixed, resulting in software which *becomes buggy, unreliable and difficult to understand* [13]. This is prevented by explicitly defining the context boundaries of each model [13].

> *A Bounded Context delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it related to other contexts. [13, p. 336]*

Instead of refactoring or rewriting a legacy system, the DDD concept of a Bounded Context provides an alternate approach. Complex and strategically important subdomains are remodelled into explicit Domain Models within a Bubble Context [15]. A Bubble Context is a small Bounded Context which is separated from the legacy model through an Anticorruption Layer (ACL) [13], an explicit boundary within which *the team can evolve*

a model that addresses the chosen area, relatively unconstrained by the concepts of the legacy systems [15].

As shown in Figure 2, the legacy system and the Bubble Context co-exist, allowing the new model to mature over time until it meets all specified requirements. While the Bubble Context operates independently of the legacy system and has its own model and data store, it may be necessary to keep its data synchronised with the legacy system. The ACL coordinates this synchronisation by translating *conceptual objects and actions from one model and protocol to another* [13].

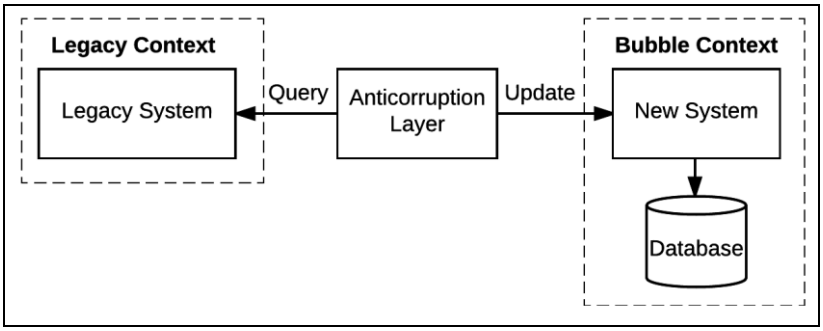


Figure 2: Bubble Context synchronised via an Anticorruption Layer

5 High-Level Design

The Billings Bubble Context shown in Figure 3 enables a new Billings system to be modelled and implemented as an evolutionary prototype [11] without affecting existing functionality in D3.

The ACL in Figure 2 uses a polling 'query-update' process to keep the Bubble Context synchronised with the legacy system. Polling D3 to determine the state of each policy would be impractical as it would result in excessive network traffic and would negatively impact the performance of existing procedures, scripts and batch processes. Figure 3 shows an event-driven [16] approach where the ACL is notified of changes to D3 policy files through asynchronous Event Messages [17]. An event-driven approach is more efficient as MegaSure's policies undergo relatively few changes after they have been captured.

As shown in Figure 3, the *D3 Event Publisher* is a service which is external to D3, acting as a Channel Adapter [17] to integrate D3 with Message-Oriented Middleware (MOM) [18] such as RabbitMQ [19]. The *D3 Event Publisher* uses the Pick Operating System's Open Systems File Interface [10] to monitor the file system for changes to D3 policy files. Each change to a policy file is converted to an Event Message which is placed on a Message Channel [17], also known as Message Queue [17].

The *D3 Event Integrator service* is the ACL for the Billings Bubble Context, dequeuing D3 Event Messages from the Message Channel and translating them into Billings Command Messages [17]. Command Messages are dispatched to Application Services [20] within the Billings System which handle the commands and perform the required actions on the Domain Model.

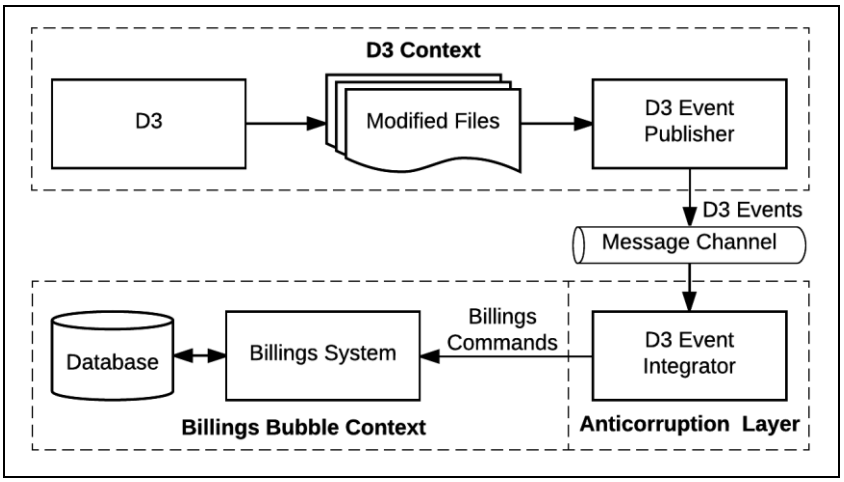


Figure 3: Billings Bubble Context

6 Detailed Design

This section presents the detailed design of the Billings Anticorruption Layer and the Billings Domain Model.

6.1 Anticorruption Layer

Figure 4 shows a Class Diagram [21] of the primary classes and interfaces of the ACL. The ACL uses the Observer [22] pattern to decouple [17] the receiving of an Event Message from its translation into one or more Billings Command Messages. The *CommandDispatcher* class implements the *Observer* interface, enabling an instance of the *CommandDispatcher* to be passed as a parameter to the *subscribe* method on an object which implements the *MessageReceiver* interface. As shown in Figure 5, the *MessageReceive*r calls the *onReceived* method of the *CommandDispatcher* when a message has been received, passing the Event Message object as a parameter.

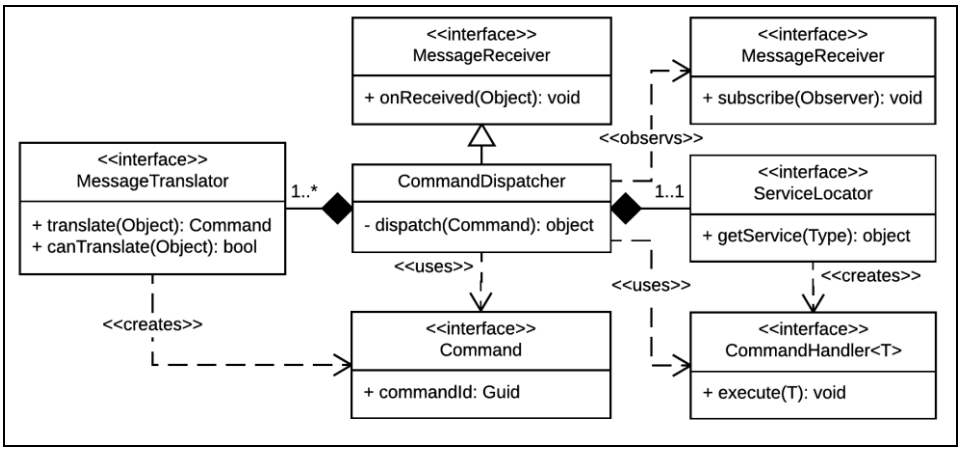


Figure 4: Class Diagram for the Anticorruption Layer

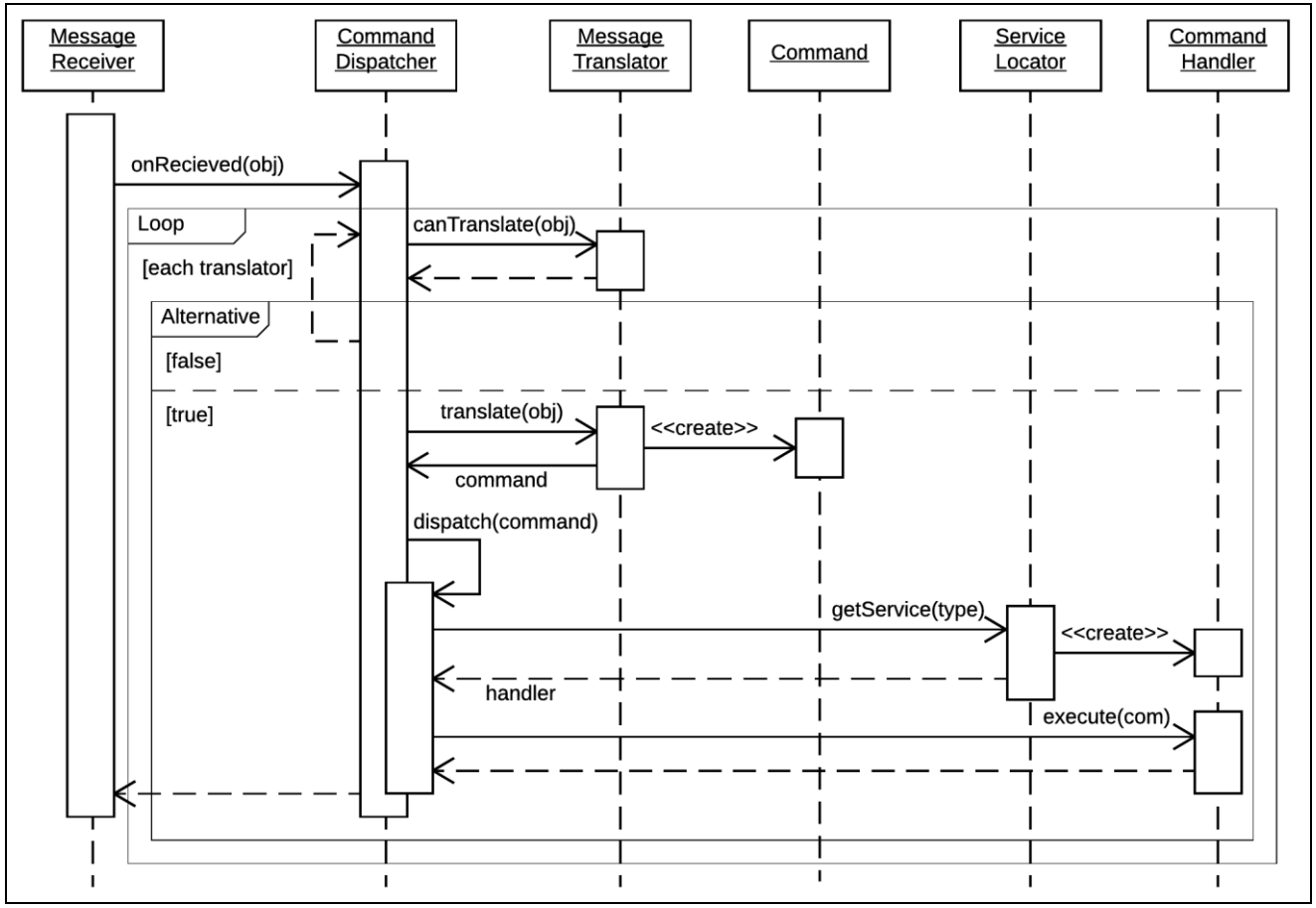


Figure 5: Sequence Diagram for Anticorruption Layer

MessageTranslator objects are Abstract Factories [22] which implement different Strategies [22] for converting Event Messages. The *onReceived* method of the *CommandDispatcher* calls the *canTranslate* method of each *MessageTranslator* to find the appropriate translator, it then calls the *translate* Factory Method [22] to convert the Event Message object to a *Command* object which is passed to the *dispatch* method.

The *dispatch* method of the *CommandDispatcher* is responsible for identifying the *CommandHandler* type for a *Command* object, requesting an instance of it from the *ServiceLocator* [23]. The *getService* method of the *ServiceLocator* is a Factory Method which is responsible for instantiating a *CommandHandler* and its dependencies. The *ServiceLocator* may contain Builders [22] for each *CommandHandler* type or it may use an Inversion of Control Container [24] such as Autofac [25].

The *CommandHandler* interface uses Generics [26] to indicate which command type it handles; for example, *CommandHandler<RaiseMontlyPremium>* would indicate that the *CommandHandler* executes the *RaiseMontlyPremium* Command Message. Generic typing allows the *ServiceLocator* to instantiate the correct *CommandHandler*, but results in the *execute* method requiring a parameter of the type specified by the generic argument of the *CommandHandler* interface. Attempting to pass a *Command* interface object to the execute method will result in a compile time error. This is prevented by casting the *CommandHandler* object to a dynamic [27] type, bypassing compile-time type checking and dynamically casting the *Command* object to its concrete type at runtime [27].

6.2 Domain Model

The Class Diagram in Figure 6 shows the Aggregates [13] of Billings Domain Model and their respective relationships. Each Aggregate is a consistency boundary for a collection of encapsulated Entities [13] and Value Objects [14], ensuring that it can never enter an invalid sate by controlling access to these objects [13]. Figure 7 provides a high-level Class Diagram of each of each Billings Aggregate.

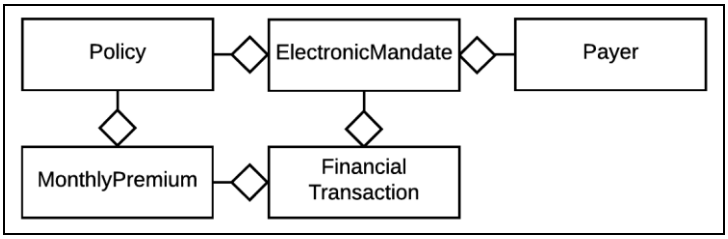


Figure 6: Simple Class Diagram for Billings Aggregates

Aggregates do not contain direct references to other Aggregates as this would create an interconnected graph of dependant objects which would defeat the purpose of having each Aggregate act as an independent consistency boundary [13]. Each Aggregate type has a matching Aggregate Identity [13] type which is used to uniquely identify it; for example, the Class Diagram in Figure 8 shows that each *MonthlyPremium* Aggregate is uniquely identified by a *MonthlyPremiumId* object. Interactions between Aggregates are coordinated by Domain Services [13] which use the Aggregate Identity types to retrieve Aggregates from Aggregate Repositories [13].

6.2.1 Billings Domain Model

The *Policy* Aggregate is uniquely identified by a *PolicyNumber* and contains information which is relevant to the Billings domain such as the current premium and the policy status. The Billings domain is not responsible for policy administration; therefore, *Policy* aggregates are synchronised with changes being made to D3 policy data.

The *Payer* Aggregate represents a person responsible for paying the monthly premium of one or more policies and is uniquely identified by their thirteen-digit South African ID number. The *Payer* Aggregate contains contact information, basic personal details such as name and surname, and a list of *BankAccount* Value Objects. The payer for a policy may be a different individual than the policy holder, however for the prototype stage *Payer* aggregates are synchronised with changes being made to D3 policy-holder data.

The *ElectronicMandate* Aggregate represents the payer's electronic authorisation to debit a monthly premium, for a specific policy, from their nominated bank account. The *Mandate* Aggregate is uniquely identified by an *AuthorisationCode*, a unique value returned by the payer's bank after the electronic mandate process has been completed. The Aggregate contains *BankAccount*, *DebitLimit* and *DebitDay* Value Objects, and *PolicyNumber* and PayerIdNumber Aggerate Identity objects. During the prototype phase, *Mandate* Aggregates are synchronised with changes being made to D3 policy-holder bank-account data.

The *FinancialTransaction* Aggregate tracks the progress of a submitted Authenticated Collections transaction that has been submitted to a bank. The *FinancialTransaction* Aggregate is uniquely identified by a unique *FinancialTransactionId*. During the prototype stage, Authenticated Collections transactions will be processed by a Service Stub [14] instead of being submitted to an actual bank.

The Class Diagram in Figure 8 shows the detailed design of the *MontlyPremium* Aggregate, it is uniquely identified by a combination of the P*olicyNumber* and the C*alendarMonth* in which the premium is due. A *RaiseMontlyBilling* Command Message is executed through a batch process at the beginning of each month for all active policies. The *RaiseMontlyBillingHandler* uses the *MontlyPremiumFactory* to create an instance of the *MontlyPremium* Aggregate which is persisted by the *MontlyPremiumRepository* to a data store.

7 Project Implementation

The prototype phase of the Billings project is intended to allow the development team to gain experience in the Billings problem domain, evaluate different technology platforms, and uncover hidden complexity within the existing D3 billings process. The exploratory nature of this phase makes it impractical to use a Waterfall [11] development approach as requirements are likely to be emergent and undergo change as knowledge is gained.

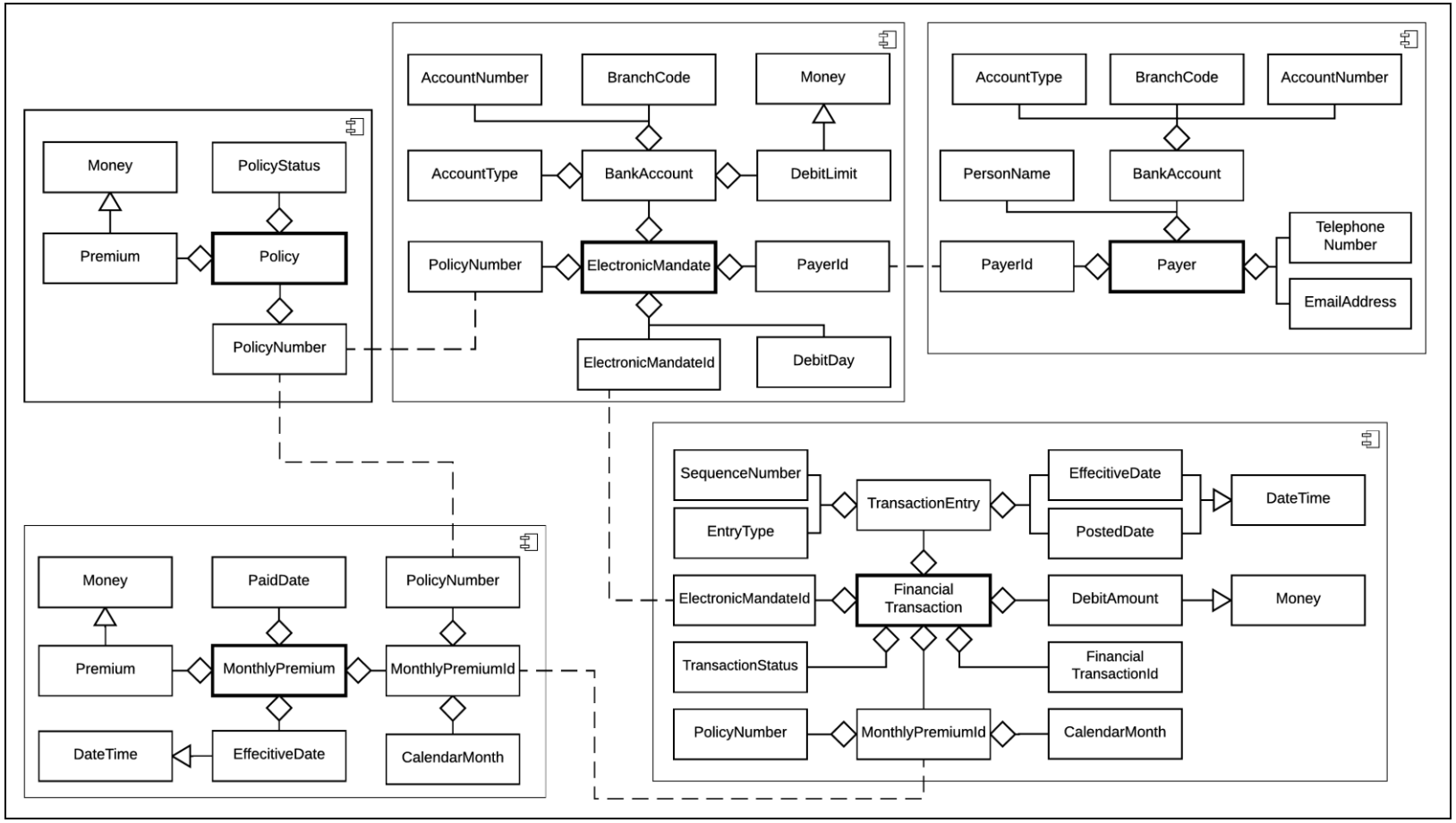


Figure 7: Billings Domain Model

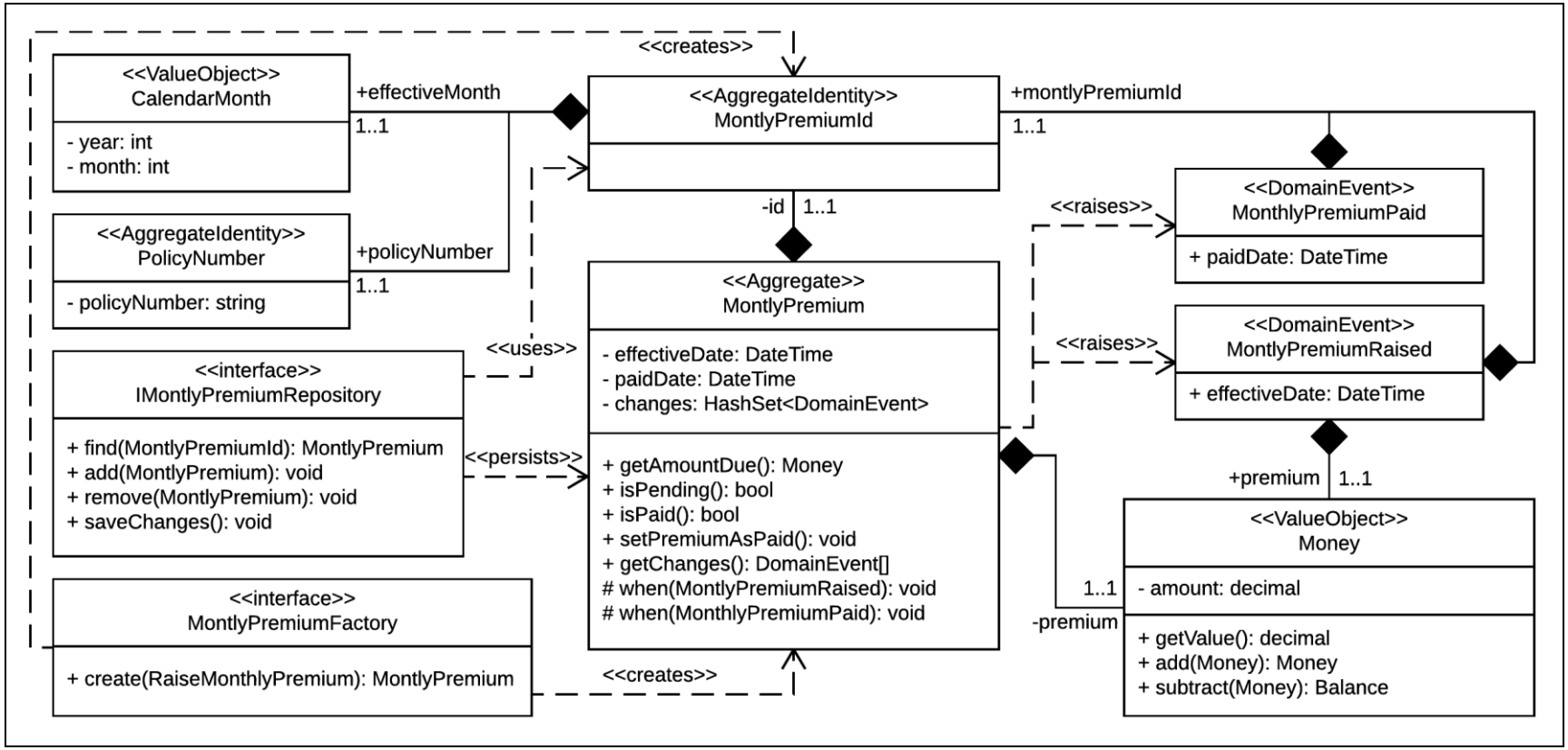


Figure 8: Class Diagram for the Monthly Premium Aggregate

An Agile development approach is more *is most appropriate in cases when there is a need to get operational feedback on an initial capability before defining and developing the next increment's content* [28]. Teams, using a Disciplined Agile [29] approach, will be allowed to self-organise and evolve their own way-of-working, guided by frameworks such as SEMAT Essence [30] and Boehm's Incremental Commitment Spiral Model (ICSM) [28].

> *[Disciplined Agile] is performed in a highly collaborative, disciplined, and self-organizing manner within an appropriate governance framework, with active stakeholder participation to ensure that the team understands and addresses the changing needs of its stakeholders. Disciplined agile delivery teams provide repeatable results by adopting just the right amount of ceremony for the situation which they face [28].*

Requirements for the prototype phase were elicited through wireframe screen-designs and captured in the User Story [12] format. The Planning Poker [31] Agile estimation technique will be used by teams to provide estimates of the level of effort required to implement each User Story. User Stories which are estimated to require more than a week's worth of effort to implement will be broken down into smaller User Stories and re-estimated, this reduces the risk over or under estimating the level of effort required.

Each User Story will be assigned two estimates, a most-likely and a worst-case estimate. By providing two estimates, the level uncertainty or risk associated with the User Story can be inferred by evaluating the difference between the two values; as an example, a User Story with an estimate of 4 to 5 days is considered less risky than one with an estimate of 4 to 15 days [31].

Release Planning [31] will be used to provide a high-level project plan in which User Stories are assigned to each development iteration based on their priority. User Stories will be prioritised by risk, with high-risk items being allocated a higher priority. Implementing high-risk User Stories earlier avoids potentially expensive rework which may be required if the risk materialises at a later stage of the project. The Release Plan is not a static document and is continuously updated as new User Stories are created and priorities change [31].

Configuration Management [32] is required to store and retrieve all artefacts associated with the project. The following software packages have been selected for this purpose:

- JIRA [33] provides a repository for bug reports, User Stories, Release Plans and project documentation.
- Bitbucket [33] is a version control system used to store and retrieve source code.
- Bamboo [33] is a Continuous Delivery [32] system and stores configuration parameters to be used during deployment. Configuration parameters include; database connection-strings, passwords and runtime configuration settings.

Implementation of the prototype Billings system is split across three Feature Teams [29] to reduce the number of new concepts and technologies each team must deal, and enabling

concurrent development of orthogonal features. The Messaging Infrastructure team are responsible for the evaluation of Message-Oriented Middleware platforms and the creation of reusable components which enable systems to send and receive messages asynchronously. The Billings Domain Team are responsible for implementing the Billings Domain Model and Application Services which are the direct clients of this model. The D3 Integration Team is responsible for creating the *D3 Event Publisher* and *D3 Event Integrator* systems.

Separate Feature teams require additional coordination to ensure that they are working toward a common goal. Daily Stand-up [34] meetings involving all teams will be held to ensure that everyone is kept up-to-date with progress being made. Bi-weekly Retrospective [34] meetings will provide an opportunity for the teams to evaluate their current way-of-working and openly address any dysfunctions within the team.

Requirements and the existing design are likely to change as knowledge is gained about the problem domain, potentially requiring sections of the code base to be Refactored [35] to provide a more expressive model or modify existing functionality. Teams will be required to use Test-Driven Development [35] and Continuous Integration [32] to ensure that systems are comprehensively covered by automated Unit-Tests [35].

8 Critical Review

This section evaluates whether the requirements specified in section 3.2 have been met and critically evaluates the design. Examples are given of how Object-Oriented Design [36] principles have been applied to create a flexible and extensible design.

8.1 Validation of Requirements and Constraints

The requirement in section 3.2.1 specifies that the policy payer should be treated as a different entity than the policy holder. The requirement is met through the creation of the *Payer* Aggregate which represents the person responsible for paying the monthly premium of one or more policies.

The requirement in section 3.2.2 specifies that the system must provide a consolidated view of the premium collection status for all policies paid for by a client. The *ElectronicMandate* Aggregate links a *Payer* to a *Policy*, making it possible to select all *ElectronicMandate* objects which are associated with a *Payer*. *PolicyNumber* objects referenced by the retrieved *ElectronicMandate* objects are used retrieve *MontlyPremium* Aggregates relating to the current calendar month. The retrieved *MontlyPremium* objects are displayed to the user as show in Figure 9 in Appendix A – Screen Designs.

The requirement in section 3.2.3 specifies that all debits made to a bank account must be associated with a valid electronic mandate. The requirement is met by having each *FiancialTransaction* require a reference to a *AuthorisationCode* object which identifies an *ElectronicMandate* Aggregate. The *ElectronicMandate* Aggregate contains an immutable *BankAccount* Value Object which is used at runtime to determine which bank account the *FiancialTransaction* relates to, this makes it impossible to create a *FiancialTransaction* in the absence of an *ElectronicMandate*.

The requirement in section 3.2.4 specifies that delays in the electronic mandate process must not affect policy sales. The *Policy* and *Payer* Aggregates can be created independently of each other; this allows the *ElectronicMandate*, which links a *Payer* to a *Policy*, to be created once the asynchronous electronic mandate process is completed. The requirement is met as a Sales Agent can continue with the next sale without having to wait for the client to authorise the electronic mandate with their bank.

The constraint in section 3.3 requires the Billings System to integrate with D3 without affecting or requiring changes to existing D3 functionality. The Bubble Context approach allows the Billings System to be created on a new platform outside of D3, while the *D3 Event Publisher* and *D3 Event Integrator* ensure that neither D3 or the Billings System are directly coupled to each other.

8.2 Application of Object-Oriented Design Principles and Design Patterns

The SOLID OOD Principles [36] and OO Design Patterns [22] were used to create a design which is extensible and maintainable. Examples of how these principles were applied are given below.

The *CommandDispatcher* favours composition-over inheritance [37], with dependencies being resolved at runtime through constructor-based dependency injection [24], enabling the behaviour of the *CommandDispatcher* to be modified without requiring changes; this adheres to the Open Closed Principle [36] and Dependency Inversion Principle [36].

A *CommandDispatcher* subclass can be registered as an Observer [22] on a *MessageReceiver* instance, this adheres to the Liskov Substitution Principle [36] which states that *derived classes must be substitutable for their base classes.*

The *CommandDispatcher* uses *MessageTranslator* objects to convert received messages to *Command* objects. Each *MessageTranslator* implements a different conversion Strategy [22] and is responsible for converting a single Event Message type to a single *Command* object type, this adheres to the Single Responsibility Principle [36].

All interfaces in the Anticorruption Layer and Domain Model expose fine-grained contracts focusing on related functionality, this adheres to the Interface Segregation Principle [36].

The Observer, Repository and Service Locator patterns were used to decouple the Anticorruption layer and Domain Model from infrastructure specific code, enabling Unit-Test to replace these dependencies with Test Doubles [38] and allowing concurrent development of orthogonal features.

8.3 Critique

Keeping Aggregates decoupled from each other by referencing Aggregate Identity objects complies with the Law of Demeter [36] as each Aggregate can only make decisions based on its internal information and is unable to invoke functionality on another Aggregate. Aggregates can be developed independently of each other and Unit-Testing is simplified as Test Doubles [38] of Aggregates are not required. A disadvantage of this approach is

that immutable Aggregate Identity classes must be created to represent each Aggregate type.

Decoupled Aggregates makes it possible to directly store each Aggregate in a Document Database [39]. An interconnected web of Aggregate objects would result in an entire object graph being persisted and retrieved as a single Serialised Large Object [14], blurring the consistency boundaries of each Aggregate. As an example; retrieving a *ElectronicMandate* Aggregate would also retrieve the *Policy* and *Payer* Aggregates if they were not decoupled.

Using asynchronous messaging for integration between the *D3 Event Publisher* and the *D3 Event Integrator* introduces additional complexity to the system as this requires the team to evaluate and integrate with Message-Oriented Middleware. Remote Procedure Calls (RPC) [18] made through Web Services [18] provide a more familiar option, however this would tightly couple the *D3 Event Publisher* to the *D3 Event Integrator*. RPC calls made by the *D3 Event Publisher* to the *D3 Event Integrator* would fail if the system is temporarily unavailable, requiring the *D3 Event Publisher* to implement complex retry logic.

The Command [22] pattern encapsulates the parameters, dependencies and behaviour required to complete a request. Using the standard Command pattern would require a *MessageTranslator* class to be responsible for translating an Event Message and instantiating the Command's dependencies, violating the Single Responsibility Principle. The design for the Anticorruption Layer has separate *Command* and *CommandHandler* objects. *Command* objects are Data Transfer Objects (DTO) [14] which represent a request and contain no behaviour, while *CommandHandler* objects encapsulate the behaviour and dependencies required to complete the request. The separation ensures that a *MessageTranslator* class is only responsible for translating an Event Message into a *Command* object, with the instantiation of the *CommandHandler* and its dependencies being deferred to a Service Locator.

Using the Domain-Driven Design concept of a Bubble Context and an Event-Driven Architecture [16] allows the Billings System to evolve and mature over time without being directly coupled to, or affecting existing D3 functionality, however this approach poses several challenges:

- The *D3 Event Publisher* must be running to detect changes occurring within the D3 file system, downtime experienced by this service will potentially result in Billings Bubble Context going out of sync with D3.
- The *D3 Event Publisher* will only publish Event Messages for new or updated policy files, the Bubble Context will therefore not contain information for most existing policies.
- Downtime experienced by the D3 Event Integrator will potentially result in the Bubble Context's data reflecting outdated information.
- Translating data from the D3 model to the Billings Domain Model may be complicated by corrupt or missing data which the Billings Domain Model

requires; for example, The *Payer* is uniquely identified by a valid South African Identity Number, however in some cases the current policy-holder's Identity Number may be missing or invalid.

9 Conclusion

The DDD concept of a Bounded Context is shown to provide an alternate approach to rewriting or refactoring legacy systems. The Billings subdomain is remodelled into an explicit Domain Model within a Bubble Context, separated from the D3 model through an Anticorruption Layer, allowing the new model to evolve independently and unconstrained by legacy concepts. Legacy functionality is unaffected by the new model and co-exists with the Bubble Context, allowing the new model to mature over time until it meets all specified requirements.

The Domain-Driven Design approach is not without challenges. Data consistency between D3 and the Billings Bubble Context, and translating conceptual objects and actions from one model to another are problematic due to technology and data quality limitations. The prototype stage is intended to uncover such problems to acquire knowledge which will guide the design of future project phases.

A project implementation approach has been presented in which Feature Teams, using a Disciplined Agile approach, will be allowed to self-organise and evolve their own way-of-working. Self-organisation is guided a minimum set of prescribed Software Engineering practices, and frameworks such as SEMAT Essence and Boehm's Incremental Commitment Spiral Model.

The design presented in this paper provided a detailed view of the Anticorruption Layer and the Monthly Premium Aggregate, with additional work required to provide a detailed design of other Aggregates and the D3 Event Publisher.

References

[1] Payments Association of South Africa, “Debit Payments,” Payments Association of South Africa, 2016. [Online]. Available: http://www.pasa.org.za/about-payments/intro-to-payment-streams/debit-payments. [Accessed 17 June 2016].

[2] J. Young, M. Dubois and O. Trigubets, “Authenticated Collections,” July 2016. [Online]. Available: https://home.kpmg.com/content/dam/kpmg/pdf/2016/07/za-authenticated-collections-2.pdf. [Accessed 3 August 2016].

[3] C. Tait, “PASA Launch Authenticated Collections to Curb Debit Order Abuse,” Payments Association of South Africa, 15 March 2016. [Online]. Available: http://www.pasa.org.za/home/2016/03/15/news. [Accessed 10 June 2016].

[4] Technopedia, “Legacy System,” Technopedia, 2016. [Online]. Available: https://www.techopedia.com/definition/635/legacy-system. [Accessed 27 November 2016].

[5] M. Feathers, Working Effectively with Legacy Code, Pearson Education, 2005.

[6] B. Benfield, Life Assuance Company Management, Primary Asset Administrative Services, 2013.

[7] Investopedia, “What is the financial services sector?,” Investopedia, 3 March 2015. [Online]. Available: http://www.investopedia.com/ask/answers/030315/what-financial-services-sector.asp. [Accessed 16 November 2016].

[8] Techopedia, “Mainframe,” Techopedia, 2016. [Online]. Available: https://www.techopedia.com/definition/24356/mainframe. [Accessed 27 November 2016].

[9] R. Pick, “An overview of the Pick Operating System,” in *AFIPS Proceedings of the 1987 National Computer Conference*, Chicago, Illinois, 1987.

[10] Pick Systems Inc., Pick Systems Reference Manual, Pick Systems Inc., 1997.

[11] H. Van Vliet, Software Engineering: Principles and Practice, John Wiley & Sons Ltd., 2012.

[12] M. Cohn, User Stories Applied: For Agile Software Development, Addison Wesley, 2004.

[13] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, 2004.

[14] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003.

[15] E. Evans, “Getting Started with DDD When Surrounded by Legacy Systems,” April 2016. [Online]. Available: http://domainlanguage.com/wp-content/uploads/2016/04/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf. [Accessed 3 August 2016].

[16] M. Chandy and R. Schulte, Event Processing: Designing IT Systems for Agile Companies, McGraw-Hill, 2010.

[17] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2004.

[18] R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2012.

[19] RabbitMQ, "What is RabbitMQ," RabbitMQ, 2007. [Online]. Available: https://www.rabbitmq.com/. [Accessed 1 November 2016].

[20] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2013.

[21] R. Miles and K. Hamilton, Learning UML 2.0, O'Reilly Media, Inc., 2006.

[22] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 2011.

[23] Microsoft Developer Network, "The Service Locator Pattern," Microsoft, 2016 . [Online]. Available: https://msdn.microsoft.com/en-us/library/ff648968.aspx. [Accessed 19 November 2016].

[24] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," MartinFowler.com, 23 January 2004. [Online]. Available: http://martinfowler.com/articles/injection.html. [Accessed 2016 April 2004].

[25] Autofac, "Autofac," autofac.org, 2013. [Online]. Available: https://autofac.org/. [Accessed 20 April 2016].

[26] Microsoft Developer Network, "Generics (C# Programming Guide)," Microsoft, 20 July 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/512aeb7t.aspx. [Accessed 19 November 2016].

[27] Microsoft Developer Network, "Using Type dynamic (C# Programming Guide," Microsoft, 20 July 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd264736.aspx. [Accessed 19 November 2016].

[28] B. Boehm, J. A. Lane, S. Koolmanojwong and R. Turner, "The Incremental Commitment Spiral Model," Pearson Education, 2014.

[29] S. Ambler, "The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments," IBM Corporation, 2009.

[30] I. (. Object Management Group, Kernel and Language for Software Engineering Methods (Essence), v1.1, Object Management Group, Inc. (OMG), 2015.

[31] M. Cohn, Agile Estimating and Planning, Pearson Education, 2006.

[32] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison Wesley Professional , 2010.

[33] Atlassian, "Tools for teams, from startup to enterprise," [Online]. Available: https://www.atlassian.com/. [Accessed 5 November 2016].

[34] K. Schwaber and J. Sutherland, "The Scrum Guide," July 2016. [Online]. Available: www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf. [Accessed 22 November 2016].

[35] S. Freeman and N. Pryce, Growing object-oriented software, guided by test, Boston: Pearson Education, 2010.

[36] R. C. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall, 2006.

[37] S. Lowe, “Composition vs. Inheritance: How to Choose?,” ThoughtWorks, 12 May 2015. [Online]. Available: https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose. [Accessed 10 June 2016].

[38] M. Fowler, “Test Double,” martinfowler.com, 17 January 2006. [Online]. Available: http://www.martinfowler.com/bliki/TestDouble.html. [Accessed 21 June 20116].

[39] P. Sadalage and M. Fowler, NoSQL Distilled: A Brief Guide to the World of Polyglot Persistence, Addison-Wesley, 2013.

Appendix A – Screen Designs

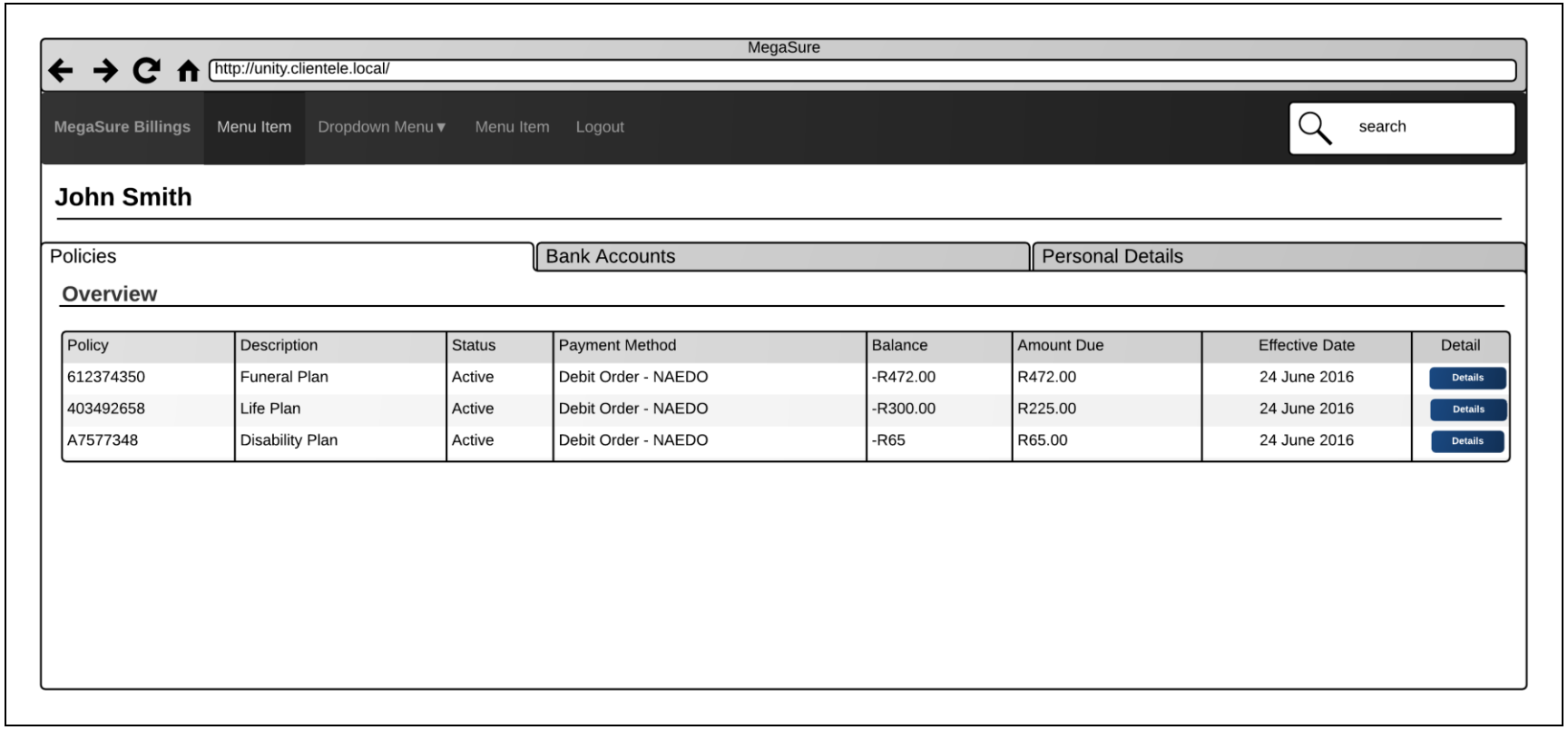


Figure 9: Customer-level billings screen design.

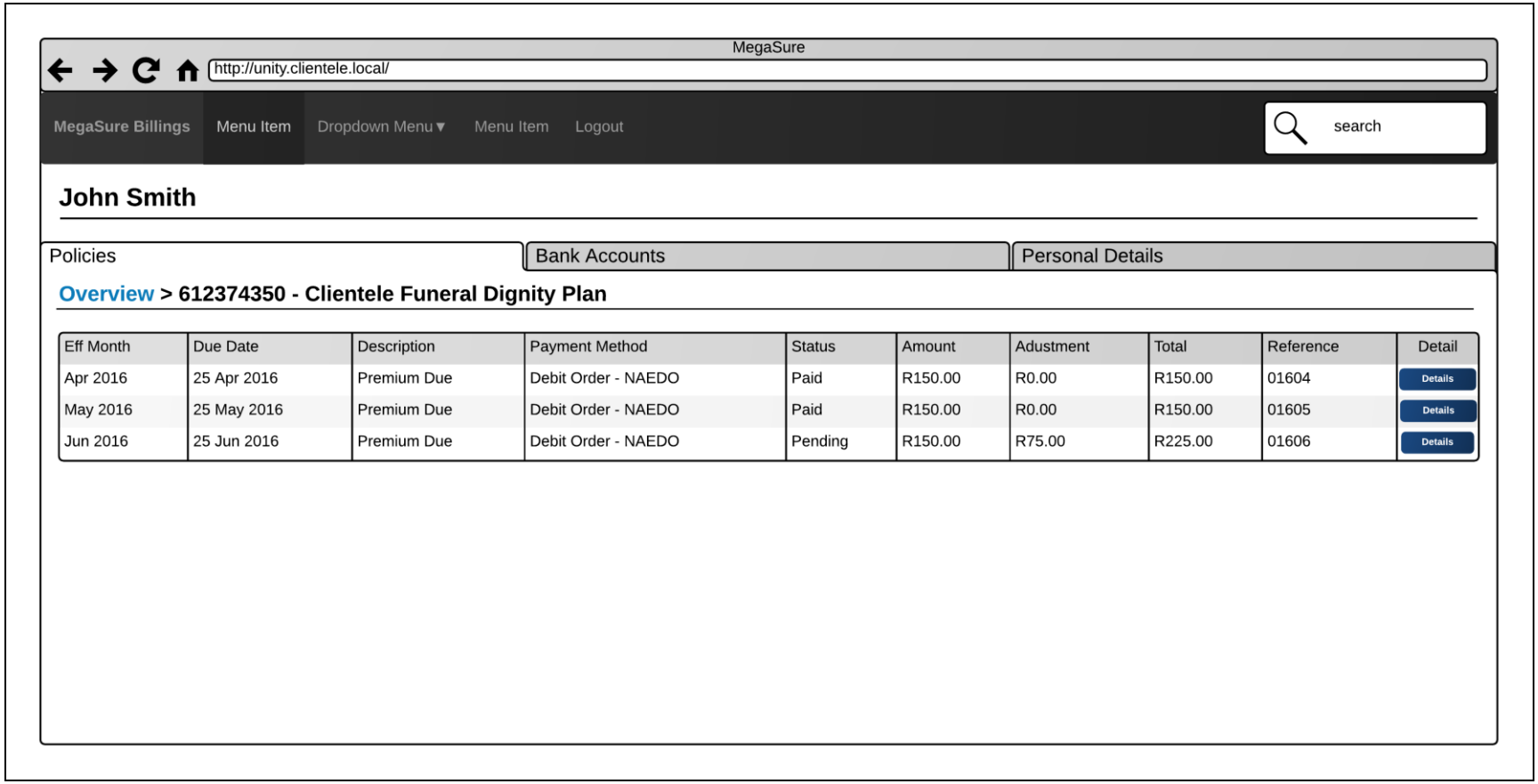
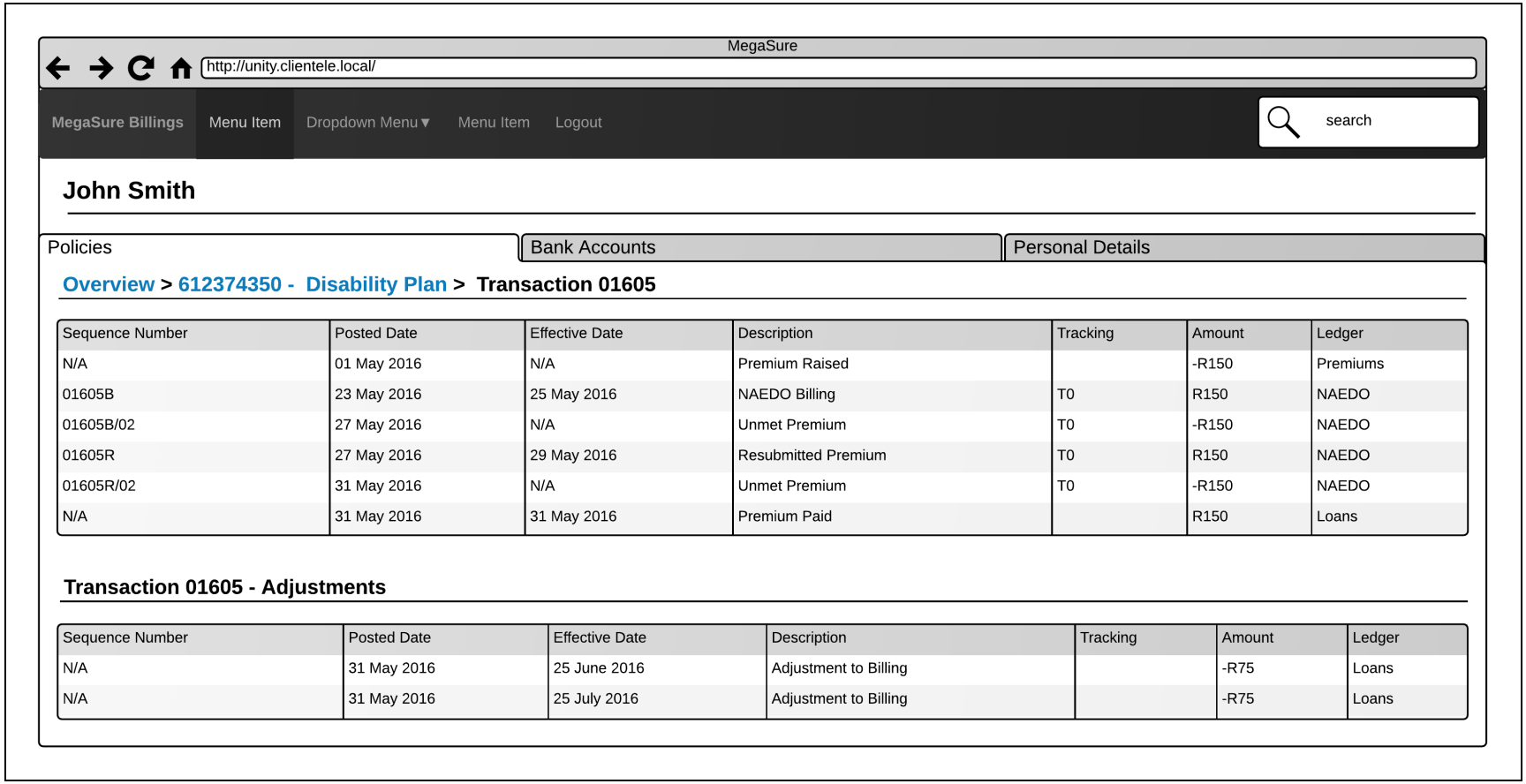


Figure 10: Policy-level billings screen design.



Sequence Number	Posted Date	Effective Date	Description	Tracking	Amount	Ledger
N/A	01 May 2016	N/A	Premium Raised		-R150	Premiums
01605B	23 May 2016	25 May 2016	NAEDO Billing	T0	R150	NAEDO
01605B/02	27 May 2016	N/A	Unmet Premium	T0	-R150	NAEDO
01605R	27 May 2016	29 May 2016	Resubmitted Premium	T0	R150	NAEDO
01605R/02	31 May 2016	N/A	Unmet Premium	T0	-R150	NAEDO
N/A	31 May 2016	31 May 2016	Premium Paid		R150	Loans

Sequence Number	Posted Date	Effective Date	Description	Tracking	Amount	Ledger
N/A	31 May 2016	25 June 2016	Adjustment to Billing		-R75	Loans
N/A	31 May 2016	25 July 2016	Adjustment to Billing		-R75	Loans

Figure 11: Transaction-level billings screen design.

Appendix B – Raise Monthly Billing Sequence Diagram

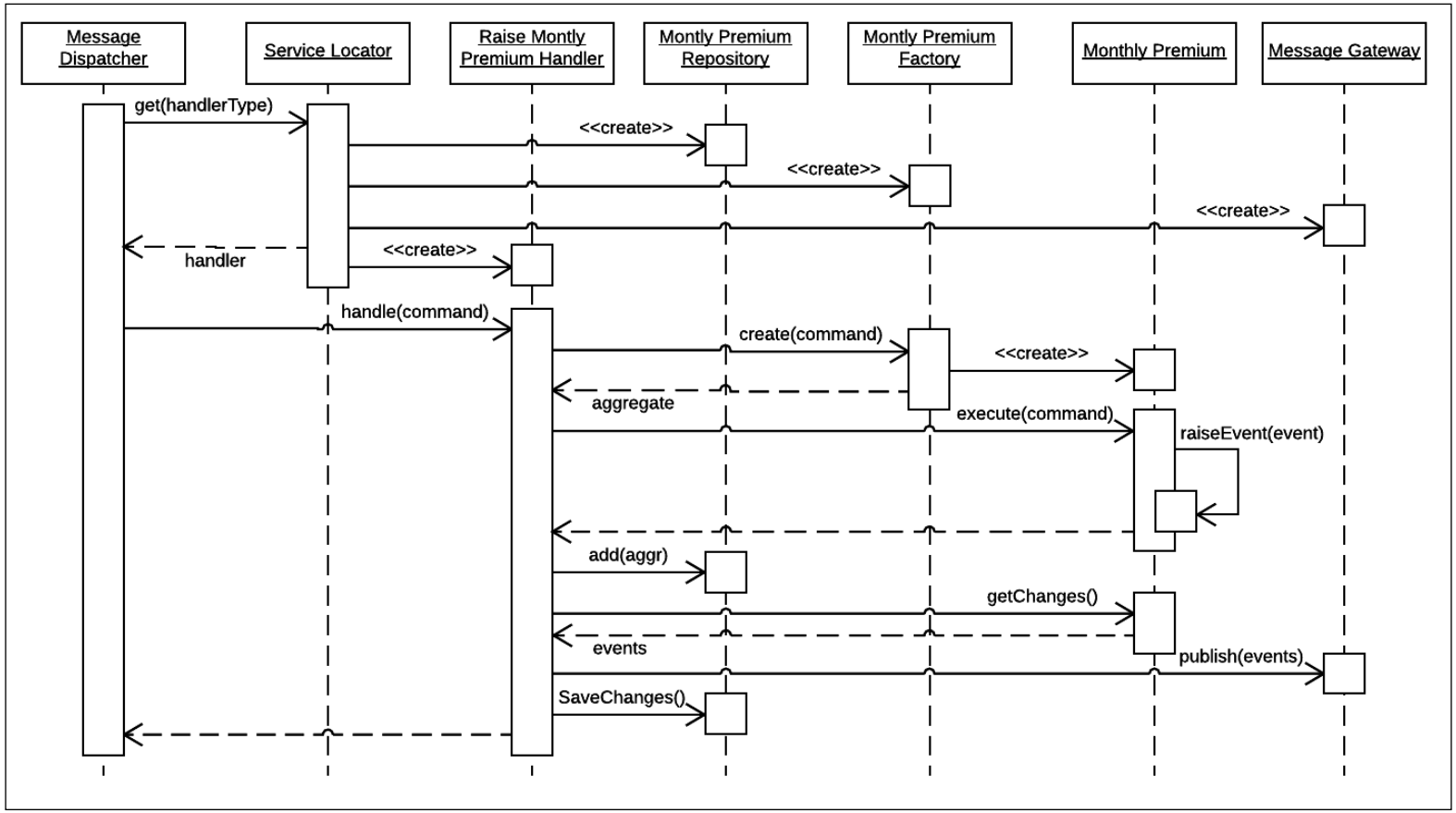


Figure 12: Sequence Diagram for the Raise Monthly Premium command handler