Lxledger: powerglue for small enterprise plain text accounting

Pieter Vermeylen

Haaga-Helia University of Applied Sciences
Business Information Technology
Bachelor's Thesis
2025

Abstract

Author(s)
Pieter Vermeylen

Degree
Bachelor of Business Administration

Report/Thesis Title
Lxledger: powerglue for small enterprise plain text accounting

Number of pages and appendix pages
30 + 3

Plain text accounting in which accounting entries are saved in the plain text format and edited using a plain text editor may provide the lowest barrier of entry for programmers to get started with accounting. Out of the box however, plain text accounting falls short for business accounting purposes.

This thesis explores whether the development of a command line application called lxledger can speed up the accounting of Luminix. Luminix is a small language technology company that is using plain text accounting for its simplicity. However, manually entering invoices, bills and bank statements has been slow and error prone. Tools that automate these actions are widespread and generally well received in accounting.

To guide lxledger software development, a plan-driven software development methodology was chosen over an agile approach. This choice was made to gain experience with plan-driven software development and because it is a good fit for the lxledger project. Keep It Simple Stupid (KISS) was also selected as a guiding principle for the software project.

The thesis structure broadly follows the Software Development Life Cycle (SDLC) from design to acceptance testing. The program scope and features are defined using gradually more specific diagrams. The initial diagrams identify actors, tasks, data sources and the relationships between them. This was then translated into implementation by adding the data layer and application API diagrams. The implementation of the lxledger user interface, business logic, data layer, testing framework and acceptance testing are briefly introduced.

Both the objective to speed up Luminix accounting and the plan-driven approach to software development are considered a success. All planned features have been implemented. Real world usage has shown tangible accounting speed improvements.

The thesis concludes by suggesting future lxledger improvements, explaining the use of AI and recommending a nuanced approach to software development methodology that uses elements of plan-driven and agile approaches based on how well they fit a particular project.

Key words
Small enterprises, accounting, case study, software development methodology, plan-driven design

Table of Contents

1 Introduction

1.1 Background

This thesis focuses on the accounting needs of Luminix, which is a small language technology company founded in 2011. Luminix is the author's company. Currently plain text accounting is used for its simplicity, but the business accounting is too labour intensive with regards to:

- Updating the books to match the latest bank statements.
- Entering invoices.
- Entering bills.
- Reconciling bills and invoices with bank statements.
- Generating the monthly VAT report for the tax authorities.

While the plain text files are human readable, the accounting information is typically extracted using a command line application. An introduction to plain text accounting is provided in section 1.3.

The hledger application used for the accounting of Luminix comes out of the box with a compact feature-set, but the simplicity of the plain text accounting format makes creating a custom solution possible. Other hledger users report they are using hledger for business accounting purposes as well. They also report customizing the hledger experience with scripts. (Hartwell, 2021)

The basic premise of this thesis is that the labour-intensive accounting tasks above are low-hanging fruit to speed up with automation.

1.2 Task automation benefits and compatibility

Research on the influence of information technology on accounting efficiency is generally positive (Assad, Hamdan and Zakaria, 2024). It even has the potential to increase work-life balance (Cooper *et al.*, 2022). Saving time and improving efficiency are also listed as accounting automation benefits (Rawashdeh, Bakhit and Abaalkhail, 2023). However, it is important to focus on tasks that can be automated, and whether a task can be automated or not may be difficult to find out without deeper investigation (Korhonen *et al.*, 2020).

The advantage of automation is the greatest for mundane tasks where much information needs to be transferred (Keys and Zhang, 2020; Remlein, Nowak and Romanchuk, 2024). This makes entering invoices, bills or bank statements prime candidates for automation. Reconciling entries is trickier. Whereas typically the sums of invoices and payment entries will match, this may break in the case of human error such as when a client does not transfer the correct payment. Therefore, the automated process should allow for human intervention without breaking.

1.3 Introduction to plain text accounting

Luminix is using double-entry accounting with hledger to keep track of incoming bills and outgoing invoices. Hledger is by its own account the most user-friendly plain text accounting app (Michael, Simon and contributors, 2024).

```
2025/01/05 * startup investment
    bankaccount                         1000.00 EUR
    investmentcapital                  -1000.00 EUR

2025/01/15 * Cloudprovider Inc; bill:00001,due:2025/01/28,Cloud Platform
    costs:cloudservices:vat-general       10.00
    vat-total:vat-receivable               2.55
    payable                              -12.55

2025/01/28 * Cloudprovider Inc; bill_payment:00001
    payable                               12.55
    bankaccount                          -12.55

2025/01/29 ! Customer Smiths Inc; invoice:00001,due:2025/02/11
    receivable                           125.50
    vat-total:vat-payable                -25.50
    sales:fi:vat-general                -100.00

2025/02/11 ! Customer Smiths Inc; invoice_pament:00001
    receivable                          -125.50
    bankaccount                          125.50
```

Figure 1. Example plain text ledger

Figure 1 shows the contents of a sample plain text ledger file. Each ledger entry is formatted according to specific rules. Ledger entries start with a date followed by an optional marker and then a description. The following lines need to be indented and contain account-specific rows.

The plain text accounting file in Figure 1 contains:

- A startup investment, raising the bank account balance from 0 to 1000.
- An incoming bill 1 from a supplier that has been registered into the accounting before it has been paid. This makes it possible to track debt (payables).
- A payment matching bill 1, which now returns the payable accounts balance to 0. The amount of debt is now 0 again.
- An outgoing invoice 1, which is recorded to the books when the invoice is sent, so we know what income (receivable) to expect and what kind of income we just made (sales:fi:vat-general).
- A payment matching invoice 1, which has arrived in the next month. This returns the receivable account’s balance back to 0. The bank account balance is raised by 125.50.

With a basic understanding of how double-entry accounting works, the plain text ledger file is easy to understand. Based on the entries above, the bank account balance calculation for February 11, 2025, should be 1000 - 12.55 + 125.50 = 1112.95.

The sales for the year amount to 100 and the expenses to 10, so the profit so far is 90. The tax office is owed the tax of our invoice sent (25.5) minus the tax paid to the cloud provider (2.55).

The hledger command line tool helps with these calculations. To figure out the bank account balance at the end of January the command *hledger -f ./sample.books bal bankaccount -e 2025-02-01* can be used.

```
          987.45 EUR  bankaccount
--------------------
          987.45 EUR
```

Figure 2. Output of hledger command to show account balance.

It is also possible to get a view of all the transactions related to the bank account by using *hledger -f ./sample.books reg bankaccount -e 2025-03-01.*

```
2025-01-05 startup investment     bankaccount             1000.00 EUR   1000.00 EUR
2025-01-28 Cloudprovider Inc      bankaccount              -12.55 EUR    987.45 EUR
2025-02-11 Customer Smiths Inc    bankaccount              125.50 EUR   1112.95 EUR
```

Figure 3. Output of hledger command to show account register

1.4 Objectives, scope and target audience

The primary goal of the thesis is to automate accounting tasks to speed up accounting and reduce errors. The accounting automation tool that will be developed to do so is called lxledger. It will be made available on GitLab under the MIT license. The software development will be deemed a success if lxledger speeds up the Luminix accounting.

The secondary objective is to apply a plan-driven approach to software project development to gain firsthand experience of the advantages and disadvantages compared to emergent design approaches.

The thesis documents the software design methodology, design and technical implementation of the lxledger command line application. Plan-driven design and the models that have guided the development are gradually introduced and illustrated with examples from the lxledger software development.

Comparisons of various accounting tools or a discussion on the benefits of plain text accounting are outside the scope of the thesis. The thesis touches upon various accounting principles but does not explain them in depth.

The target audience of this software development project thesis is anyone who is interested in creating simple command line python applications, learning about plan-driven software development methodology or plain text accounting.

References in the thesis were managed using Mendeley Cite.

2 Lxledger software development methodology

2.1 Introduction

To get from big idea to working software product there are two main software development project methodologies. One is the plan-driven approach, which is also referred to as traditional, Big Design Up Front (BDUF), rigorous design, or the waterfall model. The competing software development methodology is known as iterative, Agile, emergent design, JIT, or ad hoc design. (Fagarasan et al., 2021; Pressman and Maxim, 2015; Britton, 2016; Spurrier and Topi, 2023)

There seems to be some confusion regarding the origins of the waterfall model or its name (Tobi, 2012). However, it is broadly understood as a model that describes the software development life cycle (SDLC) as consisting out of distinct steps that are linearly progressed through with defined acceptance stages. Figure 4 shows an example of a waterfall model.

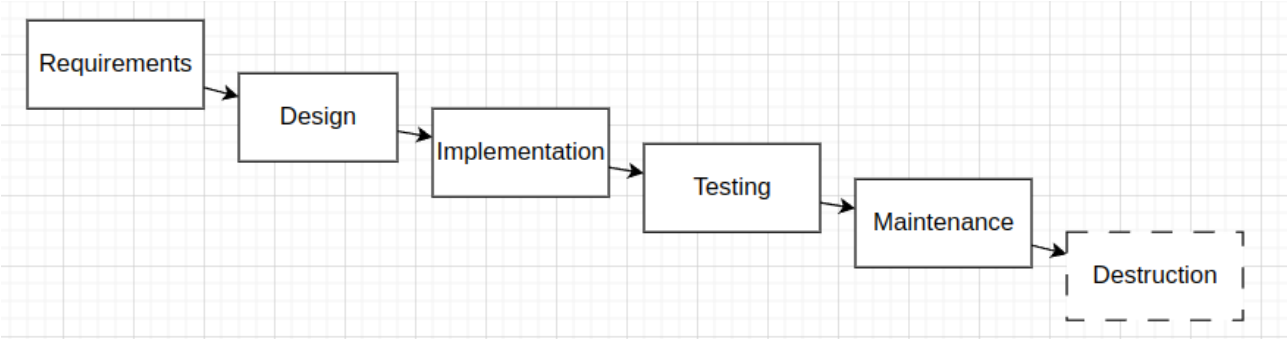

Figure 4. Depiction of stages in a waterfall methodology

2.2 Selecting the right methodology

According to for instance Fagarasan *et al.,* the documentation-heavy and process-oriented water-fall model has been gradually losing favour since the Agile Manifesto was published in 2001 (Beck *et al.*, 2001). Iterative approaches typically favour the team's tacit knowledge over documentation and focus on delivering at least some useful features faster. Agile projects are seen as having lower risk and according to Khoza and Marnewick as being more successful (Khoza and Marnewick, 2020), but they typically require more customer involvement throughout the duration of the project.

Experts seem to agree that plan-based approaches are not outdated (Fagarasan *et al.*, 2021), but that different projects require different approaches. Projects with clear requirements (Fagarasan *et al.*, 2021) that will not change, without fault tolerance and with functional integrity (Ünal, Öztürk and Demirag, 2023) and projects with many non-functional requirements such as usability, reliability, performance or supportability may suit plan-driven approaches better (Spurrier and Topi, 2023).

Some favour a hybrid approach with elements from plan-driven and agile methodologies (Port and Bui, 2009; Iwersen, Zem and Penteado Neto, 2023) and some argue that the methodology matters less than people think (Thummadi and Lyytinen, 2020). It appears that ultimately, the only way to find out for sure is to try.

The main argument for selecting the plan-driven approach is that multiple sources highlight the importance of static requirements as the most important marker of a good fit for plan-driven approaches (Thesing, Feldmann and Burchardt, 2021; Ünal, Öztürk and Demirag, 2023). Since the basic requirements of the lxledger application can be clearly defined and are unlikely to change during development, this project should be a good fit for the plan-driven approach. The realm of accounting in general is highly regulated and the accounting processes that need to be optimized are well defined.

2.3 Keep It Simple Stupid (KISS) as guiding principle

The main principle of software development that guides this project is Keep It Simple Stupid (KISS). This is in line with the plain text accounting approach. The goal is to design a solution that is as simple as possible without compromising on required features and usability.

The need for a KISS approach stems from the fact that software projects are complex. In fact, Hardenberg argues that software projects are stable at a certain level of complexity. Actions can be taken to reduce the complexity, but that frees up complexity tolerance to add more complexity again. Software projects suffer from complexity homeostasis. (Hardenberg, 2024) The complexity homeostasis of this project is kept deliberately low to meet the non-functional requirements of ease of maintenance and ease of use. It is also seen as a challenge for the plan-based approach. It should be interesting to find out, whether planning will reinforce the KISS principle or work against it by creating additional documentation.

3 Lxledger design

3.1 Requirements specifications or requirements design

Typically gathering the requirements is seen as a step prior to starting the design. Britton argues that it is important to look at this stage from a design point of view. Requirements are not gathered, they are designed (Britton, 2016). The same idea can also be found in the term requirements engineering (Pressman and Maxim, 2015, p. 132). Specifying the requirements is an active activity in which choices must be made. Therefore, it is difficult to separate from the design stage.

3.2 Lxledger design starting point: the big idea

Entering all plain text accounting entries manually is labour intensive and error prone. The lxledger tool should automate this process. The example ledger file examined in section 1.3 shows three different entry sources: bills received, invoices sent, and bank account statements provided by the bank.

Since business accounting is evidence based, numbered copies of invoices and bills need to be recorded as proof of the accounting entries in the books (*1558/1995* | *Lainsäädäntö* | *Finlex*, 1995). Invoices and bills are typically sent in paper or electronic format and need to be numbered and recorded somewhere for accounting purposes. Luminix accounting already stores invoices and bills as image or pdf files in separate invoices and bills directories. These directories provide an obvious source from which to gather the information needed to create the bills and invoices ledger entries.

To automatically enter bank statements, Nordea's NDA files can be used. These files are available through Nordea net banking. NDA files are fixed-width text files that can be parsed to extract the necessary information to create the ledger bank account entries.

Once all bill, invoice and bank statements are recorded in the books, the last job to automate is matching them. This process is called reconciling. By reconciling negative bank account entries with bill entries and positive bank account entries with invoice entries, it is possible to keep track of what bills and invoices have been paid for instance.

Finally, while hledger comes with reporting capabilities, some reports for the tax office can be too complicated to generate with hledger. An example of such a report is the amount of tax to be paid to the Finnish tax authority. This report needs to combine employee payment data from the last month with the VAT of the month prior to that. Getting this information should be just one command instead of having to manually combine the information of different hledger reports.

3.3 Lxledger context and integration design

The lxledger requirements design was started by identifying different user roles, tasks and data sources involved in the accounting process. Then a diagram was drawn to gain an overview and to understand the interdependence between the identified elements. This approach is according to Britton's suggestions regarding requirements design (Britton, 2016). To consider future application requirements, more roles and tasks were identified than would be covered by the application. The elements were then grouped by whether they would be within the scope or not. For clarity, relationships of elements not within the application scope were not included in the diagram.

Britton refers to this stage as context design, but the use of diagrams and models to help specify the requirements is common practice (Pressman and Maxim, 2015). Since this design step focuses primarily on what the system should do, it can be seen as focusing on defining functional requirements. The table below shows the brainstorming result for the lxledger application.

Table 1. Identifying actors, tasks and data sources

User groups	Accountant, business owner, project manager
Tasks	• Save bill entry to accounting with reference to proof • Save invoice entry to accounting with reference to proof • Record bank statements to accounting • Reconcile statements • Update supplier information • Set accounting rules. • Generate monthly VAT report • Process payroll statements • Create invoice • Track project profit.
Data sources	Books, Omavero (Finnish tax authority portal), companies, invoices, bills.

The identified items were then placed on a diagram, classified as being out of scope or within scope, and relationships between the different elements were drawn using arrows. See Figure 5 for the result.

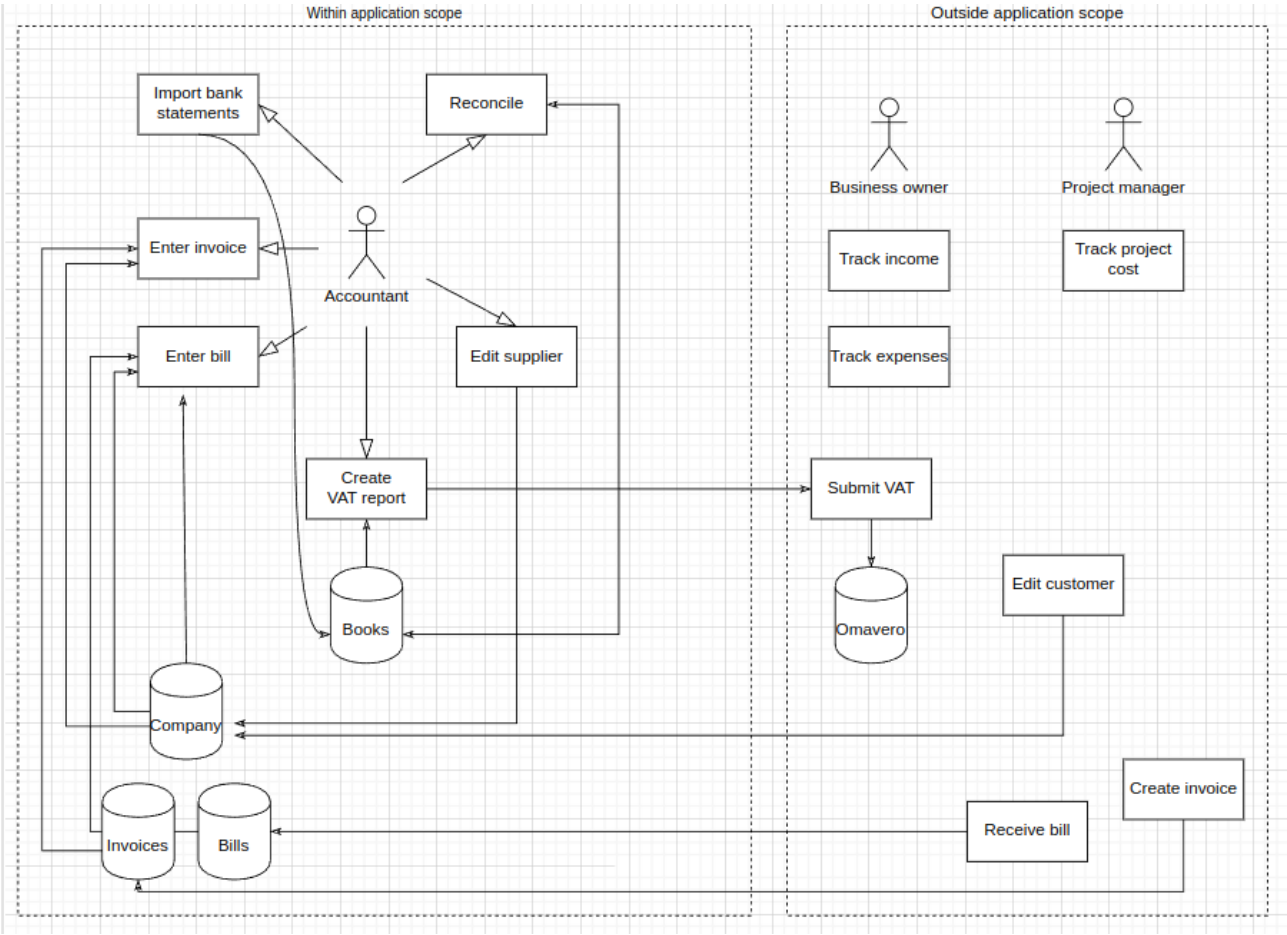

Figure 5. Lxledger initial design drawing

This helped narrow down the core requirements to:

- bill processing
- invoice processing
- bank statement processing
- reconciling accounting entries
- VAT report

To further narrow down exactly what features should be handled by what application, the identified elements were then grouped by services or applications. Britton defines this stage as integration design. This stage also identifies services that could enhance the core application in the future.

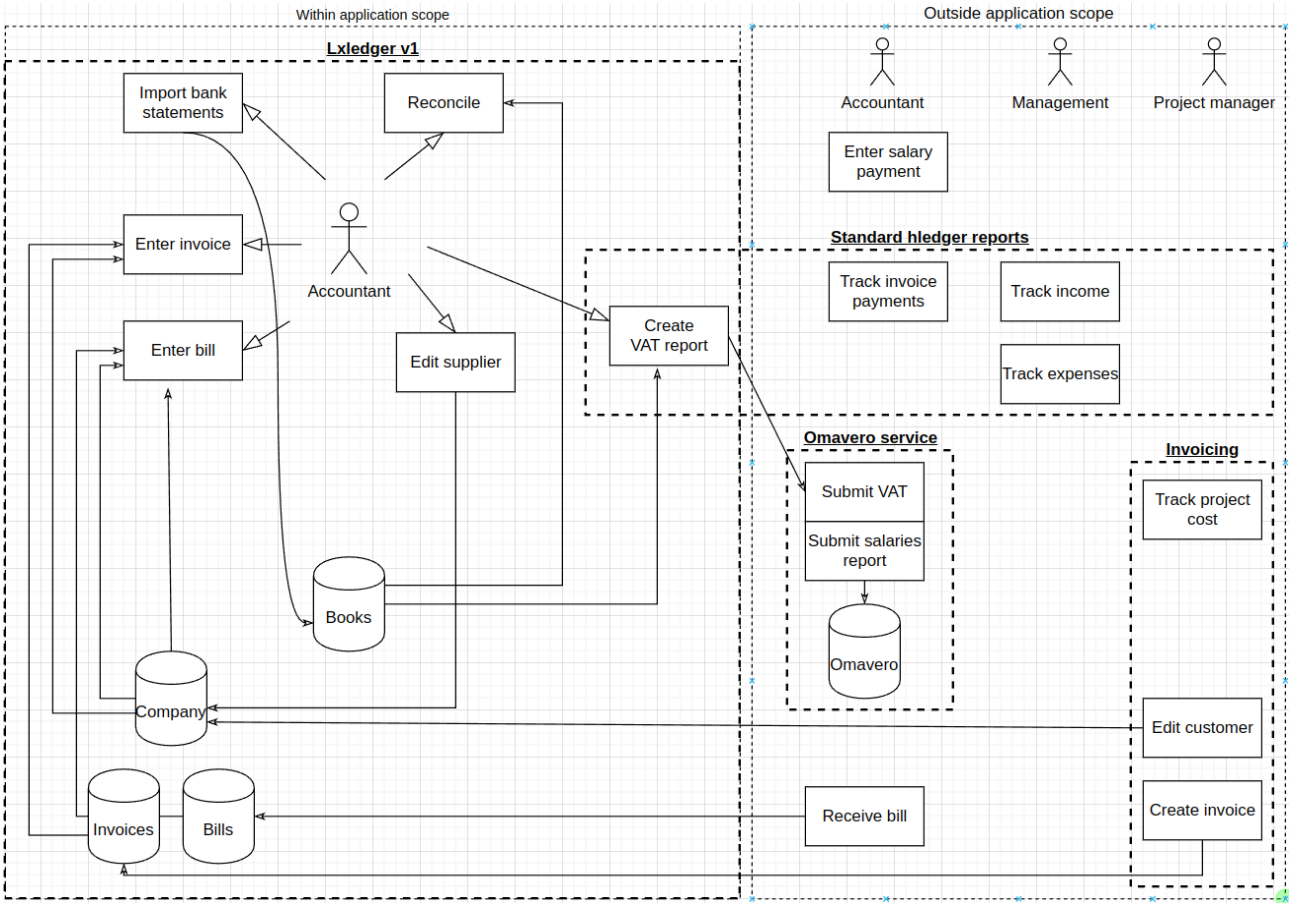

Figure 6. Diagram grouping by applications and services

This highlighted for instance that:

- The VAT report mainly depends on standard hledger reports.
- In the future, an invoicing application could integrate with the lxledger accounting by adding invoices to the invoices data storage. Invoice creation should not be part of the core lxledger application.
- The Omavero (Finnish tax authority) integration should be its own service.

Working on these design drawings provided the basis for the next steps of the design. The data store elements inspired the data design, the main actor (accountant) informed decisions about the user interface design and the relationships (arrows) between the different elements provided hints on how to structure the code to reduce complexity (interdependency).

3.4 Data layer design

A supporting diagram was drawn to gain an understanding of what the data design could look like.

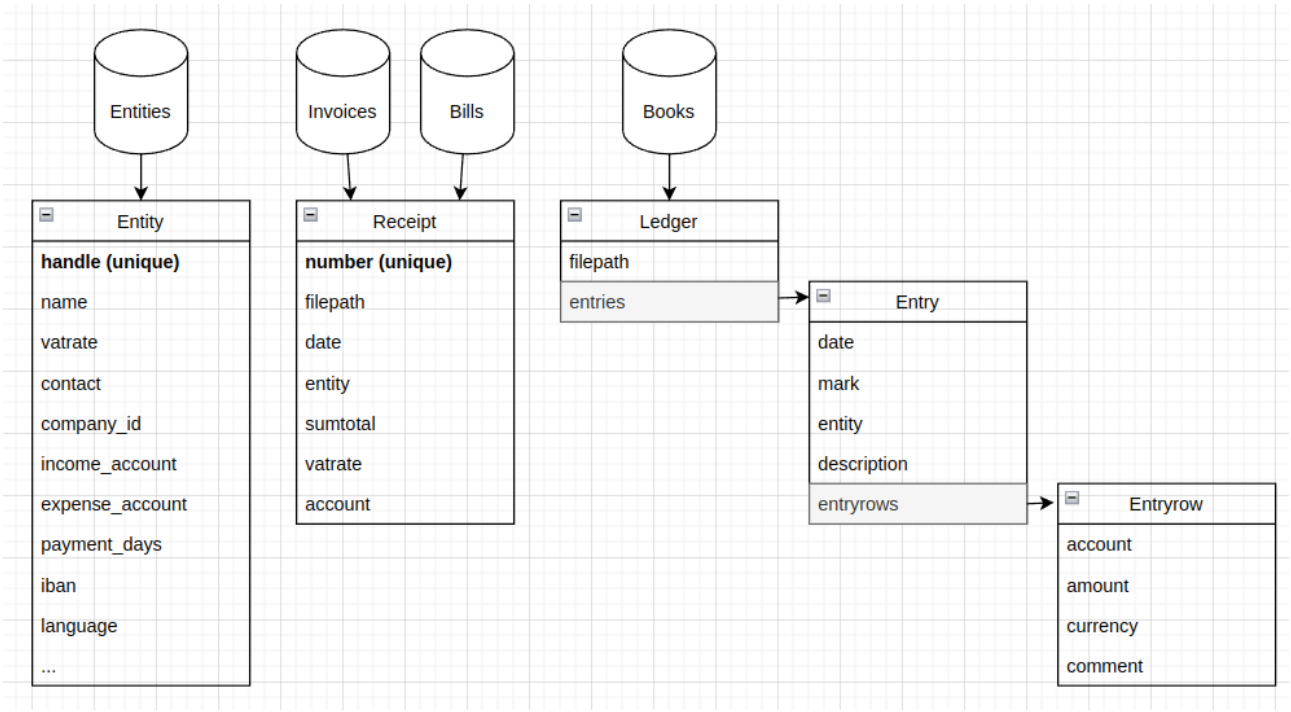

Figure 7. Data layer design drawing.

During this design phase, it became obvious that:

- Companies would be better identified as entities, since not all customers are necessarily companies.
- The number of fields for an entity should be flexible. On every iteration, more possible fields sprung to mind. Note that some of these were not strictly needed for lxledger but may be needed by future integrations. For instance, an invoicing tool would need to know the language preference of an entity, but for the lxledger CLI this is irrelevant.

Pressman & Maxim mention there are different architectural design genres. The lxledger architectural design can be classified as data centred. Different applications can have independent access to the data. (Pressman and Maxim, 2015) Partly this is made possible thanks to the absence of the concurrency non-functional constraint. Since the application, or in fact the whole application ecosystem, is used by at most one user at a time, typical challenges in a data-centred approach such as concurrency are not a major concern.

3.5 User interface design

3.5.1 Guiding principles

Theo Mandel coined three golden rules of interface design (Mandel, 1997):

1. Place the user in control
2. Reduce the user's memory load
3. Make the interface consistent

Human understanding is shaped by our experiences and the metaphors humans create. Fundamentally, humans understand one thing in terms of another (Lakoff and Johnson, 1980). Therefore, an intuitive user interface design fits the metaphors or knowledge structures the user is familiar with. Theo Mandel's golden rules can be seen as sanity checks that make sure this is the case. One important implication is that it is important to understand who the users of the application are and what prior knowledge they have that could provide them with a better user experience.

3.5.2 Lxledger UI design choices

Command Line Interface

The main target user of the application, the author, has extended experience with command line applications. One application that springs to mind is the command line interface of GIT. The GIT CLI is consistent and has excellent documentation that helps reduce the user's memory load.

As discovered in the early design stages, our application should be able to:

- process new bills, invoices and bank statements
- reconcile accounting entries
- generate reports

Interaction with the lxledger UI was made as natural as possible by:

- Using natural language that is easy to remember.
- Providing help to the user at each stage to further reduce the memory load.

The result can be seen in below.

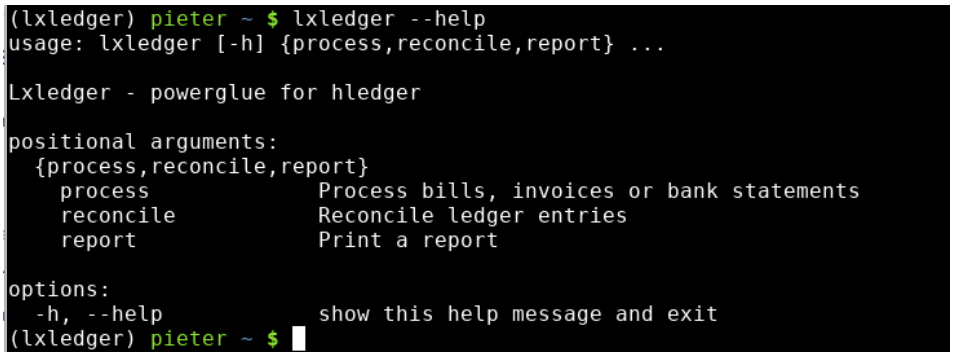

Figure 8 Output of the lxledger --help command

By typing *lxledger --help*, the program displays the minimum amount of information. To find out what the available options are for processing, dig deeper with the command *lxledger process --help*.

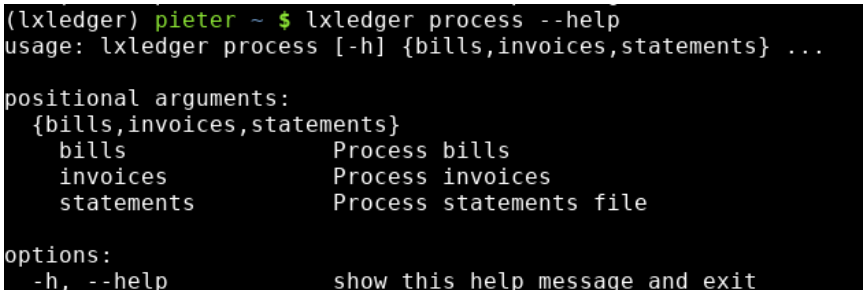

Figure 9 Output of lxledger process --help

To get help for the specific command to process bills, use *lxledger process bills --help*

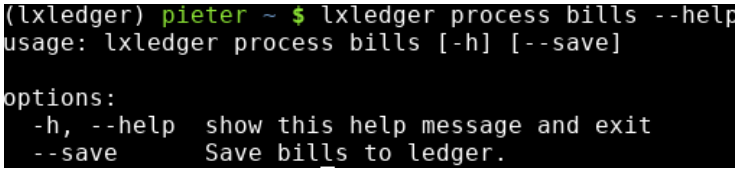

Figure 10 Output of lxledger process bills --help

This tells the user that the final command to process bills and save them to the ledger is *lxledger process bills --save.*

The language is natural and does not reveal the technical implementation detail. To be consistent, the other commands follow the same pattern. First iterations of the design failed in that regard. For instance, bank account statements were initially placed under the ”nordea” subcommand. This broke both the consistency of the UI (lxledger <verb> <object>) and failed to hide that the statement processing depends on the nordea module.

Data manipulation

In addition to the actions performed by the CLI, the lxledger software requires data in the form of entity information and bill or invoice data. To adhere to KISS, these are maintained through user interfaces familiar to the user and that provide the user with many options. Providing users with many interface options is seen as placing the user in control, which is one of the golden rules mentioned earlier (Mandel, 1997, p. 5-5).

Bill and invoice data are stored in separate folders with filenames that match a certain pattern. The patterns for bills and invoices were kept as similar as possible to improve the user experience. Adding bills and invoices requires copying a file to a directory and renaming it to match the patterns:

- -yymmdd-entityhandle-sumtotal.filextension for bills
- invoicenumber-yymmdd-entityhandle-sumtotal.filextension for invoices

Entity information can be added or removed by editing an INI configuration file. INI files are widely used to store configuration and have an easy-to-use syntax. INI files are also user friendly because they allow defining default values. This reduces the amount of information that needs to be listed, keeping the data overload to a minimum. An example entities INI file is listed in Figure 11.

```
[DEFAULT]
vatrate = 25.5
income_account = sales:fi:vat-general
expense_account = costs:services:fi:vat-general
payment_days = 14

[acme_holding]
name = ACME Holding Inc
entityid = FI12345678
contact_person = "Mrs. Smith"

[acme_catering]
name = ACME Catering Inc
entityid = FI12345679
vatrate = 14
expense_account = costs:services:fi:vat-medium
contact_person = "Mr. Smith"
```

Figure 11. Example entities INI file

While editing a text file to add entity information may not suit every user, it keeps things simple and suits the main intended audience of lxledger.

3.6 Lxledger architectural and technological design

The next stage in the design was to start making technical decisions. The design specification part of the technical design can be further subdivided into different categories, but whether a decision is a purely architectural decision, a design constraint, or a technology preference is considered unimportant. The main goal is to find answers to technical questions related to the implementation. To make these decisions easier, a decisions description template was used. The template is a simplified version of the Architecture Decision Description Template suggested by Pressman & Maxim (Pressman and Maxim, 2015, p. 257).

Table 2 Design decision description template (adapted from Pressman and Maxim, 2015, p. 257)

Design issue	
Category	
Resolution	
Arguments	
Alternatives	

Table 3 Design description of entity information

Design issue	How to store entity information?
Category	Data storage
Resolution	Entity information is stored in a plain text INI file that needs to be edited by the user using an editor of choice.
Arguments	NoSQL-style storage does not require complex schema maintenance. Eliminates the need for a separate UI. The number of entities that need to be stored is relatively small. Entities are mainly accessed based on their handle. No complex searches or queries needed.
Alternatives	SQLite, MongoDB

For more examples, see Appendix 2.

4 Lxledger implementation

4.1 Finding a balanced level of abstraction

Each implementation decision was made considering fundamental software design concepts such as abstraction, separation of concerns, modularity, information hiding, functional independence, decoupling and testability (Pressman and Maxim, 2015).

A key feature of the Python programming language is the concept of a module to help structure code. The main guiding principles to structure the code were KISS and separation of concerns. Based on the information from the requirements design drawings, a rough draft of the lxledger API was created. This draft was then refined as the implementation progressed. Figure 12 shows the last iteration of the lxledger API drawing.

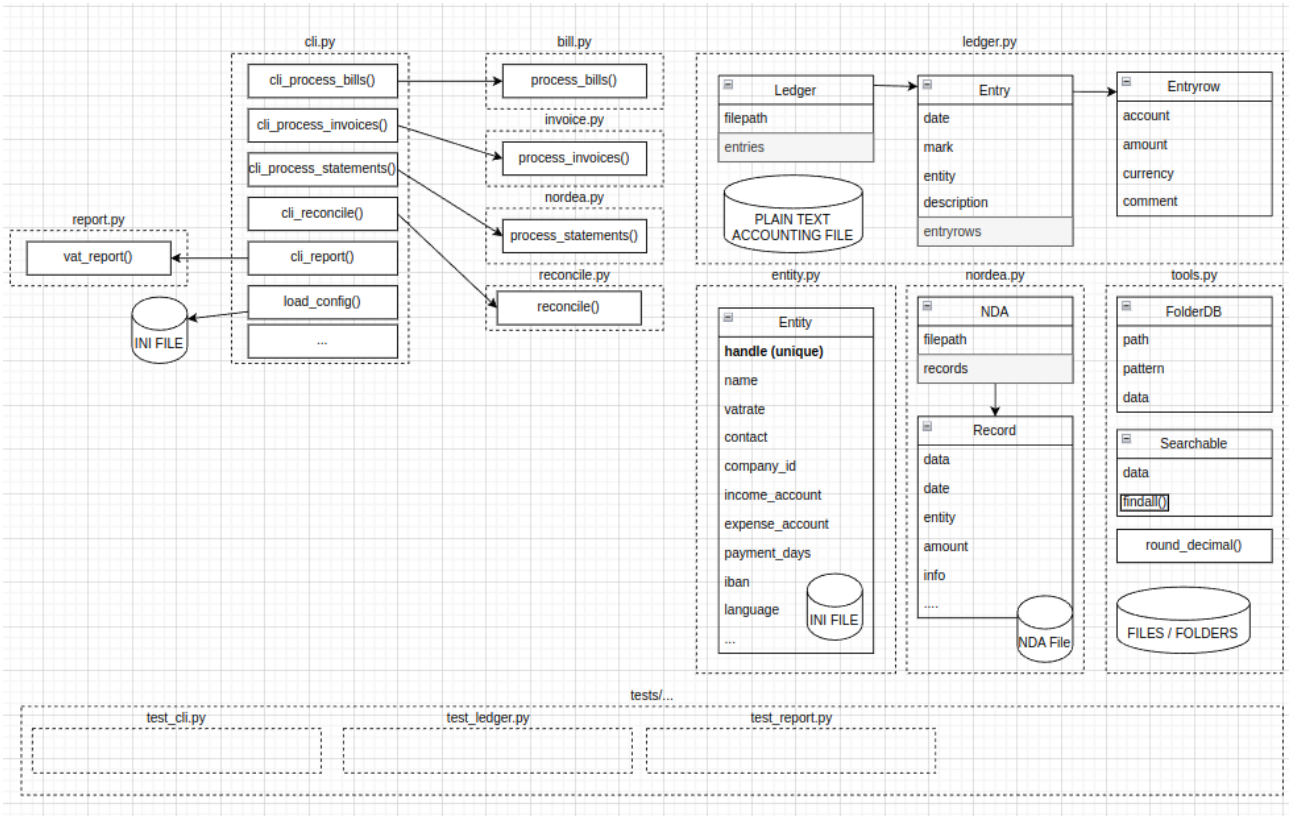

Figure 12. Last iteration of API drawing

Note the similarities between the integration design drawing (Figure 6) and the API drawing above. Like the accountant in the integration design drawing, the cli module is at the centre surrounded by the different actions it needs to perform. Each action is implemented as a separate module decoupled from the other actions. The action modules hide their implementation and only expose the

public functions needed by cli.py. For the action modules to gain access to the data they need, different data-centric modules with data classes were created. However, every action's business logic is entirely contained in the action module.

As mentioned, classes were used to represent data, since that is where they excel, but otherwise pure functions were preferred. Pure functions are functions that return the same output for the same input. In other words, there is no hidden state that can influence the outcome. This approach was chosen to improve testability and to make debugging easier. For examples of this approach, see any of the action modules. Figure 13 shows the use of the partial function in the report module.

```
def vat_report(books_fp, date):
    """return all data necessary to fill out monthly VAT report"""
    grab_balance = partial(_grab_balance, books_fp)
```

Figure 13. Example use of functional programming features.

4.2 Full lxledger source code

Note that in the following sections only some implementation details will be highlighted. The full source code is available under the MIT license at https://gitlab.com/piiteri/lxledger. Follow the readme instructions to run the code. The lxledger API documentation generated with pdoc is separately available as a GitLab web page at https://lxledger-9a8b0d.gitlab.io/lxledger.html.

4.3 Processing bills and invoices

During the design stage, bills and invoices were referred to collectively as receipts. See for instance the data layer diagram (Figure 7). Since bills and invoices contain similar data and are stored in a similar way, it made sense to create a common FolderDB class. Since the class is shared between both modules, it is part of the tools module. For more context, see the API design drawing (Figure 12). The FolderDB class implementation is listed in Figure 14.

```
class FolderDB(Searchable):
    def __init__(self, folder, regex_pattern):
        super().__init__()
        self.folder = folder
        self.regex_pattern = re.compile(regex_pattern)

    def getattr_(self, obj, attr):
        return obj[attr]

    @property
    def data(self):
        for f in os.listdir(self.folder):
            fp = os.path.join(self.folder, f)
            if os.path.isfile(fp):
                match = self.regex_pattern.match(f)
                if match is not None:
                    r = match.groupdict()
                    # insert full filepath in result
                    r["filepath"] = fp
                    r["filename"] = f
                    yield r
                else:
                    logging.info(f"{f!r} does not match regex")
```

Figure 14. FolderDB class implementation

To create a FolderDB instance, provide it with a path and a regex pattern that matches the files of interest. The data property returns a Python generator containing the regex match data enriched with the filepath. Any files that do not match the regex are logged as info messages, since that can help the user spot they have entered an invalid filename.

The invoice and bill modules create a FolderDB instance to get the information they need. This information then needs to be stored in the ledger books. To access the books, the ledger module is used.

The ledger module was the most complex module to implement. To aid its development unit tests were created (see section 4.8 Unit tests). The ledger module contains the Ledger, Entry and EntryRow classes to provide access to ledger data. For more on the specifics of this implementation, see the lxledger API documentation listed in section 4.2 above.

4.4 Processing bank statements

The nordea module exposes the process_statements function. It contains the business logic for adding new ledger entries. To gain access to the data needed, the NDA and Record classes were implemented. Since no documentation on the NDA fixed width text file format was readily available, some reverse engineering was required. This was helped significantly by the code terotil has made available on Github (terotil, no date).

Like the invoice and bill modules, once the process_statements function has gained access to the data contained in the NDA, it creates ledger entries.

4.5 Reconciling accounting entries

The reconcile module depends on the ledger module to gain access to the ledger data. It gathers the unreconciled ledger entries, which are marked with an exclamation mark, and then tries to match them to payments, which can be identified because they do not have any marks.

If the accounting entry description and entry row amounts match between a payment and a bill or invoice, it is likely they can be reconciled. In that case, both entries are marked with an asterisk.

A more complex approach to matching was also considered in which payment reference numbers would be used to reconcile statements.

4.6 Generating VAT reports

The report module exposes only the vat_report and omavero_saldo functions, but it has numerous private functions. These can be subdivided into three main categories:

- Date helpers such as _get_end_of_last_month()
- hledger access functions that run the hledger application and extract balance data from it
- plain text formatting helpers to create pleasing reports

To understand the main party trick of this module, see Figure 15.

```
def _grab_balance(books_fp, account, end_day, start_day=None):
    if start_day:
        cmd = "hledger -f {} balance acct:{} -b {} -e {}"
        return _grab_sum(_run(cmd.format(books_fp, account, start_day, end_day)))
    else:
        cmd = "hledger -f {} balance acct:{} -e {}"
        return _grab_sum(_run(cmd.format(books_fp, account, end_day)))
```

Figure 15 Implementation of the grab_balance function

The private function grab_balance runs hledger command line commands and extracts the account balance from the hledger output. The balance is then used for more complex calculations and returned as a beautifully formatted plain text report. For an example of a plain text report generated by this module, see Figure 19 part of the appendices.

4.7 Command line interface, logging and configuration

The cli.py module is the main entry point of the application. Since it is the entry point, the configuration and logging setup are performed here. Logging uses the built-in logging module. The lxledger configuration file is stored as an INI file for consistency and ease of use.

The most interesting part of the CLI implementation is the use of the Python standard library argparse module. Figure 16 contains a code sample that shows how the command and sub-command parsing with help text is achieved for the *process bills* command. The code below creates the user interface experience described in section 3.5.2.

```
parser = argparse.ArgumentParser(description="Lxledger - powerglue for hledger")
subparsers = parser.add_subparsers(dest="command", required=True)

# Process
process = subparsers.add_parser(
    "process", help="Process bills, invoices or bank statements"
)
process.set_defaults(func=cli_process)
process_subparsers = process.add_subparsers(dest="subcommand", required=True)

# Process bills
process_bills = process_subparsers.add_parser("bills", help="Process bills")
process_bills.set_defaults(func=cli_process_bills)
process_bills.add_argument(
    "--save", action="store_true", help="Save bills to ledger."
)
```

Figure 16 Implementation of process bills CLI

As can be seen in Figure 16, when the user types *lxledger process bills* and presses enter, argparse calls the cli_process_bills function and supplies it with any arguments that were provided. In this case the only available argument is the save-flag. The cli_process_bills function parses the lxledger configuration file and then calls the function process_bills from the bill module, providing it with the command line flag value (save), the bills path and the ledger books filepath. The process_bills module then takes care of the actual business logic.

4.8 Unit tests

As is conventional in python modules (Reitz and Schlusser, 2024), all tests are located under the tests subdirectory. Each module can have its own unit tests in a module called test_<modulename>. For instance, the ledger module has its unit tests in test_ledger.

Unit tests were only written for complex critical code. In this case, the module that parses the ledger plain text books and that modifies the books.

```
TEST_BOOKS_FP = get_test_fp("./_data/test.books")

class TestLedger(unittest.TestCase):
    def test_Ledger(self):
        books = ledger.Ledger(TEST_BOOKS_FP)
        self.assertEqual(len(list(books)), 7)
    def test_Ledger_find(self):
        books = ledger.Ledger(TEST_BOOKS_FP)
        results = list(books.find(desc="VENDOR", mark=""))
        self.assertEqual(len(results), 1)
        results = list(books.find(tags={"bill": None}))
        self.assertEqual(len(results), 1)
```

Figure 17 Ledger class unit test example

Figure 17 shows that a test class is created by inheriting from the Python standard library unit test TestCase class and then specifying tests for each class method we want to test. The first method in the example above verifies that the ledger.Ledger class correctly identifies 7 entries in the provided test.books file. The second method tests whether the search method finds the correct results depending on a variety of search criteria. Arguably, this unit test could be subdivided into two different tests.

To test whether all unit tests pass, we can use the ”make test” command. Assuming all tests pass, the output looks like this:

```
(lxledger) pieter ~/code/lxledger $ make test
python -m unittest discover -s lxledger/tests
.........
----------------------------------------------------------------------
Ran 9 tests in 0.005s

OK
```

Figure 18 Unit tests output

As an experiment, the code of the test_cli module was generated primarily using AI. In the end unit testing of the cli module was abandoned in favour of end-to-end testing. More information about the use of AI can be found in section 6.3.

5 Lxledger acceptance testing

5.1 End to end testing as acceptance testing

E2E testing verifies that the whole application works as it should from the point of view of the end user. For this project, this is deemed sufficient to accept the product. Acceptance testing is testing done by the end user before they accept the ordered software product as done. Ideally it should be more extensive than the E2E test plan described below.

5.2 Test plan

The E2E test plan for lxledger is to run the following commands in succession:

- lxledger process bills --save
- lxledger process invoices --save
- lxledger process statements 2501 --save
- lxledger reconcile all --save
- lxledger report vat

The expected result consists of:

- A printed VAT report.
- A ledger file containing seven entries in total.
- And four reconciled ledger entries.
- Two renamed invoice files in the invoices directory.
- Two renamed bill files in the bills directory.

Running the test requires setting up:

- An lxledger.ini configuration file specifying where the different files and directories are located.
- Ledger books containing one startup investment statement.
- An entities INI file.
- A directory containing two bills.
- A directory containing three invoices, two of which still need processing.
- A directory with a Nordea bank statements file in NDA format. The NDA file contains statements that match a payment for bill 1 and a payment for invoice 1.

To make this operation easier, an e2e template directory was created under the tests subdirectory. Running the Makefile command “make e2e” creates a fresh e2e testing setup. After running the

commands, the results are manually verified. The test passed the manual verification. Detailed test results are provided in Appendix 3.

6 Discussion

6.1 Objective 1: speed up accounting with lxledger application

6.1.1 Results

All functional requirements that were designed at the start of the project were successfully implemented. The amount of time needed for accounting has dropped significantly. The time savings increase with the number of invoices, bills or bank statements. Real world comparisons indicate that on average entering and reconciling a bill or invoice took five minutes or more without lxledger. This has dropped to about one minute per entry at most. The total time spent on accounting has dropped from one hour to about fifteen minutes per month.

In addition to a reduction in time, there is also a reduction in cognitive load. While entering bills and invoices using a predetermined pattern requires more work, this has become a routine and has reduced the number of decisions that must be made. Manually reconciling entries and comparing the numbers required more concentration and was prone to errors.

6.1.2 Future

Nordea NDA files may contain complex financial statements in which one statement contains different VAT percentages. Support for parsing these complex statements could further reduce the need for manual intervention. Instead of using Nordea NDA files, fetching data via API could further reduce the need for manual steps. Another module could be created for submitting the tax information via API to the Finnish tax authorities.

Careful coding, unit tests, E2E tests and ledger value verification using the external hledger application provide some assurance that the numbers are correct, but more unit tests and program use are needed to exterminate bugs that are lurking in the far corners of the code. The testing and logging frameworks are already in place to do so.

While the project has been published under the MIT license on GitLab, better documentation is needed to grow the project. The API documentation created by the pdoc Python library requires better code documentation to shine.

More configurability must be added. Currently the action modules contain hardcoded account names. This prohibits other users from adapting the program to their own requirements.

6.2 Objective 2: gain familiarity with plan-driven software development

Putting the time into developing various models of the application and pondering various technical design challenges took relatively little time in comparison to the benefits experienced. The biggest value was felt when determining how to structure the code. Whether this was because of the design or because the problems to be solved were not that complex is hard to tell, but in previous emergent design projects this has been considerably more difficult.

The integration design suggested by Britton proved useful. By starting with simple questions and gradually refining the design, it became easier to define the scope of the application and make important architectural design choices that benefit the current application without blocking future improvements to the entire ecosystem. Overall, decisions have felt more informed and less random.

The biggest personal takeaway is that some form of plan-based design is always worth it. There is a false dichotomy between plan-driven and agile approaches. Agile approaches also benefit from good design and architecture. The biggest difference between the approaches is in how much of the total scope is analysed in depth before starting implementation. Perhaps all software projects need to start with context and integration design. Once the integration design has been completed, an informed decision can be made on the ideal balance between plan-driven and agile.

6.3 Use of AI and sustainability

As mentioned in section 4.8 on unit tests, the test_cli unit test module was created using GitHub Copilot. Apart from the test_cli module, the use of AI to generate code has been minimal. ChatGPT was mainly used for brainstorming and to find help for specific problems. Examples of ChatGPT prompts used:

- As a senior Python programmer, how would you structure a new Python application?
- I have read that Makefiles can be useful in Python, can you give some examples of how to use them?
- What is the semantic difference between a bill and an invoice?
- I am designing a database schema and need to represent both companies and individuals by one table. What would you call this table?

The use of AI was deliberately sparse. If a search engine could provide the answer, a regular search was preferred for sustainability reasons.

6.4 Project evaluation

When submitting the thesis project plan, time was highlighted as a significant risk. Creating a functional business accounting tool from scratch was ambitious and more time was spent implementing lxledger than originally provisioned. In that sense, the project is representative of most software projects, regardless of software development project methodology (Ceschi *et al.*, 2005).

In addition to the lxledger application code, the project has provided practical software design experience and illuminated terminology related to accounting automation and software project methodologies.

References

1558/1995 | *Lainsäädäntö* | *Finlex* (1995). Available at: https://www.finlex.fi/fi/lainsaadanto/1995/1558#chp_2__sec_11v20100520__heading (Accessed: 20 April 2025).

Assad, S.N.B., Hamdan, N. Bin and Zakaria, N.B. (2024) 'The Influence of Information Technology on Enhancing the Efficiency and Effectiveness of Accounting Data', *Global business and management research*, 16(3S), pp. 280–307.

Beck, K. *et al.* (2001) *Agile Manifesto for Software Development* | *Agile Alliance*. Available at: http://www.agilealliance.org/agile101/the-agile-manifesto/ (Accessed: 19 April 2025).

Britton, C. (2016) *Designing the Requirements: Building Applications that the User Wants and Needs.* Addison-Wesley.

Ceschi, M. *et al.* (2005) 'Project management in plan-based and agile companies', *IEEE software*, 22(3), pp. 21–27. Available at: https://doi.org/10.1109/MS.2005.75.

Cooper, L.A. *et al.* (2022) 'Perceptions of Robotic Process Automation in Big 4 Public Accounting Firms: Do Firm Leaders and Lower-Level Employees Agree?', *Journal of emerging technologies in accounting*, 19(1), pp. 33–51. Available at: https://doi.org/10.2308/JETA-2020-085.

Fagarasan, C. *et al.* (2021) 'Agile, waterfall and iterative approach in information technology projects', *IOP conference series. Materials Science and Engineering*, 1169(1), p. 12025. Available at: https://doi.org/10.1088/1757-899X/1169/1/012025.

Hartwell, T.B. (2021) *Who's using ledger? · ledger/ledger Wiki · GitHub*. Available at: https://github.com/ledger/ledger/wiki/Who's-using-ledger%3F (Accessed: 20 April 2025).

Iwersen, L.H.L., Zem, L. and Penteado Neto, R. de A. (2023) 'CHOOSING THE BEST PROJECT MANAGEMENT METHODOLOGY FOR RESEARCH AND DEVELOPMENT PROJECTS: AGILE, WATERFALL, OR HYBRID?', *Revista Foco*, 16(11), p. e3336. Available at: https://doi.org/10.54751/revistafoco.v16n11-112.

Keys, B. and Zhang, Y. (James) (2020) 'Introducing RPA in an undergraduate AIS course: Three RPA exercises on process automations in accounting', *Journal of emerging technologies in accounting*, 17(2), pp. 25–30. Available at: https://doi.org/10.2308/jeta-2020-033.

Khoza, L.T. and Marnewick, C. (2020) ‘Waterfall and Agile information system project success rates - a South African perspective’, *South African computer journal = Suid-Afrikaanse rekenaartydskrif*, 32(1), pp. 43–73. Available at: https://doi.org/10.18489/sacj.v32i1.683.

Korhonen, T. *et al.* (2020) ‘Exploring the programmability of management accounting work for increasing automation: an interventionist case study’, *Accounting, auditing & accountability journal*, 34(2), pp. 253–280. Available at: https://doi.org/10.1108/AAAJ-12-2016-2809.

Lakoff, G. and Johnson, M. (1980) *Metaphors we live by*. Chicago: University of Chicago Press.

Mandel, T. (1997) *The Elements of User Interface Design.* Wiley.

Michael, Simon and contributors (2024) *What is Plain Text Accounting ? - plaintextaccounting.org*. Available at: https://plaintextaccounting.org/What-is-Plain-Text-Accounting#plain-text-accounting-apps (Accessed: 19 April 2025).

Port, D. and Bui, T. (2009) ‘Simulating mixed agile and plan-based requirements prioritization strategies: proof-of-concept and practical implications’, *European journal of information systems*, 18(4), pp. 317–331. Available at: https://doi.org/10.1057/ejis.2009.19.

Pressman, R.S. and Maxim, B.R. (2015) *Software engineering: A practitioner’s approach.* 8th edn. New York: McGraw-Hill.

terotil (no date) *Processor for Nordea machine readable bank statements (NDA format).* Available at: https://gist.github.com/terotil/4505776 (Accessed: 19 April 2025).

Rawashdeh, A., Bakhit, M. and Abaalkhail, L. (2023) ‘Determinants of artificial intelligence adoption in SMEs: The mediating role of accounting automation’, *International journal of data and network science (Print)*, 7(1), pp. 25–34. Available at: https://doi.org/10.5267/j.ijdns.2022.12.010.

Reitz, K. and Schlusser, T. (2024) *Structuring Your Project — The Hitchhiker’s Guide to Python*. Available at: https://docs.python-guide.org/writing/structure/ (Accessed: 19 April 2025).

Remlein, M., Nowak, D. and Romanchuk, K. (2024) ‘The benefits of implementing Robotic Process Automation in Accounting’, *Zeszyty teoretyczne rachunkowości*, 48(3), pp. 133–153. Available at: https://doi.org/10.5604/01.3001.0054.7260.

Spurrier, G. and Topi, H. (2023) ‘Teaching How to Select an Optimal Agile, Plan-Driven, or Hybrid Software Development Approach: Lessons from Enterprise Software Development Leaders’, *Journal of information systems education*, 34(2), pp. 148–178. Available at: http://eric.ed.gov/ERICWebPortal/detail?accno=EJ1390455.

Thesing, T., Feldmann, C. and Burchardt, M. (2021) ‘Agile versus Waterfall Project Management: Decision Model for Selecting the Appropriate Approach to a Project’, *Procedia computer science*, 181, pp. 746–756. Available at: https://doi.org/10.1016/j.procs.2021.01.227.

Thummadi, B.V. and Lyytinen, K. (2020) ‘How Much Method-in-Use Matters? A Case Study of Agile and Waterfall Software Projects’, *Journal of the Association for Information Systems*, 21, pp. 864–900. Available at: https://doi.org/10.17705/1jais.00623.

Ünal, C., Öztürk, E.N.D. and Demirag, A. (2023) ‘Analysis and Comparision of Waterfall Model and Agile Approach in Software Projects’, *Academic Journal of Information Technology*, 14(54), pp. 183–203. Available at: https://doi.org/10.5824/ajite.2023.03.002.x.

Hardenberg, P. van (2024) *Why Can’t We Make Simple Software? - Peter van Hardenberg - YouTube*. Available at: https://www.youtube.com/watch?v=czzAVuVz7u4 (Accessed: 19 April 2025).

Tobi (2012) *Why Waterfall was a big misunderstanding from the beginning – reading the original paper – Journeys of a not so young anymore Software Engineer*. Available at: https://pragtob.wordpress.com/2012/03/02/why-waterfall-was-a-big-misunderstanding-from-the-beginning-reading-the-original-paper/ (Accessed: 19 April 2025).

Appendices

Appendix 1: Lxledger source code

The full source code created as part of this thesis is available under the MIT license at https://gitlab.com/piiteri/lxledger. Follow the readme instructions to run the code. The lxledger API documentation generated with pdoc is separately available as a GitLab web page at https://lxledger-9a8b0d.gitlab.io/lxledger.html.

Appendix 2: Design decision descriptions

Below is a selection of technical design questions that were answered using the decision description template.

Table 4 Design description of programming language choice

Design issue	What programming language to use?
Category	Technology and tools
Resolution	Python 3
Arguments	Project member proficiency with Python. Large standard library likely does not require the need for extra libraries, reducing long-term maintenance.

Table 5 Design description of API documentation

Design issue	How to handle API documentation?
Category	Technology and tools
Resolution	Use the pdoc python package.
Arguments	Extremely lightweight. Generates simple HTML API documentation. Improves long-term maintenance thanks to better technical documentation.
Alternatives	Sphinx

Table 6 Design description of testing framework selection

Design issue	What testing framework?
Category	Technology and tools
Resolution	Use the python standard library unittest module.
Arguments	Well established. No extra packages required.
Alternatives	pytest

Table 7 Design description for version control

Design issue	What version control?
Category	technology and tools
Resolution	Git. Project stored on gitlab.com
Arguments	Project member familiarity with GitLab and git.

Table 8 Design description of logging solution

Design issue	How to handle logging?
Category	Technology and tools
Resolution	Use python’s logging module
Arguments	Python best practice

Appendix 3: E2E test results

Test plan commands:

- lxledger process bills --save
- lxledger process invoices --save
- lxledger process statements 2501 --save
- lxledger reconcile all --save
- lxledger report vat

Manual confirmation of the results show that the E2E ran without errors. All invoices, bills and statements were correctly imported and reconciled.

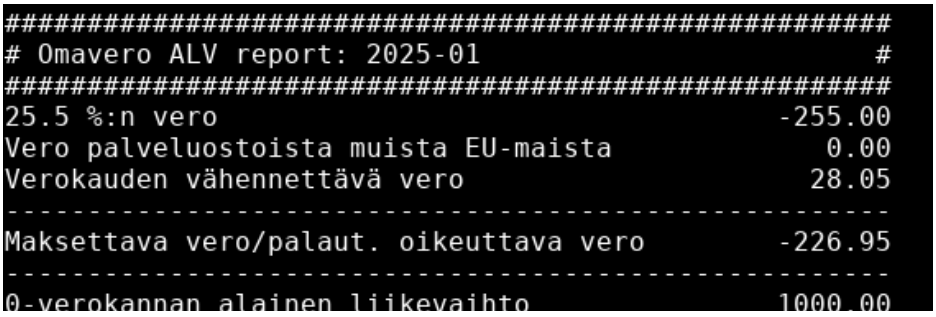

Figure 19. Vat report output

```
2025/01/05 * startup investment
    pankkitili      1000.00 EUR
    peruspääoma    -1000.00 EUR

2025/01/05 * BILL SUPPLIER 1; bill:1,due:2025/01/19
    ostovelat       -12.55 EUR
    atk:yvk          10.00 EUR
    alv-saamiset      2.55 EUR

2025/01/07 * INVOICE CUSTOMER 1; invoice:000074,due:2025/01/21
    myyntisaamiset     1255.00 EUR
    myynti:fi:yvk     -1000.00 EUR
    alv-velka          -255.00 EUR

2025/01/15 ! BILL SUPPLIER 2; bill:2,due:2025/01/29
    ostovelat       -125.50 EUR
    po:fi:yvk        100.00 EUR
    alv-saamiset      25.50 EUR

2025/01/16 * (747) PAYMENT CUSTOMER 1
    pankkitili         1255.00 EUR
    myyntisaamiset    -1255.00 EUR

2025/01/17 ! INVOICE CUSTOMER 2; invoice:000075,due:2025/01/31
    myyntisaamiset     1000.00 EUR
    myynti:eu:0       -1000.00 EUR

2025/01/22 * PAYMENT SUPPLIER 1
    pankkitili    -12.55 EUR
    ostovelat      12.55 EUR
```

Figure 22. Contents of the ledger file books.txt after executing all commands.

```
20:50 OK_000073-241217-customerY-123.txt
20:50 OK_000074-250107-customer1-1255.txt
20:50 OK_000075-250117-customer2-1000.txt
```

Figure 21. Invoices directory contents

```
20:50 000001-250105-supplier1-12_55.txt
20:50 000002-250115-supplier2-125_5.txt
```

Figure 20. Bills directory contents