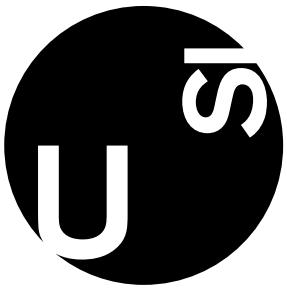


Università della Svizzera italiana

Faculty of Informatics

Lifeware
life insurance software service

Bachelor Thesis

June 17, 2024

Inspecting Objects — Then, There, Here and Now

Real-time round-trip Glamorous Toolkit views for industrial web-development

Kyla Kaplan

Abstract

Glamorous Toolkit (GT) [1], based on the Pharo [2] core environment, provides a rich ecosystem for software development in Smalltalk. GT's programming philosophy of Moldable Development™ [3] enables a uniform environment made of visual and interactive operators that can be combined inexpensively in a variety of ways to create quick and efficient tools that enable developer productivity.

This project aims to create an additional interactive and visual support layer for a Smalltalk-based web application within an industrial context to enable the object-oriented approach for inspecting objects within the browser with the help of Glamorous Toolkit. It will also delve into a case-study discussion on different programming paradigms in terms of their efficacy in legacy coding projects. We conducted this project jointly with the partnership of the life-insurance software company Lifeware SA [4] running a legacy CRM system based on VisualWorks' Smalltalk dialect, and the GT development team @ feenk.com.

Advisor
Prof. Michele Lanza
Co-advisor
Dr. Vincent Blondeau

Advisor's approval (Prof. Michele Lanza): Date:

Contents

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. Michele Lanza, for his storytelling of first-hand experiences, unwavering support, and insightful feedback throughout this project. His expertise and encouragement have been instrumental in helping shape this work.

I extend my heartfelt thanks to the team at Lifeware SA for providing me the opportunity to work on a real-world project during my studies and for their continuous cooperation and support. A particular gratitude towards Dr. Tudor Girba and Dr. Andrei Chis from the GT development team @ feenk.com for their infinite patience, technical assistance and for sharing their knowledge and resources.

Lastly, I wish to acknowledge the continuing open-source communities behind Smalltalk that keep the language alive. Without their dedication and resourceful contributions, this project would not have been possibly.

Thank you to everyone for your support!

1 Introduction

Many legacy systems permeate the software development field to this day, and, by virtue of its being, will continue to exist insofar as developers continue to build programs and software systems. “Legacy” systems are defined as such due to a multitude of factors — including their high cost for future maintainability and steep learning curves as time progresses and newer technologies are introduced. Oftentimes, the “technical expense” to migrate an older technical stack outweighs the costs of simply re-implementing similar modern tools directly as time goes by.

Within the current day software engineering field, the cultural significance of legacy systems permeates the older generations of developers, but foregoes the newer generation as it is “looked down upon” for the lower odds of providing a robust technical stack for most current-day end-user/consumer needs, as well as short career growth opportunities. Additionally, there exists a vast divide for the support and “compatibilities” between these systems. Many tools, bridges, and ports to other systems end up never being developed, or never gaining enough support; whilst the current open-source versions of these systems, lacking said support, become less and less maintained, and eventually completely archived.

As a prime example, during the Covid-19 pandemic, the governors of New Jersey and Kansas issued public pleas for volunteers with any level of COBOL proficiency. This "40-plus-year-old" computer language underpins the Department of Labor unemployment benefits system, which urgently required patching due to a surge in unemployment rates and subsequent system crashes [5]. Due to the rapid technological inventions and advancements of the last century, within a world which continues to rely heavily on large-scale software infrastructures, we are first to witness the evolution of a crisis situation unfold because of the mass extinction of a generation of specialists in these direly-needed fields. According to an article published by Mechanical Orchard, "on average, 31% of an organization’s technology is made up of legacy systems", and "60-80% of IT budgets are allocated to keeping them running" [6].

The main focus of this project is contained within the context of software in an "industrial" application — on systems that are built and age with legacy code. Continuing forward in this report, I will be narrowing the scope of the discussion about legacy programs within an industrial context surrounding software systems built solely in and for Smalltalk.

1.1 A Brief History of Smalltalk — Then & Today

Prior to understanding the industrial context of this project, it is first important to note the significance and evolution of Smalltalk as a programming language. Smalltalk-80 — the first publicly available and most notable version of Smalltalk, was introduced back in 1980 at Xerox Palo Alto Research Center (PARC) under the guidance of Alan Kay and Dan Ingalls [7]. It became an influential programming language due its embodiment of object-oriented programming (OOP) principles and its capacity for enabling flexible and expressive programming paradigms; including, but not limited to: dynamic typing, object message passing, and more. The main driving philosophy behind Smalltalk, is that **everything** is an object. Each object is an instance of a class; each class is an instance of the metaclass of that class. The core fundamental theories of OOP such as polymorphism, inheritance and encapsulation were all first seen in Smalltalk-80. Additionally, it also introduced the MVC (model-view-controller) design pattern [8].

Most importantly, the Smalltalk development environment at the time featured an intuitive graphical user interface with live-coding capabilities, allowing for a groundbreaking iterative programming experience.

VisualWorks [9], originally introduced by ParcPlace systems, became the most prominent

Smalltalk development environment due its comprehensive set of tools, and more specifically its intuitive debugger and object inspectors. This revolutionized programmers abilities to be able to interact live with program objects, and stack trace the system during execution; with an even more interesting application once extrapolated to web-based technologies (for which it was not initially designed for, but rather client-service platforms). Unfortunately, VisualWorks became a solely proprietary software with high licensing costs and fewer new features developed over time, thus sheltering it from smaller development teams with smaller budgets, and therefore failing to gain community support against other emerging enterprise computing trends.

Despite the potential influence it could foresee, Smalltalk forewent a widespread adoption in commercial applications, apart from the academic and research community. Subsequently, Smalltalk heavily influenced the next generation of modern programming languages, such as Java and Ruby, which nowadays endure as a vehicle for the foundational model for object-oriented design principles.

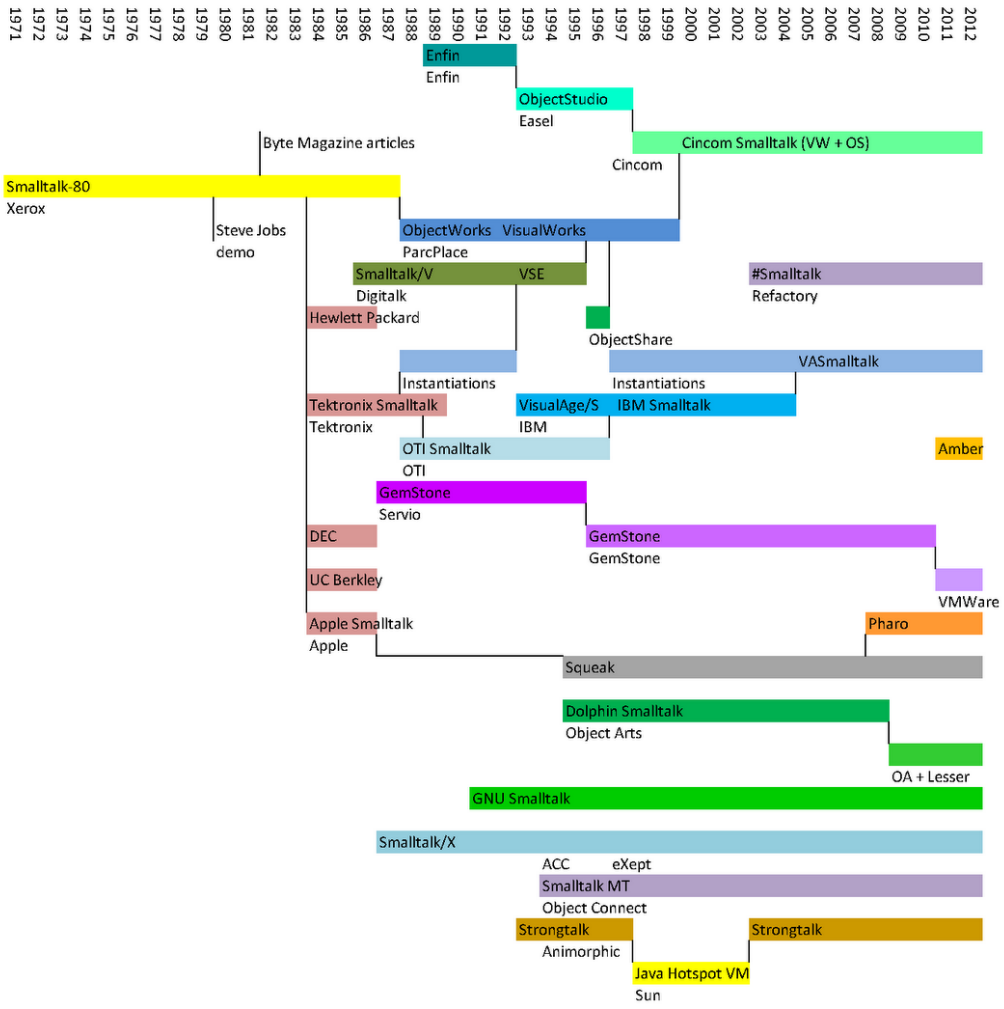
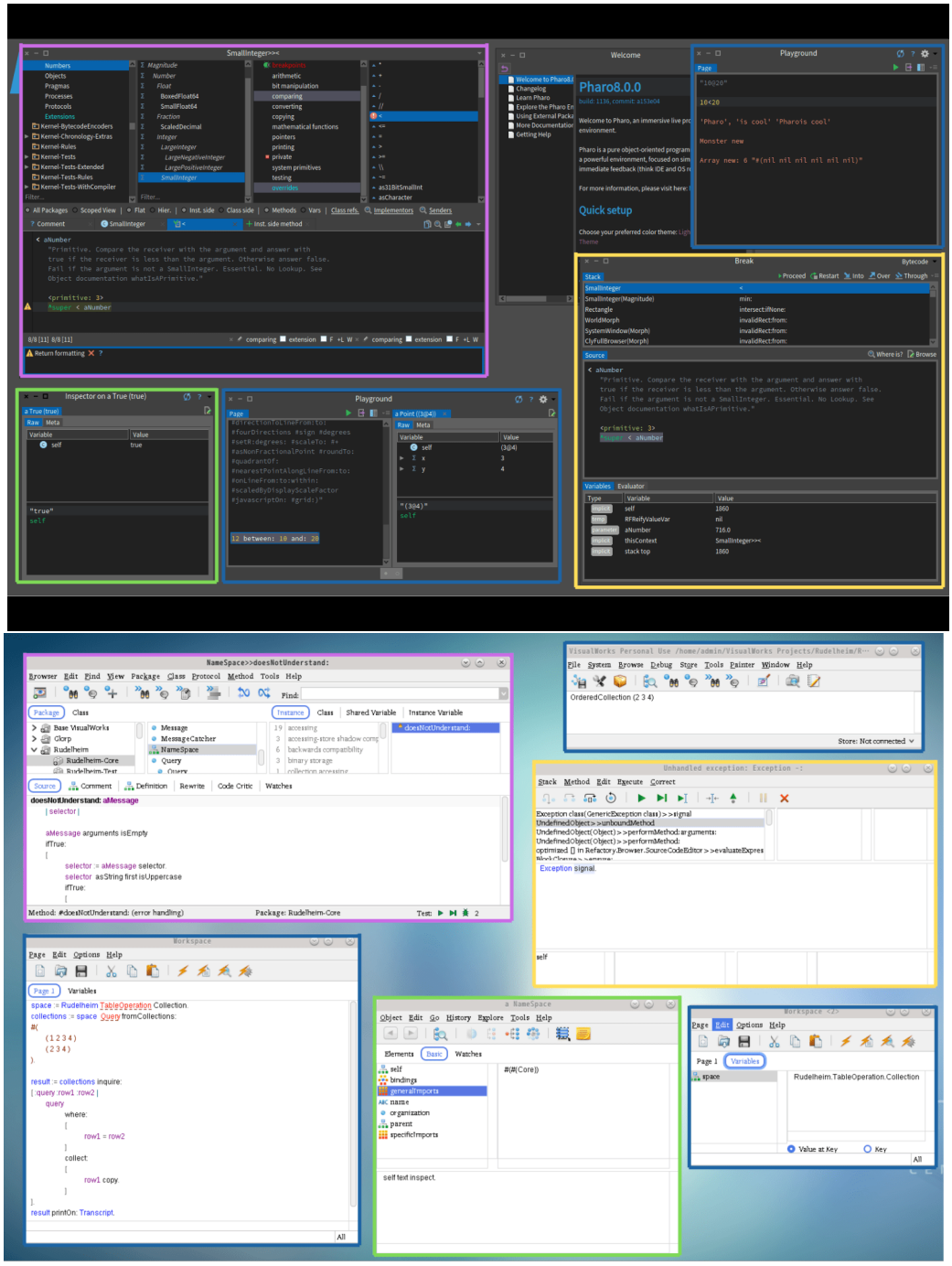


Figure 1. A brief history of Smalltalk dialects (from 1971 to 2012)[10]

Through the early aughts, Smalltalk saw itself forked into several different dialects: Squeak, Dolphin, and more. Figure 1 shows a brief timeline of some of the branches of the Smalltalk dialects. Most notably, Pharo (as a fork off of Squeak) came to dominate the scene, led by a group of

Smalltalk enthusiasts driven by the mind of Stéphane Ducasse at INRIA, who sought to revive the slowly-evaporating Smalltalk ecosystem (even to this day) [2]. With Pharo, a vibrant community re-emerged utilizing the full-fledged capabilities of the language, a faster just-in-time compiler (JIT), as well as possessing a whole new suite of third-party libraries and frameworks. Particularly, Pharo iss open-source and easily-accessible to newcomers with an abundance of resources on approaching OOP in Smalltalk.

Visually, Pharo follows a very similar-looking programming environment for developers. Below is a screenshot of both of the IDE's layouts, as well as their debuggers and object inspectors.



Source: VisualWorks [9]

Figure 2. Highlighted differences in IDE's (Pharo: top, VisualWorks: bottom)

Even without looking closely, one can see that, although Smalltalk diverged under many different influences, and although the time between the creation of VisualWorks and Pharo spans

almost 20 years, they are visually similar. They consist of a "world" in which each window and tab represents a way for the developer to "approach" the system. Trying to mimic both of the IDE's, the legend for Figure 2 is such:

- purple indicates a System Browser window, allowing for creating/editing methods (messages) and exploring classes, packages and protocols
- blue are Playgrounds/Workspaces are a free-form typing of Smalltalk expressions and executing them
- green are Object Inspectors, which include a "variable pane", and a "object/class/meta view"
- yellow highlights the Live Debugger, which contains a "stack trace" to trace execution behaviour, along with a "variable pane", a "source view" allowing to step in, through and over code, while allowing developers to inspect objects along the way

An important key feature of Smalltalk is its reflective property, which was defined in 1993 by Bobrow, Gabriel & White [11] in their paper "The Shape of the Design Space" on OOP programming practices: "*Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution*", following that "*Reification is the mechanism for encoding execution state as data*". Figure 3 shows an intuitive way to visualize it. The way these Smalltalk IDE's are formed allows for the programmer to "interject" the system at run-time, and modify the program, whilst "reflecting" the object, or even the state of the system, back to the Virtual Machine (VM). This is a big reason why the "object inspectors" and "live debuggers" visually look the way they do; their main use is to exploit the "reflection" and "reification" properties of the language, by allowing the user to "intercess" the program at run-time.

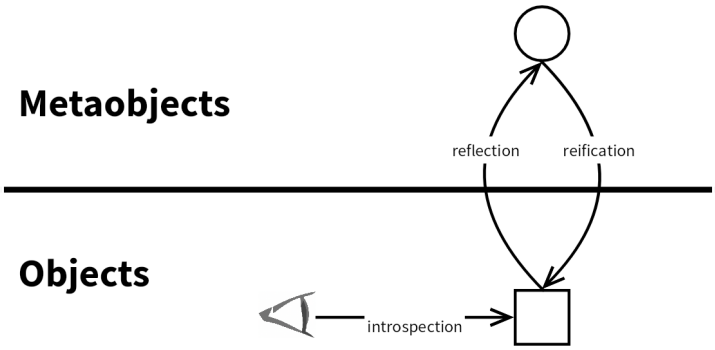


Figure 3. Reflection and Reification [12]

As a short addendum to the history of Smalltalk — the Smalltalk VM is the fundamental basis on which Smalltalk runs; consisting of an interpreter, bytecode/JIT compiler, garbage collector and process scheduler. The entire state of the Smalltalk environment is written to an image file, which can be saved and restored, and thus, be ported across different operating systems and platforms.

> "*Most of the existing object-oriented systems that permit meta-level system manipulation, have been constructed using Lisp-based metacircular interpreters. Reflection is then implemented by modifying the language's interpreter. Smalltalk-80, on the other hand, uses a virtual machine. Although the Smalltalk-80 virtual machine is a byte code interpreter that is usually implemented in machine language or C, the official definition of the virtual machine is written in Smalltalk. Moreover, the Smalltalk-80 debugger uses a byte-code interpreter written in Smalltalk. Indeed, debugging and tracing are frequently cited as applications that are well addressed using reflection.*" [13]

1.1.1 GemStone

Although more commonly there exist Relational Database Management Systems (RDBMS) storing data in traditional rows and columns, for Smalltalk programming an Object-Oriented Database Management System (OODBMS) is needed. In 1982, GemStone was founded by Richard Lamb and Mike Stonebreaker, who developed the first version of a distributed object management system focused on handling complex data relationships typical in object-oriented programming.

GemStone distinguishes itself from other systems due its high-level programming environment based on Smalltalk, unique in its domain application. Particularly, in GemStone objects can be made persistent, meaning their state (data) is stored in the database and survives beyond the application's runtime. Persistent objects are stored in a way that they can be retrieved, updated, and deleted just like traditional database records, but with the added flexibility of object-oriented interactions such as the aforementioned live inspection and debugging features. Each persistent object has a unique identity (OID - Object ID), which differentiates it from other objects. Relationships between objects can be naturally modeled using references (pointers) rather than foreign keys, making it easier to navigate complex data structures, especially those in an industrial context. The system provides mechanisms for versioning objects, allowing for historical data tracking and auditing changes over time. Concurrency control ensures that multiple users can work with the database simultaneously without conflicts, often implemented using optimistic or pessimistic locking strategies.

Just like all Smalltalk dialect VM's, GemStone supports automatic garbage collection for better resource allocation. It also supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, allowing for multiple operations on persistent objects to be executed as a single unit, maintaining database integrity even in the case of errors or system failures. Since it can support several terabytes of data, it makes it perfect for use in industrial applications [14].

1.1.2 Glamorous Toolkit

The main solution of this project is developed in and for Glamorous Toolkit (GT), and as such, it is crucial to understand how GT came to be an important part of the story of Smalltalk in the present-day.

Fast forward through the 1990's, when in 2002, Dr. Tudor Girba moved to Switzerland to pursue his research in software engineering productivity. At the time, he joined the Software Composition Group at the University of Bern under Oscar Nierstrasz, and joined efforts with Stéphane Ducasse and Michele Lanza on Moose, a platform for software and data analysis [15]. The Moose platform soon became quite influential, and with time, Tudor Girba soon became a core contributor to Pharo along Stéphane Ducasse. In 2011, he forked from the Pharo project and founded the Glamorous Toolkit project. At the time, in a grassroots effort, he began working with sub-divided teams across the globe, in an attempt to build on and re-define the full Smalltalk development experience. [16]

Built on top of the Pharo core, GT is very well-documented, with many resources available as a low entry-barrier for newcomers, and the development team behind it are active daily in releasing bug fixes and features [1]. GT can embed environments or virtual machines of other languages, allowing direct execution, debugging and interaction within the GT environment; apart from Smalltalk, GT supports Python, JavaScript and GraphQL.

Girba's belief that "all software systems should be explainable" is the main inspiration for the new IDE, focused on even more interactive and visual approaches for software development than its predecessors. As is prescient to its name, GT consists of a "toolkit", which contains

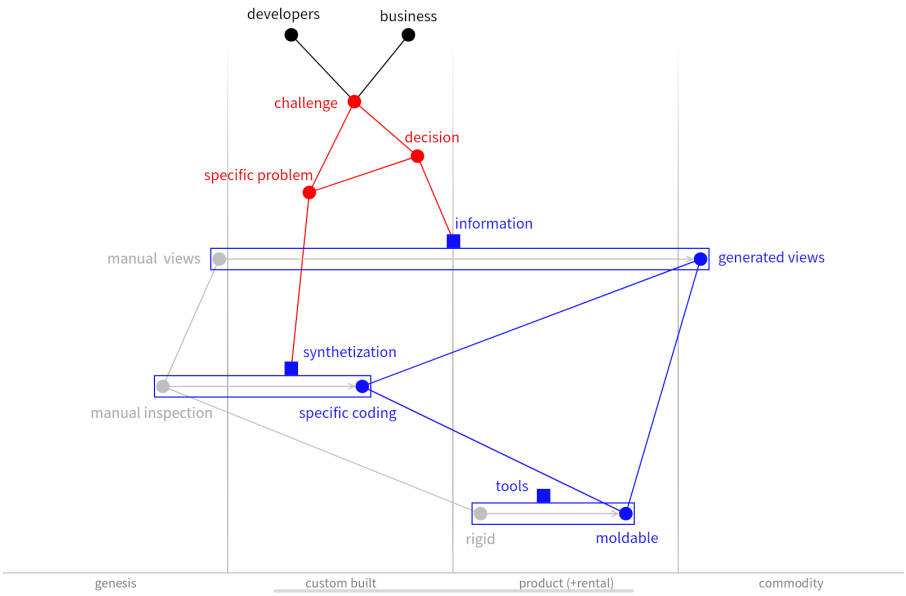
a set of "tools" tailored to specific domains and tasks, and over time, it evolved into a fully-comprehensive environment that emphasizes live and moldable development experiences. Moldable Development™ [3] implies that developers can modify their own development tools and environment "on-the-fly" to better suit their task at hand. This is achieved by means of custom views, actions, searches, examples, and other tools.

Assuredly, aspects of GT's graphical user interface (GUI) incorporates inspiration from its predecessors, however a way in which it is separate is the concept of "Glamorous Views", which are a novel way of visualizing and interacting with data in live programming environments. These views provide developers with customizable and interactive displays of information, enabling them to gain insights into their code, data structures, and system behavior. This key aspect will be a point of further exploration in this project.

Below, in Figure 4, we see a Wardley Map, showing some of the decision-making support that happens on both the technical and business levels of a company.

As we can see from the map, the tools that we build to "make a sense" of these systems are most often contextually rigid, and in the long-term, not economically viable for development (especially in legacy systems). The main argument for Moldable Development is the contrasting way in which it allows for custom tools to be created quickly. One of these tools are the generated views — which are contextual, moldable, and most importantly, quick to build. As Tudor Girba puts it:

> "*These tools offer contextual views that summarize parts of the system from some perspectives. The availability of contextual views on demand changes decision making fundamentally, and with it, the very act of programming. The principle is that whenever a problem is not comfortable enough, you build a tool that makes it comfortable. This can happen multiple times a day even for a single developer. To put it in perspective, in a typical IDE today, you might have dozens of extensions. When practicing Moldable Development, you can easily get to thousands of tools per system* [17]."



Source: From the Glamorous Toolkit Book [17]

Figure 4. Wardley Map of the Moldable Development™ philosophy from the GT Lepiter book

Most developers spend their time reading either code or documentation. In order to tackle any problem on either a technical or business level, a developer must tackle a vast amount of

information in order get a grasp on a system. As a system scales, most often, the grasp of the developer on the system loosens. This key decision-making is made manually, and within very specific contexts and constraints.

In the Moldable Development process, developers with technical expertise create said custom tools, while stakeholders with business or strategic insight ensure these tools address key questions and support actionable decisions. This collaboration establishes a feedback loop where precise, context-rich information guides the development process, leading to more effective and well-informed decisions; a by-product of which are the custom tools themselves, that will likely reap continuous benefits going forward. Generating custom views to replace manual inspection creates a new feedback loop, enabling people to make decisions based on precise information about their system on the spot.

As these custom views will be the focus of the project, in Figure 5, we can see an example of an "Address Book" object in a custom view — specifically the `Contacts with details` view. Here, a developer, with no additional need for understanding the context of the system at this point or visual cues can quickly understand what object is being currently inspected. Besides the Smalltalk code snippet below making most of the context explicit, what we can see in Figure 6 are the `Contacts list` and `Circular` views. If we look at the arrows mapping each contact, we see that it can get confusing if there are significantly more objects and relationships. After these views have been encoded once, they can be reused for the visualization of a `Contact` object, and added with further views if necessary. Below is a snippet of code of a message `gtViewContactsOn:` to which `aView` object is passed to. In this case, the class of `GtABAddressBook` "knows" how to visualize this information in accordance with the `<gtView>` pragma (similar to annotations in Java). An in-depth explanation for `gtViews` can be found further in Section 3 of this report.

```
1  GtABAddressBook >> gtViewContactsOn: aView
2      <gtView>
3      ^ aView columnedList
4        title: 'Contacts with details' translated;
5        priority: 5;
6        items: [ self contacts ];
7        column: 'Avatar'
8            icon: [ :aContact | aContact avatar asElement asScalableElement size: 32 @ 32 ]
9            width: 75;
10       column: 'Name' text: [ :aContact | aContact fullName ];
11       column: 'Phone' text: [ :aContact | aContact telephone ]
```

A developer, accompanied with these building blocks for understanding a system, as well as these custom `gtViews`, can be more efficient in discovering the context of the system, in turn being more efficient in his development process.

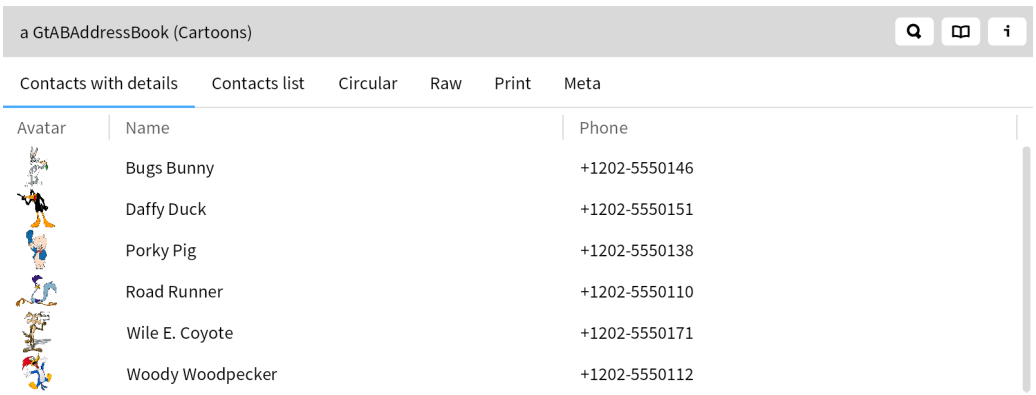
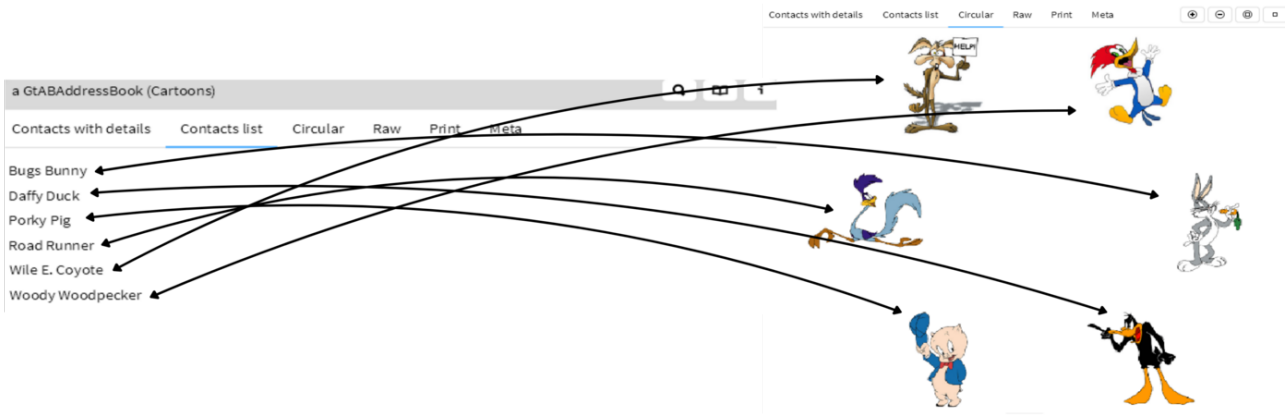


Figure 5. Example of a custom view for an "Address Book" object

Each object is accompanied by its `Raw`, `Print` and `Meta` default views. While the `Meta` view is a

way to access the class definition of the object, the `Raw` and `Print` views show the more "familiar" ways of inspecting objects. In Figure 7, we can see the `Raw` view of the object which in turn carries the variables and values of the inspected object in a tabular format. These "simple" views will be further referenced in this report as "implicit views". These implicit views do not allow for a deeper contextual ability to comprehend the meaningful properties that are truly carried by each contect without a much more thorough inspection of the encapsulation of each property.



Source: from the GT Book [17]

Figure 6. Additional examples of custom views for an "Address Book" object

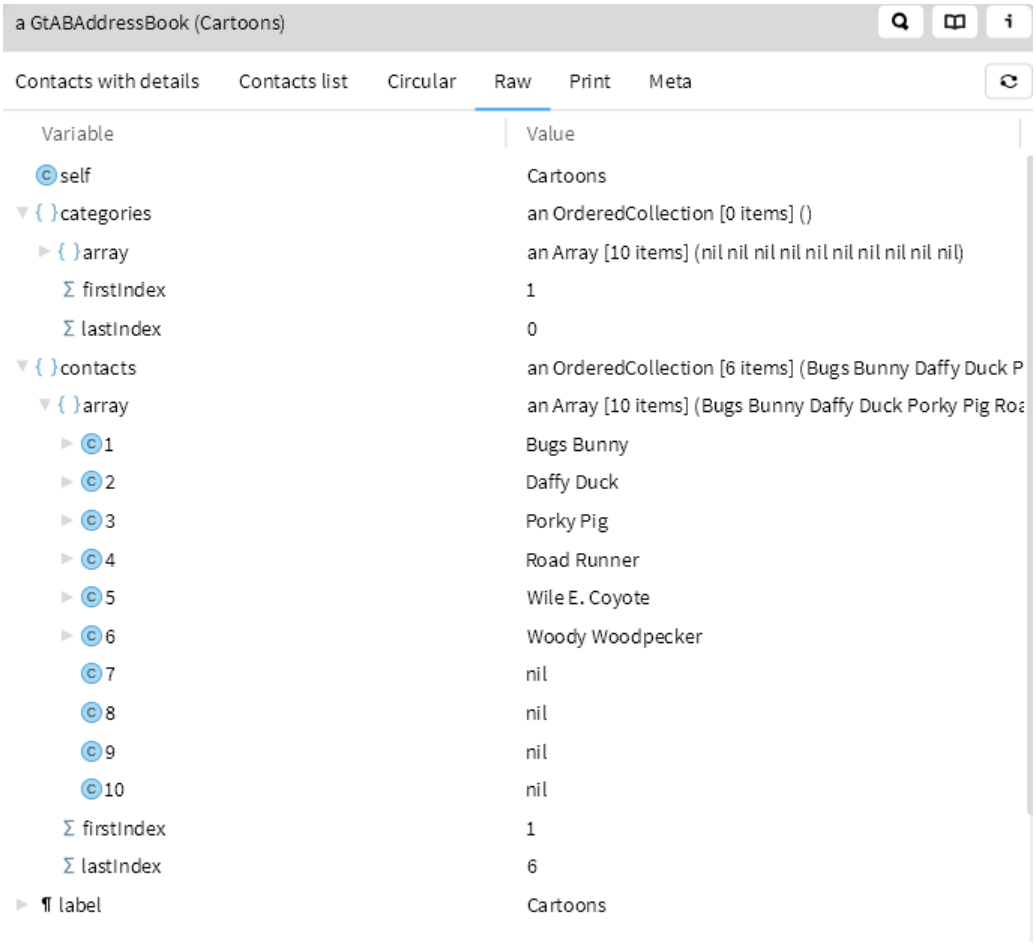


Figure 7. Example of the `Raw` views for an object in GT

1.2 Industry: Life Insurance

1.2.1 Company History

Lifeware SA [4] is a financial technology company providing software solutions for life insurance clients and brokers. The company was founded 25 years ago by Dr. Massimo Arnoldi, who, after having received and completed his PhD from Eidgenössische Technische Hochschule (ETH) Zürich, began his professional career working at Credit Suisse Life as head of the newly-established development team for the life insurance department. Inspired, he, along with a team of developers at the time, sought to transform the "business logic" behind the life insurance business that was weighing down larger enterprises, and founded their own company a few years later, which would end up becoming Lifeware [18].

What they foresaw in their initial pursuit was the impact that the new generation of technologies (e.g. web development) would have on the cost of developing insurance products. By staying "lean" in a field which remained inherently bureaucratic and the "human" cost of day-to-day operations remains (to this day!) very high, Lifeware had found a way to rely on their automatized custom-built platforms to maintain a high-quality of services for their clients. By making a software-as-a-service (SAAS) competent enough to distribute and maintain said clients' products, as well opening up the possibility of creating in-house tools and approaches to their software development, Lifeware's key advantage over other companies in the field was and is its ability to significantly reduce the cost of development and increase the speed of its go-to-market platforms, which drives the unconventional practices at the company. Figure 8 outlines Lifeware's business offering. This is important to understand in the context of the types of business objects they handle in their code.

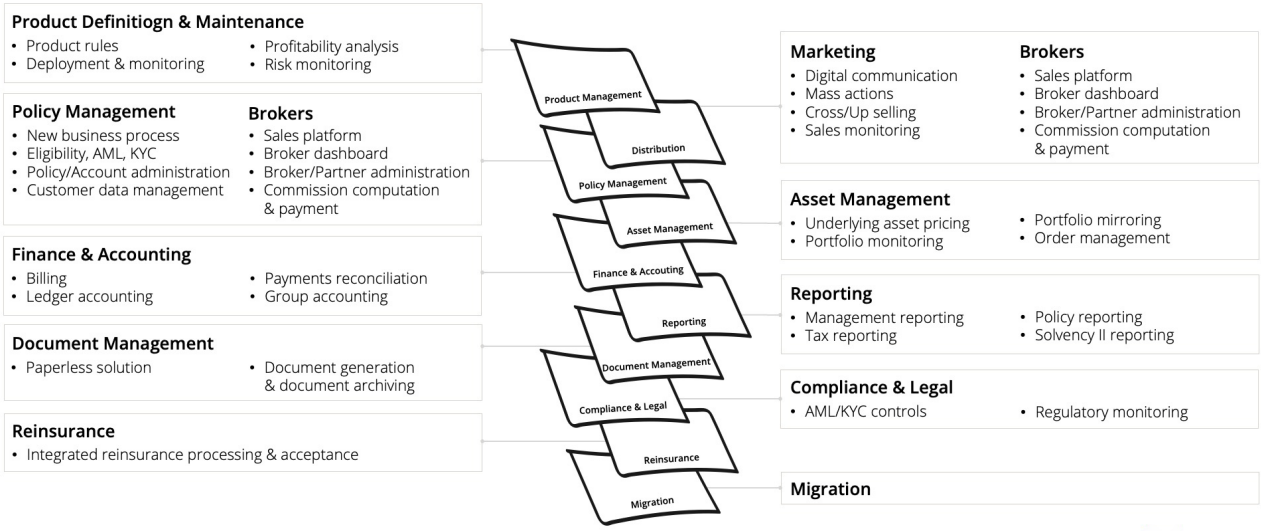


Figure 8. Lifeware Offering

As of 2024, Lifeware has offices in several countries, including Switzerland, Luxembourg and Spain. However, in total, only around 100 employees persist. Deliberately smaller development teams per office are to ensure that XP practices can be followed, as it is harder to maintain the required level of communication on a larger scale. The dichotomous nature of the company, being both very segregated with their internal technologies along their development teams, yet supporting some of the biggest insurance company platforms today, is a resemblance of the philosophy of its founder.

Dr. Arnoldi's PhD thesis at ETH was bout "semi-structured problems" that pave a way for "decision support systems (DSS)" [19] in the realms of software engineering and development,

was a large catalyst for the way he chose to technically manage the business logic of the field. His focus on short delivery intervals and continuous communication with stakeholders and daily integration to production was an unconventional solution for the time (less so today).

Despite the "downfall of Smalltalk" that ensued through the years in the rest of the world, all development at Lifeware has remained solely in Smalltalk, and specifically in VisualWorks. Kent Beck's paradigms of Extreme Programming (XP) and Test-driven Development (TDD) went hand-in-hand with Dr. Arnoldi's Smalltalk-development philosophy at the company, and has stayed as such until a few years ago, when GT had begun to rise in popularity.

1.2.2 Project Motivation

This project is originally inspired by the unique opportunity I had as I began a part-time employment at Lifeware during my final year of studies in my Bachelors. Prior to joining Lifeware, I had no former experience with Smalltalk, and learnt of and about it on the job. Throughout the course of 6 months, I sought for as many resources and materials that were available to me in order to be able to bridge and extrapolate from my more "progressive" Bachelor-level computer science knowledge. A large part of the resources used and cited were all part of my exploration into this field, and my aim is that this report could also provide a more holistic and intuitive introduction for the next generation of programmer looking for an interesting way in the world of OOP.

The formulation of the problem statement itself was aided by the help of Dr. Tudor Girba himself, as well as Dr. Andrei Chis. Since 2016 the team has been sub-contracted for development for the custom-built GT Lifeware image.

1.3 Problem Statement & Structure

The primary problem addressed in this report is the lack of an effective tool to visualize Lifeware's business objects in `gtViews`. This limitation hinders the ability of developers and users to understand and manage their complex software system efficiently. Without clear visualizations, it is challenging to grasp the intricate relationships and dependencies within the system, leading to potential inefficiencies and errors. Moreover, the absence of well-defined views impedes future development and maintenance efforts, as it prevents a holistic understanding of the system's architecture and behavior. Addressing this problem by extending `gtViews` to include robust visualization capabilities within the browser will facilitate better system comprehension, improve adaptability to business requirements, and emphasize the importance of comprehensive views for ongoing and future projects. As an additional by-product, it could benefit the end-users of Lifeware's platforms such as client and backoffice personnel — as instead of creating business requirements to generate visualizations of said business objects, they can benefit immediately in the browser.

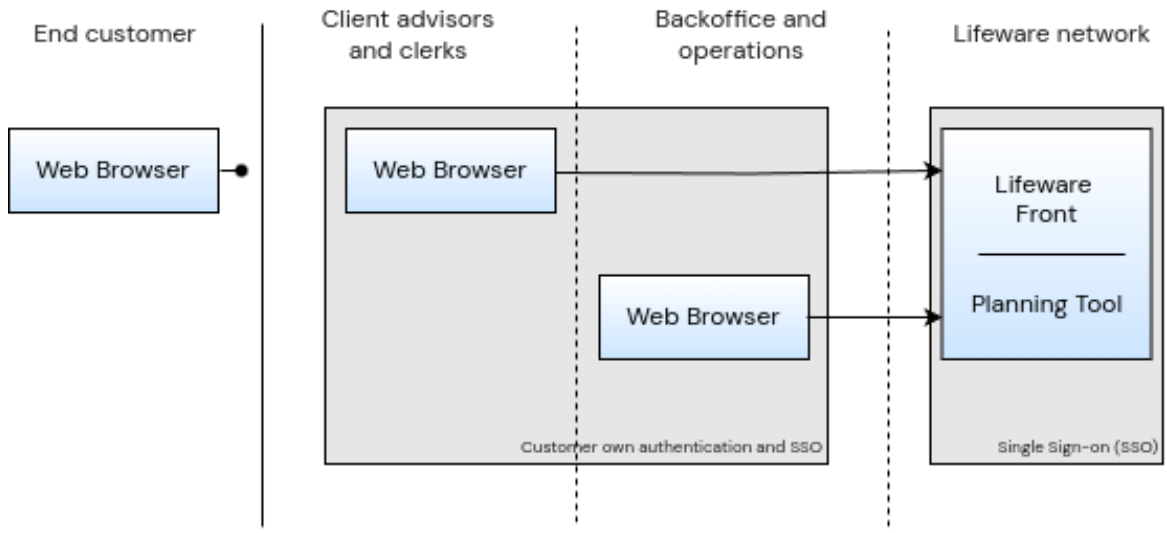
Going forward, the report will contain an overview of the following:

- Section 2 contains an explanation of Lifeware's technical context, for a better understanding of how their system is built
- Section 3.1 contains a high-level understanding of the building blocks currently present in the system to build upon for custom views
- Section 3.2 contains a further detailed example of the implementation, and different features that were explored, including testing
- Section 4 contains a summary of the report, and points for future improvement

2 Technical Context

2.1 State of Art (Current Company Infrastructure & Processes)

All Lifeware application and production code is written, developed and tested in Smalltalk. The central aspect of Lifeware's end-product is its web-application, via which clients and brokers connect to and use to upload, interact, log and store changes, etc. As its service-level agreement (SLA), the system is maintained to be at high-availability for traffic at all times, and maintains a 1:1 redundant storage architecture via 4 data centers hosting GemStone/S instances of client data, test and production code. Test databases mirror, and are replaceable by/for the production system.



Source: Internal Lifeware Wiki
Figure 9. Lifeware Intranet Application Access Layering

Developers work directly on a local Lifeware Smalltalk image (in either VisualWorks or GT), with no necessity for direct access to customer data. This image contains all of the company code, i.e. Lifeware software. It is encouraged to integrate code to production regularly (at least once a day). Depending on the client production, production is usually live-migrated at the end of every business-day.

Functional testing of development code is performed locally during development. Prior to each integration, Lifeware developers are encouraged to perform distributed testing on AWS servers for all production tests. Currently, there are over 100′000 unit tests, with around 10′000 added every year — which thanks to AWS runs on-demand in under 10 minutes. As testing prevails at Lifeware, their VisualWorks and GT platforms integrate a custom tool for inspecting and debugging either individual or large groups of tests. Functional testing is not done superficially, and can range from testing mathematical computations up to fully-realized business cases (e.g. the life-cycle of a single insurance policy over many years), and as a condition, each unit test reconstructs a part of the real production environment and all relevant events. As an example:

```
1       TestContractClass >> #testBalanceAccount
2         | environment contract |
3         environment := self setupContractEnvironment.
4         contract := environment replayAllEventsOfThisContract.
5         self
6           assertBalanceFor: contract debitAccount is: -3090.24 EUR
```

A particularly interesting sub-set of testing tools at Lifeware are those that exist for PDF's,

XLSX's, etc., allowing developers to directly evaluate the diff's of various documents. Directly from the Lifeware internal wiki:

> *Lifeware works according to the coding paradigm Test-Driven Development (TDD) part of the Extreme Programming (XP) paradigm:*
>
> - *All code must have unit tests*
> - *All code must pass all unit tests before it can be released*
> - *When a bug is found, tests are created before the bug is addressed (a bug is not an error in logic; it is a test that was not written)*
> - *Acceptance tests are run often and the results are published*

For modularization of the code, Lifeware maintains a feature-based framework, allowing for new code to minimally obtrude older code, by letting these features be switched "on-off", depending on customer requirements. Initially the feature is only active in the development environment, and then tested in the test environment, and only then and finally being enabled in production.

2.2 (TDD + XP) vs. Moldable Development

While this report focuses on the implementation of a solution for Lifeware, there is reasonable precedent to include a discussion here about the opposing programming approaches that are diverging at the company at present.

Extreme Programming (XP) is an agile software development methodology focused on speed and ease with short development cycles and minimal documentation. Some guiding values and principles of XP are: simplicity, communication, feedback, courage and respect, with a focus on smaller development teams and high levels of communication both within internal and external (client) teams. An essential practice of XP is practicing the test-driven development technique (TDD) that entails writing the minimum amount of code (an automated unit test) before writing actual production code. This is for software engineers to focus on writing code that can accomplish the truly desired function. It is industrially considered to be an intense, but worthy programming approach in the long-term, as it mitigates future code refactoring. According to this paper published by Bin Xu [20], many legacy companies either use it as a "reverse engineering" practice for their legacy systems to a successful degree, or use it consistently when creating systems from the ground up. It is of the highest priority in industrial projects to adopt software processes quickly in order to drastically shorten to time-to-market (TTM) while minimizing risk.

Renowned software engineer Kent Beck, who is prompted to having been the inventor of the XP practice, and often said to have developed/rediscovered the TTD philosophy states that:

> *The original description of TDD was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output. After I'd written the first xUnit framework in Smalltalk I remembered reading this and tried it out. That was the origin of TDD for me. When describing TDD to older programmers, I often hear, "Of course. How else could you program?" Therefore I refer to my role as "rediscovering" TDD. [21]*

On the other hand, Dr. Tudor Girba's Moldable Development (MD) approach puts a strong emphasis on creating a highly customizable development environment that can be tailored to the specific needs of each project. Unlike XP, which adheres to a strict set of practices and values aimed at ensuring high-quality software through rigorous testing and communication, MD promotes flexibility and adaptability by allowing developers to mold their tools by putting forth the hypothesis that the "shape" of the software at-hand is essential to understand first and that it has to be customized to match the context of the developer.

After having a personal discussion with Dr. Girba, he put forth the following:

> *It all starts with some challenge which can concern anything, from a narrow technical detail to a broad business concern. Regardless of the scope, every such challenge relates to a specific problem and requires a decision. Each decision requires information. Today, much information is gathered through manual inspection and put together manually, too. This does not scale as systems are too large and change too quickly to be grasped manually. In fact, developers spend most of their time figuring systems out. We want to optimize this activity by relying on generated views instead. And because systems are highly contextual, we also want these views to be contextual. This can only happen through specific coding.* [how to cite?]

While there is yet a large-scale application of the MD practice put into use, there are interesting implications to the adoption of this approach.

Within the context of Lifeware, the decisions to remain operating in a "hybrid" mode of both approaches is potentially of benefit to all end-users and developers alike. In terms of project complexity of such a highly complex codebase such as Lifeware's, there exists a benefit of including the kind of custom tools that help developer comprehension as MD. However, in terms of initial development speed, the long-withstanding XP practices that have always been at the fore-front of the development teams allow for proper focus and rapid delivery times. By all means, the decision comes down to the expertise of the teams that will have the most key role in development. As most find it easier to use TDD and XP as they are better-versed in those, some may find MD's tool creation and customization more appropriate only for specific use-cases as this project will outline.

2.3 VisualWorks vs. Glamorous Toolkit

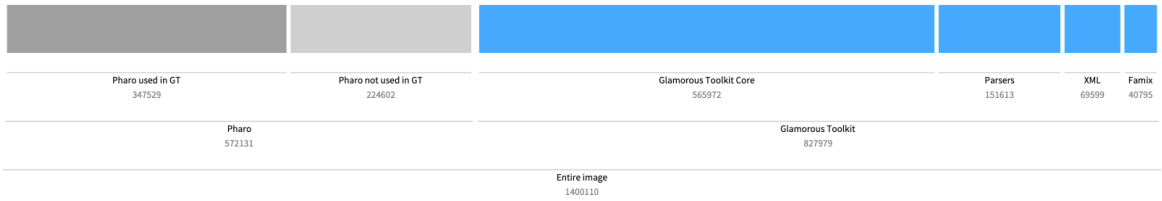
Lifeware's first client production platform, built in the late 90's, was in VisualWorks. For the last 20 years, all of Lifeware's development occurred, and still occurs to a larger extent, in VisualWorks. In the past 5 years, GT has become more prevalent in its usage at Lifeware, however, a bit differently to how it was originally intended to be according to MD practices.

There is a defined split amongst the development teams, as some regard abandoning VisualWorks as a "sunken cost" of years of custom development, with GT (particularly the Lifeware GT image) not being stable enough for daily development. Although, the GT team works directly with Lifeware developers to improve the image, and has undergone a tremendous process in the past year. Similarly, even some qualities such as the UI of GT are still disavowed by most developers, as most are used to the familiar multi-window "universe" display, as opposed to the "tab-based"

Evidently, there are arguments for the usage of either of the IDE's.

Firstly, VisualWorks, although overtime becoming out-dated, has become the basis on which most in-house development tools for Lifeware have been built on. For example, the "Remote Analyzer" tool, which allows the developer changes to be inspected prior to integrating was built solely out of Lifeware's needs for CI.

Secondly, GT is open-source, being based on Pharo. This open-source nature means that developers have access to the source code, allowing for greater customization and the ability to contribute to the development of the platform. Additionally, the open-source community around Pharo and GT fosters collaboration and innovation, providing a wealth of shared resources, tools, and extensions that can enhance development. A move towards open-source technologies could prove to be beneficial for the company in the long-term. For now GT is packaged alongside the Pharo VM (as seen by the LOC in Figure 10), as it is quite difficult to re-engineer the VM from scratch, which in comparison to VisualWorks makes it a bit "heavier" to run consistently.



Source: From the Lepiter GT Book [22]
Figure 10. Lines of Code (LOC) of Pharo and GT image

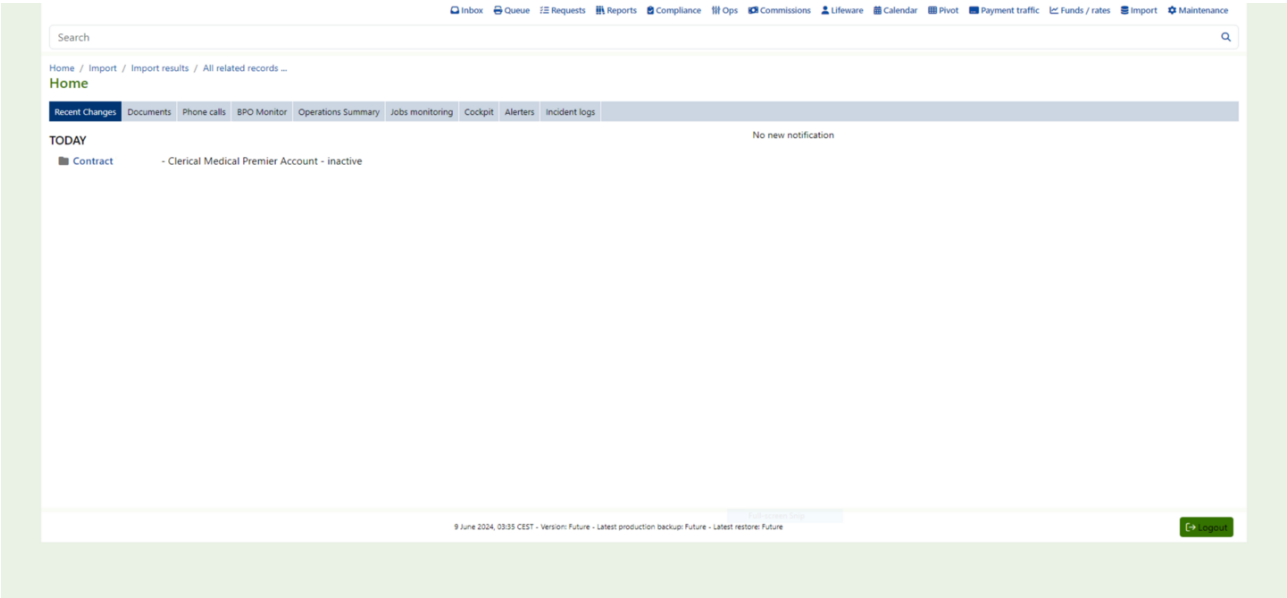
Lastly, Lifeware has already made an internal commitment towards moving their systems to GT as it provides better integration with modern development tools and ecosystems. It supports a wide range of plugins and extensions that can be easily integrated, improving the overall development workflow.

3 Solution

In order to make the further steps of implementation clearer, and to ensure Lifeware's terms of privacy are maintained for their clients, the focus of the explanation of the implementation will be on a singular test-case contained within a specifically-chosen test environment populated with data that is anonymized for this report. Identifying features of clients, customer data or production has been covered and omitted according to Lifeware customer privacy regulations. All development is contained in a snapshot on a proprietary Lifeware image and is not publicly available.

3.1 Design

As the problem states in Section 1.3, we are seeking for a way to visualize already-implemented `gtViews` of Lifeware developers, not in GT, but on their web-platforms. Most of the web applications of Lifeware follow a similar UI-layout, as shown below:



Source: Screenshot from a test environment
Figure 11. Lifeware Client and Back-office Homepage

We can see the main menu at the top and aligned right, the search bar, and the "breadcrumb" tags allowing you to view the history of navigation. This is all important to note at the start, as the modular, tab-based format of the web pages allows for quick prototyping of new pages or features while keeping everything uniform. By utilizing this layout, we can see that the main navigation is contained within the "tabs" that are of interest for viewing located below the breadcrumbs on each page.

Lifeware's web pages are rendered via their very large in-house `PageBuilder` framework (in combination with several other packages), and by exploiting the way it renders the page (i.e. all the accompanying HTML, CSS, JS), we can utilize this to render a `gtView`. Otherwise, one would have to be able to rewrite the HTML to render each view separately, every time one is created. The `PageBuilder` framework takes advantage of the MVC architecture in such a way, that in simple terms — when prompted with an object from the model, the `PageBuilder` class "knows" how to display itself in the web at any given point.

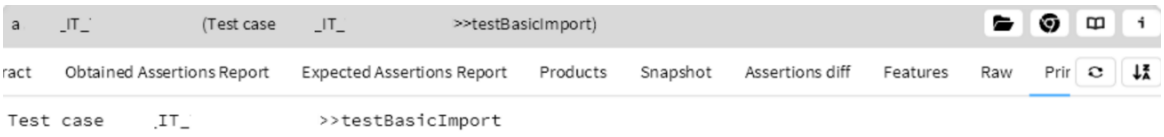
Prior to development, it is also important to understand and see prescient examples of currently implemented `gtViews`. The following query can be run in a `Playground` to see how many `gtViews` exist in the Lifeware image:

```
"Show all the methods that use the gtView pragmas"

#gtView gtPragmas
```

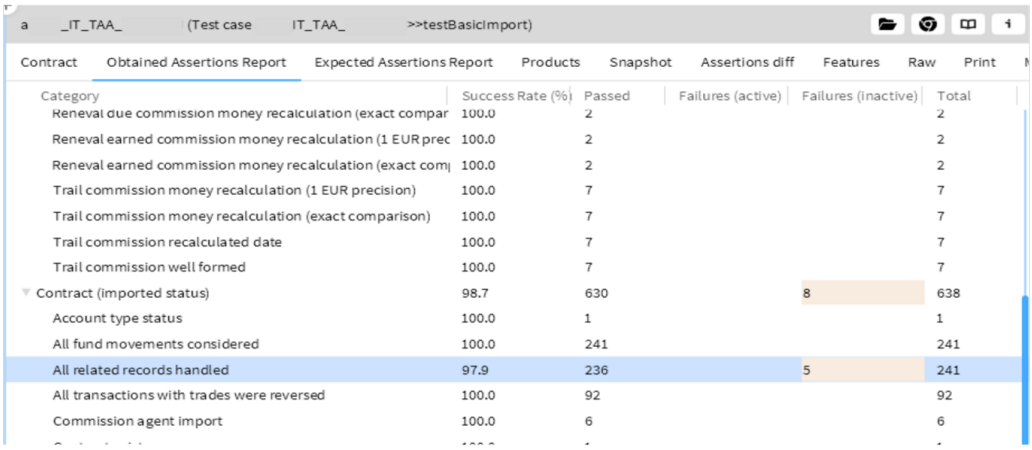
The result of this query is **4554** views. This only shows to reinforce the importance of implementing a solution to the current problem as it is a useful tool for the development at Lifeware.

For the initial implementation, the following test case has been used: `#testBasicImport` (in Figure 12).



Source: Screenshot from a test environment
Figure 12. Selected Test Case for implementation

We can see that the test case already has some pre-built views, such as the `Obtained Assertions Report`, `Expected Assertions Report`, etc. (in Figure 13).



Source: Screenshot from a test environment
Figure 13. Selected View of a Test Case for implementation

The `GtPhlowProtoView` class encapsulates a large amount of different views that come in the base GT image, and after narrowing down several specific views that are most often utilized currently in the company, the following list emerges:

- List
- Columned List
- Columned Tree
- Empty View
- Forwarding Views
- Implicit views[1]

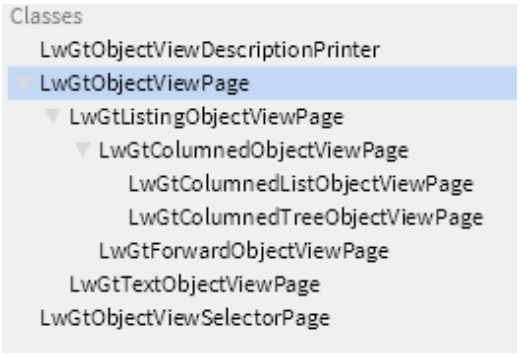
[1]As mentioned previously, these are the `Raw` and `Print` views every object is accompanied by

The implementation varies slightly for each view, but what they all require are the minimal building blocks in order to be rendered: a `model` of the object that is being passed (this is parsable into a `JSON` object), the `ViewSpecification` which contains the information of which type of view the object "asks" to be rendered in, and lastly a `ViewLabel` which gives a name and title for the specific custom view that is being rendered.

During the development process we encountered a problem with `Tree` views, as the current UI tools of Lifeware do not support visualization of tree tables, and so far only visualization of the first level can be exposed. This is a point for future improvement in Section 4.2.

3.2 Implementation

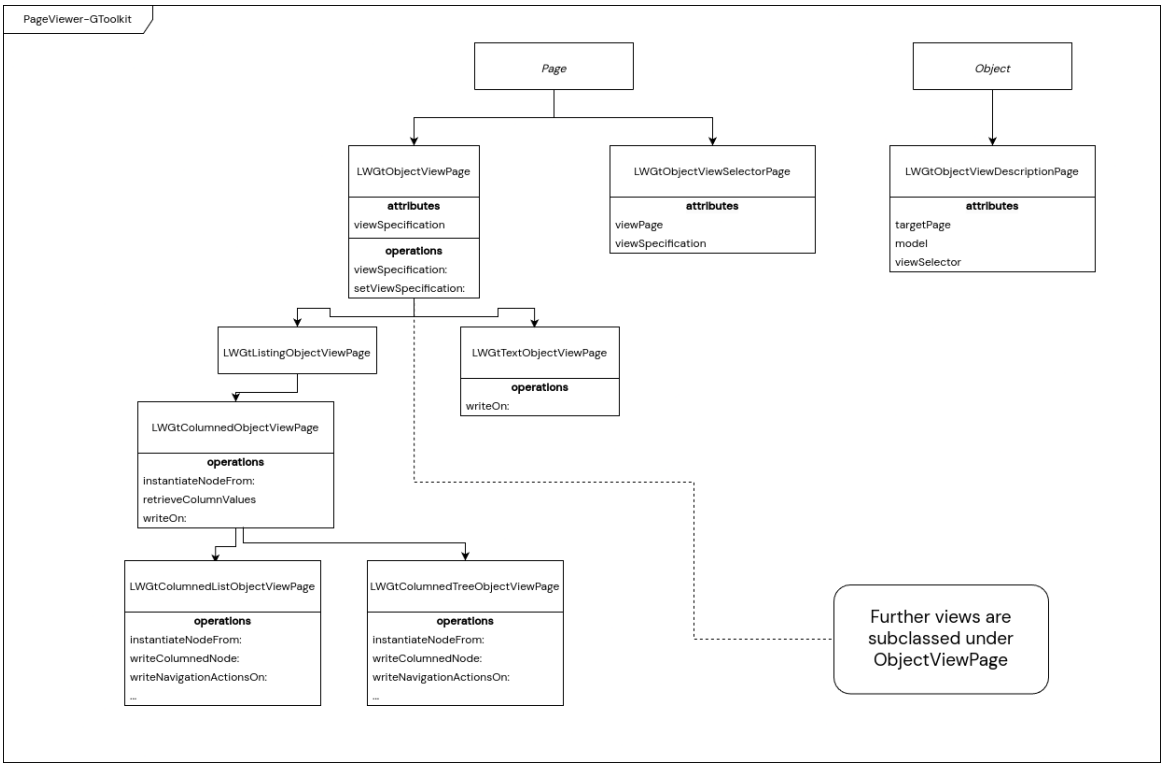
Prior to the initial development, we created a new package called `PageViewer-GToolkit` that will "connect" the rendering capabilities of the `PageBuilder` package while also utilizing the "vanilla" `GToolkit-Phlow` package functionality. The `GToolkit-Phlow` package contains the specifications and GUI decorators in order to render the views within GT, but by utilizing the similar controllers of the class, the functionality can be extrapolated in the web. In Figure 14 we can see the class hierarchy, but for now they represent a mirror to the specific `gtViews` that need to "know" how to be displayed. Figure 15 outlines an eagle-eye view of the class hierarchy for the necessary `Object View` classes.



Source: Screenshot from a test environment
Figure 14. `Object View` classes used

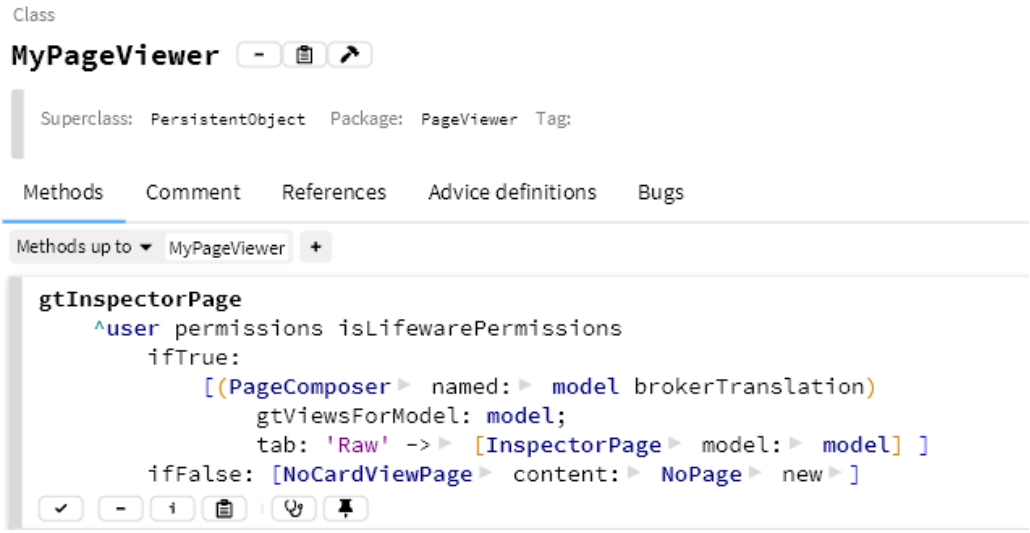
Accordingly, the sibling class `LwGtObjectViewDescriptionPrinter` contains the initializer for the `Writer` object, that instantiates the writing process of the given page (the controller part of the MVC).

The visualizing part of 'MVC' comes from a subclass of the `PageViewer` (related to the `PageBuilder` framework) class called `MyPageViewer`. This is where Lifeware developers can instantiate a "parent" page for all their different production(s) code. In this case, we create a method directly within the `MyPageViewer` class called the `gtInspectorPage`. In this case, any `gtView` that can be rendered within the browser is going to be an instance/extension of this `gtInspectorPage` (as an example, even the `Raw` and `Print` views once rendered are just a `gtInspectorPage` respectively).



Source: Created UML Diagram

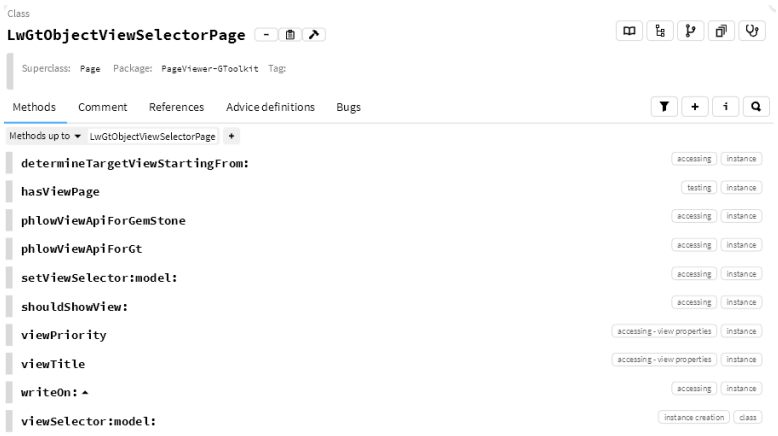
Figure 15. `Object View` Class Hierarchy



Source: Screenshot from a test environment

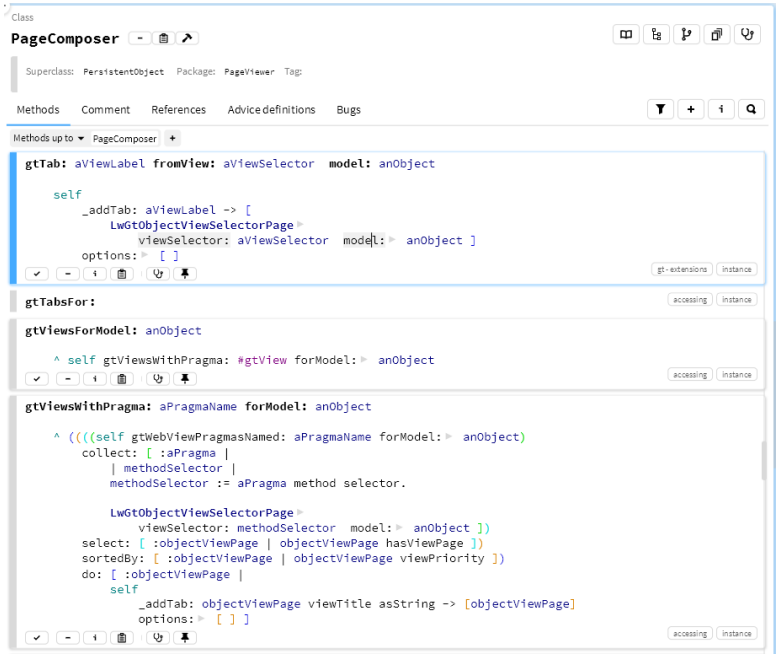
Figure 16. `MyPageViewer` class with selected method

The sibling class `LwGtObjectViewSelectorPage` is also a key part of the implementation. A `ViewSelector` object is instantiated for a given model, based on finding the respective `<gtPragma>` and querying if the model `hasViewPage` is returning `true`. That `view` gets sorted by "rank" of priority in viewing, and is then returned as the `Tab` object for display to the `PageViewer` with the appropriate `LwGtObjectViewDescriptionPrinter`.



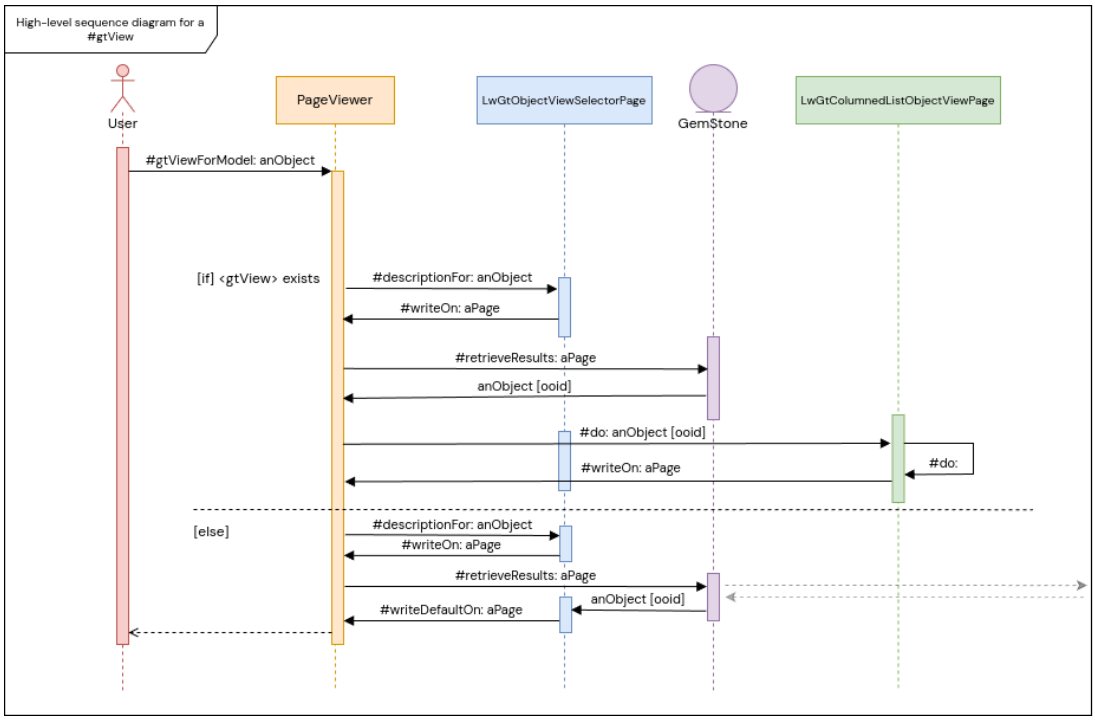
Source: Screenshot from a test environment
Figure 17. `LwGtObjectViewSelectorPage` class with selected methods

Lastly, in order to "attach" a `gtView` to any given page it is done by either "hard-coding" the tab to appear by calling the method the `#gtTab:fromView:model:`, or via the `#gtViewsForModel:` that adds the existing object views (as seen as a message sent in Figure 16). In Figure 18 we can see the full message definitions.



Source: Screenshot from the GT environment
Figure 18. `Meta` view of the `PageComposer` class and select messages

In Figure 19 we can see exactly the order of execution that occurs when attempting to call a `gtView`.



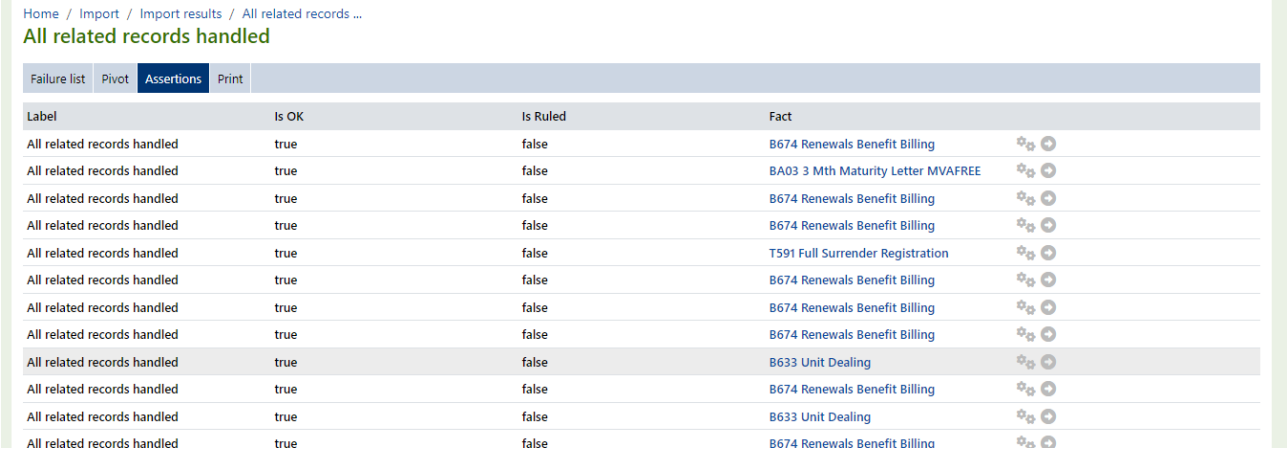
Source: Created high-level sequence diagram

Figure 19. Order of method execution when calling a `gtView` in the browser

3.2.1 Results

Recalling back to Figure 13, as part of the `#testBasicImport` test case, here we see the previously shown `AllRelatedRecordsHandled` in the highlighted row, as an object shown within GT as its `gtView`, when navigated to in GT leads to Figure 20 showing the object itself, and the newly created web version of said view in the browser can be seen in Figure 21. We can note that although this object has several `gtViews` defined, not all render in the browser tabs, as they are not part of the list of supported view types. In section 4.2, a further explanation of what supported view types can be rendered at present is elaborated on.

Figure 20. `AllRelatedRecordsHandled` object



Source: Screenshot from the GT and Test environment

Figure 21. Same `AllRelatedRecordsHandled` object rendered in a `Failures List` GT view and in the browser

3.2.2 Object Navigation

Having the ability to navigate through objects is an important additional functionality to have when inspecting objects. As mentioned previously, Lifeware developers have an custom-built way of mapping objects in their local environments to their GemStone database(s). By using that as a starting point, we included two additional ways of navigating to them when located on a `gtView` in the browser.

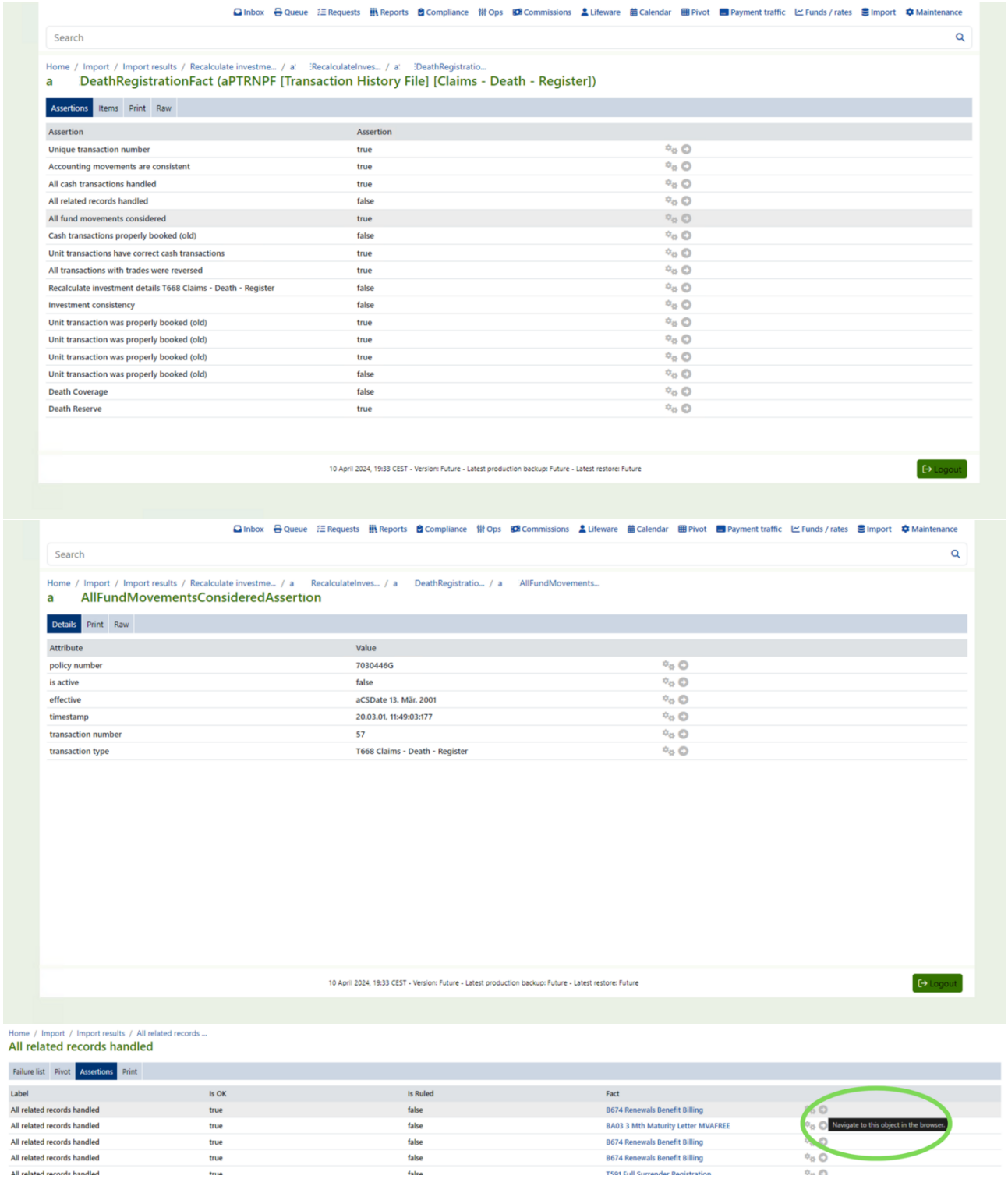
By appending a column with two icons at the end (for views that have tabular formats) — one which opens the object in the GT inspector, and the other which navigates to that object inside the browser developers can seamlesly navigate between the browser and IDE. In essence, for any object navigated to in this manner, if it does not have a way of displaying itself as a `gtView` will revert to its tabular `Raw` view.

A current limitation of this is that for now only inspecting `Persistent` objects is possible, as Lifeware's GemStone databases do not have the necessary base GT classes necessary for subclassing and reusing the views needed.

Since all of Lifeware's code is platform-independent, there are few methods that are specific for GemStone or porting to Pharo. This implies the need for system extensions with the corresponding compiler directives to express the target environment. This is crucial to maintain as a basic requirement of all Lifeware code — that development in a different environment should be able to be merged back to the original environment.

In Figure 22, we can see an example of a `DeathRegistrationFact` instance (which for all means and purposes of this report is just like any other Lifeware business object), which is shown when navigating to its `Assertions gtView`. This is a columned list view that we have seen prior. Appended we can see two icons: a gearbox and arrow. The gearbox icon maps to the selected objects `gtView` (if possible) in GT, and the arrow leads to the next objects representation within the web.

As highlighted in the corresponsing row in the `DeathRegistrationFact`, we can move towards an `AllFundMovementsConsideredAssertion` object with a `Details gtView`. [TODO: include the screenshot of mapping func for this specific object]



Source: Screenshot from a test environment

Figure 22. Basic implementation of object navigation in the browser (navigation to an `AllFundMovementsConsideredAssertion`)

3.2.3 assertView

As mentioned before, the importance of testing at Lifeware is what embodies the companies technical paradigm. Here we see the way Lifeware keeps its code platform-independent, by splitting this test case to be run solely for their GT images.

The test below utilizes the `ViewSelector` object from previously to check that the passed object's model contains a `gtView`, and then proceeds to compare the `Raw` contents of said object.

Simply put, it exploits the parsed `JSON` of the model, renders that the proper view was selected, and then checks that the view contains the same `contents` as that of the `Raw` contents.

```
Object >> LwTestCase
    assertView: aViewSelector model: anObject contents: aContentSelector
    ^self
        forGemstone: []
        forPharoBased: [
        self
            assertGtView: aViewSelector
            model: anObject
            contents: aContentSelector ]
        forVw: []
```

Currently, this test is used for only singular use cases where the `gtViews` in the browser persist. While testing has not been integrated fully-fledged in the system, the goal is that this test is mass-added to pages that call upon the creation of the `gtInspectorPage`.

4 Conclusion

4.1 Summary

Keeping in consideration the implemented solution in this report, we can conclude that the solution for the problem has indeed been implemented, and a "foundation" for `gtViews` is now available for Lifeware developers in their front-end web applications. By utilizing `gtViews`, developers can now create dynamic, visually appealing interfaces with ease, improving the user and developer experience and reducing development time. The integration of `gtViews` also facilitates better coding strategies, allowing developers to focus on building robust applications without having to "disentangle" complicated business objects to understand them.

To repeat what Dr. Girba mentioned previously: "Developers spend most of their time reading code", and this is true as well in the context of this project. In order to avoid code duplication, the "reinvention of the wheel", and keeping the solution as lean as possible for rendering purposes, as well as being unfamiliar with the vast code-base of Lifeware, the largest amount of time was allocated in order to become familiarized with already-present tools, frameworks, API's, and their respective implementations in the image.

[Lifeware LOC? 50 mil]

This is perhaps one of the largest deterrents for the attrition rates of newer generations of Smalltalk developers, as oftentimes complex systems built in Smalltalk (or most languages) do require a large amount of time of the developers to be spent analyzing code. In this way, GT is slowly turning said paradigm over, and allowing for more modern and intuitive approaches and tools to be used in its environment. The `gtViews` are one of the primary ways that allows to cut time in the development process, and in the end boosting developer productivity.

Additionally, the discussion on migration to open-source technologies is crucial, considering the general shift towards open-source solutions and the necessity for tools that align with this trend, especially given the legacy language's lack of quick, easily-built tools and limited libraries.

4.2 Future improvement

Throughout the course of the project, a few limitations were either identified or encountered along the way that hindered the implementation to certain extents.

Firstly, in order to represent more advanced kinds of views in the browser such as `Tree` or `Circular` views, their supporting rendering capability must first be implemented in the `PageBuilder` and framework. Currently, Lifeware has no support for rendering these kinds of objects in `HTML`, and will need to be extended further in the future for support of these views. This feature will be particularly useful for representing nested business objects and complex relationships, making it easier for developers to explore and interact with their data and can be noted as room for future improvement. This is also prescient for the continuing layers of a `Forwarding` view.

In line with the limitations of the `PageBuilder` framework, incorporating stylization of text in a `gtView` is currently not supported (see Figure 20 to an example of said stylization). GT comes with its own graphical stack composed of several frameworks such as `Sparta` (for the canvas), `Bloc` (the core UI framework), `Brick` (the widget library), and also `Phlow` — which is the engine behind inspector views overall. The `Phlow` engine comes with the support for rich text formatting, such as bold, italics, underlining, and different font sizes and colors. Unfortunately, for now it is still only supported only inside the inspectors. Improved text formatting will make information more readable and highlight important details, improving overall user comprehension and experience.

Secondly, due to the volume of some of Lifeware's objects, there exists a need for "lazy loading" of some these objects once in the browser. Implementing lazy loading will significantly improve performance, especially when dealing with large datasets or complex visualizations. By loading data only when it is needed from the GemStone databases, rather than all at once, we can reduce the initial load time and make the application more responsive. This will also minimize memory usage, allowing the tool to handle larger volumes of data efficiently.

Lastly, in the current implementation, we use system directives to control where these views are being implemented. This extrapolates towards the permissions of who can see these views as well, in terms of developers, clients, and backoffice personnel. A stricter implementation for permission will be necessary when implementing this feature fully in production. Also, it can act as an additional layer of support to aid Lifeware in transitioning from VisualWorks to GT; an interim way for these views to inspected in the browser for the developers who continue to use VisualWorks should be in place. This is also a way to persuade developers to move to GT in the meantime.

The outlined improvements will not only enhance the current functionality but also pave the way for future advancements and integrations at Lifeware.

5 Bibliography

References

[1] feenk.com. Glamorous Toolkit.

[2] Pharo Consortium. Pharo Documentation. https://pharo.org/documentation, n.d. Accessed: May 14, 2024.

[3] feenk.com. Moldable Development. https://moldabledevelopment.com/, n.d. Accessed: May 14, 2024.

[4] Lifeware. Lifeware: Transforming Organizations through Agility and Innovation. https://www.lifeware.ch/, 2024. Accessed: May 14, 2024.

[5] John Delaney. COBOL Programmers are Back In Demand. Seriously. https://cacmb4.acm.org/news/244370-cobol-programmers-are-back-in-demand-seriously/fulltext, 2020. Accessed: April 30, 2024.

[6] Rachit Awasthi. $1.14 Trillion to Keep the Lights on: Legacy's Drag on Productivity. https://www.mechanical-orchard.com/insights/1-14-trillion-to-keep-the-lights-on-legacys-drag-on-productivity#:~:text=On%20average%2C%2031%25%20of%20an,made%20up%20of%20legacy%20systems.&text=Maintaining%20those%20systems%20can%20be,allocated%20to%20keeping%20them%20running, 2023. Accessed: May 14, 2024.

[7] Alan C. Kay. *The early history of Smalltalk*, page 511–598. Association for Computing Machinery, New York, NY, USA, 1996.

[8] Richard Kenneth Eng. Why Smalltalk is so easy to evangelize. https://itnext.io/why-smalltalk-is-so-easy-to-evangelize-2b88b4d4605c, 2020. Accessed: May 14, 2024.

[9] Cincom Systems Inc. Cincom Smalltalk: VisualWorks. https://www.cincomsmalltalk.com/main/products/visualworks/, n.d. Accessed: May 14, 2024.

[10] David Buck. Software Reflections: Smalltalk History. https://simberon.blogspot.com/2012/04/smalltalk-history.html, 2012. Accessed: May 14, 2024.

[11] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in Context: The Shape of the Design Space. In *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 06 1993.

[12] feenk.com. What is Reflection? https://book.gtoolkit.com/what-is-reflection--etmq5j46471mqbagawov662m3, n.d. Accessed: May 14, 2024.

[13] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *Proceedings of the 4th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '89)*, pages 327–335. ACM, October 1-6 1989.

[14] George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, jun 1984.

[15] Software Composition Group at the University of Bern. Moose. https://moosetechnology.org/. Accessed: May 14, 2024.

[16] Tudor Gîrba. Curriculum Vitae. http://tudorgirba.com/download/tudorgirba-cv.pdf, n.d. Accessed: May 14, 2024.

[17] Tudor Gîrba. Mapping Moldable Development. https://book.gtoolkit.com/mapping-moldable-development-59974yztvzadec0whbcsp3np5, 2024. Accessed: May 14, 2024.

[18] Ward Cunningham. LifeTech. https://wiki.c2.com/?LifeTech, n.d. Accessed: May 14, 2024.

[19] Massimo Carlo Giuseppe Arnoldi. *Aufbau von DSS-Generatoren für Endbenutzer*. Doctoral thesis, ETH Zurich, Zürich, 1989. Diss. Techn. Wiss. ETH Zürich, Nr. 8871, 1989. Ref.: H.-J. Lüthi ; Korref.: H.-P. Frei.

[20] Bin Xu. Extreme programming for distributed legacy system reengineering. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 2, pages 160–165 Vol. 1, 2005.

[21] Extreme Programming. When is XP Appropriate? http://www.extremeprogramming.org/when.html, n.d. Accessed: May 14, 2024.

[22] feenk.com. Glamorous Toolkit and Pharo. https://lepiter.io/feenk/glamorous-toolkit-and-pharo-9q25tavxwfq6z1drwvegd5u9o/, n.d. Accessed: May 14, 2024.