An Event-Based Data Model for Extensible Accounting Systems

Solymosi Máté János
Pénzügy és Számvitel szak
Számvitel szakirány
2014

Szakszeminárium-vezető: Dr. Lakatos László Péter
Külső konzulens: Hős Attila
Én, Solymosi Máté János, teljes felelősségem tudatában kijelentem, hogy a jelen szakdolgozatban szereplő minden szövegrész, ábra és táblázat – az előírt szabályoknak megfelelően hivatkozott részek kivételével – eredeti és kizárólag a saját munkám eredménye, más dokumentumra vagy közreműködőre nem támaszkodik.
# Table of Contents

1. **Introduction** ......................................................................................................... 4

2. **An overview of existing data models** ................................................................. 7
   2.1. Double entry accounting model........................................................................... 7
       2.1.1. Essential components of the model.............................................................. 9
       2.1.2. Introduction of the business layer ............................................................... 10
       2.1.3. Redundancies in the double entry accounting model............................... 11
   2.2. Resources-Events-Agents (REA) model............................................................ 12
       2.2.1. Elements of the REA framework................................................................. 13
       2.2.2. Designing a REA-based data model........................................................... 15
       2.2.3. Redundancies in REA.................................................................................. 16

3. **Event-based data model for eliminating redundancies** ..................................... 18
   3.1. Basic features of the model................................................................................ 19
       3.1.1. Event classes and objects .......................................................................... 19
       3.1.2. Caching stock items ................................................................................... 22
   3.2. Capturing the business model............................................................................. 24
       3.2.1. Designing stock item classes .................................................................... 25
       3.2.2. Constructing event classes ........................................................................ 27
   3.3. Emulating general ledgers.................................................................................. 29
       3.3.1. The data model of general ledger emulation .............................................. 29
       3.3.2. Posting ledger entries across accounting standards .................................. 31
   3.4. Extensibility through a modular architecture.................................................... 32
       3.4.1. Managing modules ..................................................................................... 33
       3.4.2. Implementing parallel accounting.............................................................. 35

4. **Conclusion** .......................................................................................................... 36

**Appendices** ........................................................................................................... 41
   Appendix A: Figures and diagrams ......................................................................... 41
   Appendix B: Proof-of-concept implementation ...................................................... 50

**References** ............................................................................................................ 51
1. Introduction

The purpose of this thesis is to investigate the viability of using a purely event-based model for storing information in accounting systems. First, however, in order to put the discussion in the proper context, I am going to provide a brief overview of the two most researched existing accounting data models; these are the traditional double entry accounting model and the Resources-Events-Agents (REA) model. This essay is accompanied by a proof-of-concept implementation of the event-based model in the form of a simple fixed asset accounting software. Being under heavy development at the publication of this paper, this application is going to be capable of performing parallel accounting for a limited set of business events in conformance with both the IFRS and the Hungarian accounting regulations.

Designing accounting information systems in the era of computerized accounting is at least as much about software and database technologies as it is about accounting principles and methods. Therefore I attribute my choice of topic largely to my simultaneous interest in both fields, and also to my belief that there is still much room for improvement in most commercial and in-house accounting software. This belief comes from my personal experience with several accounting systems (mostly those used by SMEs), and also the opinions of experienced accounting professionals I met throughout the last few years. Like any complex software product, a typical accounting information system usually suffers from a wide range of problems, including a counter-intuitive user interface, improper access controls, or simply terrible performance. I believe, however, that, aside from critical security vulnerabilities, an incorrect, inconsistent or redundant data model is probably the most severe issue an accounting system may face, as the data model is an integral part of the system and thus cannot easily be fixed or replaced. Getting the data model right is therefore of crucial importance.

Although computerized accounting information systems have been with us for quite a while, their underlying data models tend to be based on the same basic concepts, usually closely resembling the traditional ledger-based double entry accounting model. The decision of accounting system builders to use the time-tested traditional model seems reasonable, as this system had proven to be generally useful for accounting purposes long before the era of computerized accounting, and it can also be converted into a relational database model in a relatively straightforward way. However, having such a ready-made accounting model in our hands is rather dangerous, as it might make system designers inclined to simply replicate the traditional model instead of questioning whether there are alternative approaches better suited for recording business events in a computer-based environment.

Questioning the suitability of the traditional way for storing information in computer-based accounting information systems is exactly what W. E. McCarthy did as he developed his Resources-
Events-Agents (REA) model, which advocates representing the fundamentals of a business (economic events, resources and related parties) in an entity-relationship model, instead of coercing them into the world of ledgers, accounts, debits, credits and other “artifacts” of the double-entry accounting model (McCarthry, 1979). Such a “pure” representation of business fundamentals and their relations would have been unreasonable in the era of paper-based accounting, as it does not lend itself well to physical recording and manual aggregation; the double entry accounting model has evolved with the very purpose of making these manual tasks more effective and less error-prone. Ironically, its suitability for paper-based accounting is exactly what makes the traditional system less than ideal for storing information in computer-based systems, as it has inherent redundancies that are not necessary for the computerized storage and processing of accounting data.

The development of accounting data models did not stop with REA, however. Two of the more intensively cited examples are the Triple Entry Accounting system (Ijiri, 1986) and the Items-Agents-Cash (IAC) model (Ferran & Salim, 2004), both of which aim to find an alternative to the double entry model, but with many of its redundancies eliminated. Although I understand that a thorough look on this subject would require the discussion of these additional models, the length limitations of this paper sadly prevent me from discussing them in greater detail. I would note, however, that many of these radically new approaches never made it into wide-scale production, as their implementation was, and in many cases still is, unfeasible using the technologies at hand. I also recognize the fact that the models about to be explained might seem rather aged at first glance; indeed, both REA and the computerized version of the double entry model have been around for several decades. I was, however, unable to find evidence of significant leaps in accounting data modeling since their publication, which may indicate that either most new research is now performed privately at accounting software publishers, or that most of the low-hanging fruit has been picked already, and the focus of researchers has shifted to issues around the implementation of existing models.

An important characteristic of software data models is the trade-off between normalization level and query performance; querying perfectly normalized and therefore redundancy-free relational data models usually takes several table join operations, which can be quite costly, especially if they involve a large number of rows (Codd, 1970). That is why, in the past when CPU performance and memory size were severe limiting factors, some degree of redundancy used to be welcome in accounting data models, especially if it meant a noticeable improvement in system performance (Date, 2003; Ferran & Salim, 2004). Today, in the world of cloud computing and rapidly decreasing hardware prices, however, the strategy of deliberate denormalization is becoming less and less relevant, which makes the use of well-normalized data models more feasible as they do not require frequent consistency verification and reconciliation. That is why I designed my data model to be as normalized as possible, which, however, does not necessarily result in poor performance as several other techniques, such as
result caching, exist in addition to denormalization for improving the speed of execution. These techniques come with their own share of problems and caveats, but as long as those are well understood and properly addressed, in my opinion they constitute a better alternative than degrading the data model.

As my objective was to design a data model for recording accounting events which is as normalized as possible, I decided not to wholly adopt an existing model (such as REA or IAC) as the basis of my work and then just make some improvements to it. The reason behind this decision was not that none of these models were up to the task; I believe most of them can be put to great use in computer-based systems; instead it was my personal insistence on starting the design process from scratch and carefully evaluating the underlying assumptions before accepting their validity. That of course does not mean that I did not adopt any ideas from previous research; in fact, the object-oriented approach I used to design my model has many similarities with the entity-relationship model employed in REA. In conformance with the thesis requirements, I indicated the sources of these ideas at every occurrence. I would also like to emphasize that, since this is an undergraduate-level thesis, any assumptions or conclusions stated herein are based on the knowledge I acquired during my undergraduate studies, as well as the limited amount of practical experience I gathered using several real-world accounting software systems in the past few years. Lacking extensive professional experience, I am not sure, nor can I provide evidence for whether my event-based model would be viable for wide-scale deployment in real-world systems. For now, it should be considered as a rather rudimentary experiment for replacing the general ledger with a semantic business model.

Apart from the introduction and the conclusion, this thesis is structured around two main chapters, both of them subdivided into several sections. The next chapter provides an overview of popular accounting models by summarizing previous research in the traditional double entry model and the Resources-Events-Agents (REA) model. This is followed by the third chapter in which I describe in greater detail the properties and structure of the event-based data model which is the main subject of the thesis, as well as explain the concepts and decisions behind its design. Finally, I conclude this paper by summarizing the benefits and disadvantages of the model and making suggestions for further research in order to smooth out the rough edges of its design and implementations.

Since the length of this thesis is limited, getting deep into the details and providing a complete description is possible for neither the existing data models nor the event-based data model. Given the fact that this is an accounting degree thesis, I keep the topic focused mainly on accounting and only include the minimum amount of IT and programming-related description necessary for the adequate understanding of the content. Also, an accounting system is more than simply a data model, as it consists of several different layers of functionality, such as the presentation layer for the user interface.
or the storage layer for persisting and retrieving items from the database. Describing layers other than the data model is outside the scope of my thesis and therefore not included in this document.

2. An overview of existing data models

The purpose of this chapter is to provide a more detailed description of two existing accounting data models: the double entry model and the Resources-Events-Agents (REA) model. These models were selected for comparison because of the relative abundance of research material available on them, which suggests that these are, or at least used to be, of considerable interest to researchers. I should note, however, that while this method of selection makes sense in a theoretical discussion, the situation is not so clear-cut in practice. First, being among the most researched models does not necessarily mean that a certain model has lots of software implementations, as the industry appears to be mostly happy with the traditional double entry model, and therefore has only partially adopted ideas from newer approaches such as REA. Second, the data models used in many systems often do not even conform to any standard model, as there are several valid reasons for deviating off-course; enhancing performance by denormalization is one such example. Finally, I firmly believe that no single model exists that performs the best in every situation, as their fitness depends on the requirements, features and several other factors of specific accounting system implementations. As an example, a performance-sensitive, special-purpose system used for booking large numbers of similar computer-generated transactions would not benefit much from a complex data model featuring high-level abstractions. This is in contrast with the needs of general-purpose systems with a moderate transaction volume, which are usually supposed to be able to record events and transactions of arbitrary kind and complexity; in this case higher-level abstractions are essential for the proper structuring of accounting data.

2.1. Double entry accounting model

The double entry accounting model is one of the earliest models used in computerized accounting systems, but to this day it is employed in the majority of accounting software packages. The prevalence of the model can be attributed to several factors. First, accountants who learned and have been using double entry accounting throughout their careers can without doubt better understand and learn a model which replicates and uses the structure and terminology of the double entry system. Second, the double entry model is rather simple at its core, and it can be translated into a relational database schema in a relatively straightforward way, as long as certain complex regulatory requirements are disregarded. This is both a benefit and a disadvantage at the same time: because it is so easy to put together a simple accounting application, developers with only a rudimentary
understanding of accounting are able to create working software, but usually with glaring errors such as certain actions leaving the balance sheet in an unbalanced state. I have observed accountants use such software, and it was not a pleasant experience for them to say the least.

The final reason for the prevalence of the double entry accounting model is the fact that in the field of accounting, where errors can result in serious penalties, there is a large emphasis on the reliability of the tools used for processing accounting information; therefore, innovations in such a fundamental aspect of the system as the data model can take an arbitrarily long time to become widely used and thus implemented in a large enough number of accounting software systems. The propagation of new ideas is further hindered by the “chicken-egg” syndrome apparent in the process: in order for a new idea to prove its viability, it has to be implemented in a large number of production environments; however, large-scale deployment is only conceivable in case of ideas that have already stood the test of time, as no one wants to be the guinea pig when the stakes are so high. I believe this is one of the most prevalent reasons why the progress of implementing new concepts is so slow in the accounting software industry. Another important reason is the need for complete backward compatibility, but I will address that issue in a later chapter.

The earliest implementations of the double entry accounting model copied the paper-based model verbatim, combining the challenges of a computer-based environment with all the inefficiencies of the paper-based approach (Hős & Lakatos, 2008, p. 76). Later generations eliminated most of these inefficiencies by embracing the concepts of relational database design and introducing better normalized data models on the general ledger level, but the data redundancies between the different aggregation levels are present to this day in most accounting systems. Understanding this phenomenon is crucial for spotting the real problem with many double entry models; therefore let me illustrate the previous sentence with an example. Virtually all accounting software contain an extra abstraction layer for recording invoices, which automatically books the necessary debit and credit entries upon entering the details of an invoice. This results in both the details of individual invoices and their generated booking entries being stored permanently, disregarding the fact that the latter can be automatically computed from the former and thus represents redundant data. The invoices and their booking entries are on different aggregation levels because, while almost every invoice attribute is stored in the invoice records, only a subset of them gets recorded in the general ledger (such as the booking date and the financial amounts) (Hős & Lakatos, 2008, p. 78). Related data on different aggregation levels is usually linked with cross-references (Hős & Lakatos, 2008, p. 76), although I have seen systems lacking such references in some places, making it impossible to trace automatically booked entries back to their source record. A more detailed discussion of the redundancy issue follows later in this chapter.
2.1.1. Essential components of the model

A fundamental requirement of accounting software systems is to provide the features necessary for compliance with regulatory requirements (Hős & Lakatos, 2008, p. 74). These requirements differ between countries and accounting standards; however, they usually include for example the ability to store data in a way that allows financial statements to be assembled in a straightforward way, or the enforcement of an “append-only” journal where existing ledger entries cannot physically be edited or deleted, only corrected or canceled by a subsequent entry. A basic data model which only has the minimum number of features necessary to meet the aforementioned requirements is called a “minimal” model by Hős and Lakatos (Hős & Lakatos, 2008, p. 82). The following paragraphs contain a short description of the required elements of the minimal model, assuming the use of a relational database system.

At the core of the minimal model lies a simple representation of the chart of accounts and the general ledger. Accounts and ledger entries can be represented in the table structure shown in Figure 2-1. The chart also contains a table for recording general information about the business such as the name and address of the company, as well as another table for keeping track of the posting periods. Note that it is impossible for the ledger to end up in an unbalanced state, as the table for ledger entries requires both the credit and the debit side to be incremented or decremented by the same amount. Multi-legged ledger entries are therefore not supported directly, but this issue can be remedied by either moving each leg into its own ledger entry and thus losing the built-in protection against a mismatch between the two sides, or by booking each leg of the transaction against a special technical account, and providing a facility for linking transactions that originate from the same multi-legged transaction. In practice, the former method is usually used.

The temporal nature of certain records and their attributes also poses a challenge: since most data values can change in time, in order to ensure complete consistency and accountability, the system needs to store not only the current state of the records but also keep track of their previous values. The table structure outlined above only supports temporality for ledger entries: these can be changed by canceling the incorrect entry with a negative entry and subsequently posting the correct entry. However, support for recording changes in the attributes of other entities such as accounts and general business information records would be needed as well, as it would be quite problematic for example to change the name of the company and then have the software print the new name on older reports as well. In order to keep track of changes in these records, their models have to be extended with two additional date fields recording the beginning and end dates of their effectivity, the latter being empty for the current version of the records. The table structure of the complete minimal model is shown in Figure 2-2.
The table structure of the minimal model would be sufficient for posting any number of entries into any number of accounts, without regard to the complexity of the events being recorded, and the tables are also perfectly normalized. However, this model is hardly complete enough for use in mature accounting systems, as it lacks certain crucial features such as account grouping, distinction of temporary accounts, multi-legged entries and functions that aid the automated creation of financial statements. These features are part of what Hőş and Lakatos call “optimal” model (Hőş & Lakatos, 2008, p. 105), which is an extended version of the minimal model containing among others the features listed above. The optimal model is not of significant importance in this discussion, however, as these extensions mostly implement convenience features; they do not introduce new aggregation or abstraction layers to the accounting system. That is why even the optimal model is basically just a ledger with closing and reporting capabilities, which would most likely not meet the ever-increasing demands of today’s accounting software users in terms of feature set and level of integration.

2.1.2. Introduction of the business layer

Probably the most prominent feature the optimal model and other simple ledger systems lack is the ability to store non-financial transaction-level data related to business events in a well-structured model alongside the ledger entries. A simple example for such transaction-level data would be the due date and the line items of an invoice, the choice of amortization strategy for a certain fixed asset, or any other data which cannot be recorded in the ledger because of its non-financial nature. The storage of such additional information can be achieved by the introduction of a new abstraction layer, called business layer, to the data model, which consists of entities containing the additional data the accounting system needs to store. For example, storing the due date and line items of invoices is made possible with the previously introduced invoice model, which stores the details of invoices almost exactly as they appear on the invoice receipts; the corresponding ledger entries are then generated and booked according to the invoice records in the business layer. The invoice model is one of the simplest examples of business layer entities; in theory there are no limits for the number and complexity of entity classes and the relationships between them, although practical considerations usually prevent the business layer from growing overly complex.

Having a more or less complete business layer is what makes the difference between a simple ledger system and a full-blown accounting software product, according to my opinion. However, as business layer entities are specific to certain business events and resources, the implementation of a business layer also means that the accounting data model is no longer generic in a sense that it can capture the details of every business event in the same way. This results in a hybrid data model where certain well-defined events or resources, such as fixed assets, invoices or inventory, are stored along with their related non-financial attributes and they can be automatically booked, whereas events not supported by the business model have to be entered into the ledger manually, without the possibility
of recording event-specific details in a structured way. Certain more advanced accounting systems provide varying levels of extensibility by allowing users to define custom business layer entities, or at least custom fields for the existing models. These features may help alleviate the inconveniences resulting from an incomplete business model, despite the fact that allowing possibly non-technical users to alter the data model might significantly raise the complexity and therefore decrease the robustness of the system. The implementation of such features is also not trivial in a relational database context due to the need for entity model versioning.

2.1.3. Redundancies in the double entry accounting model

The main issue with the implementation of a business layer in a double entry accounting model is the possibility of redundancies between business model records and general ledger entries; a problem I already introduced a couple of paragraphs earlier. In a well-normalized, redundancy-free data model every piece of information is only recorded once; such systems therefore do not permanently store information which is identical or can be derived from other data in the database. The greatest benefit of normalized data models is their inherent consistency: since no data is repeated, it is not possible for two representations of the same information to become inconsistent and thus require reconciliation: a lengthy and painful process for resolving the inconsistencies in the data. Based on my experience working as a product controller at a large multinational bank, I can safely claim that the lack of need for reconciliation would probably result in huge time and cost savings for the administrative departments of most companies.

On the other hand, it should be noted that certain redundancies in the data model may serve as a control function in some situations, especially if different instances of the information originate from several different sources and thus are recorded by different subsystems. In these cases inconsistencies in the data indicate a problem in at least one of the data sources, or a processing error in one of the subsystems. This control function is exactly why the paper-based double entry accounting system is so reliable in terms of detecting human errors; if something does not add up or if the debit and credit sides are not in balance, there must be a problem somewhere. However, as such errors are very rare in well-engineered computer-based systems, a better solution would be to perform the reconciliation among the sources just before the information is first entered into the accounting system, and store only the reconciled data in a normalized form afterwards. This approach improves consistency in the data model while simultaneously retaining the benefits gained from the control function of reconciliation.

In addition to redundancies arising from the storage of the same data on different abstraction and aggregation levels, redundant instances of data may also be present within the same layer by keeping both stock and flow data on record. As stock data represents the state of an object in the system and
flow data captures the changes in the state of said object, stock data for a certain date can in theory be reproduced by replaying the changes captured in the flow records up to that date (McCarthy, 1982, p. 562), provided that every change of the object is present in the database as a flow record. To illustrate this point, assume a simple business model with only three object classes: inventory items, and inventory purchase and sale events. Given that the purchase and sale event objects contain every piece of necessary information (product, quantity, price, etc.), it is possible to deduce the current stock of the inventory by only looking at data associated with the purchase and sale events. The inventory records are therefore redundant and should not be treated as permanent data. The event-based model introduced in this thesis puts significant emphasis on eliminating this type of redundancy from the data model; therefore I will address this topic in greater detail in the next chapter. I would also note that, contrary to what its name implies, the concept of double entry accounting by itself is not a redundancy (Ferran & Salim, 2004); while the amounts on the two legs of the entries are the same, one leg cannot be derived from the other as the account numbers carry unique information.

In conclusion, the double entry accounting model is an easily understandable and implementable data model which is based on the traditional paper-based double entry accounting concepts. The model can be implemented in a normalized fashion as long as its design follows the minimal or the optimal model. However, if more than a simple ledger system is required, the implementation of a business layer on top of the general ledger is necessary. Since the ledger entries constitute the master data in this model, the business layer will inherently contain redundant information, as in most cases the ledger entries can be derived from business layer records. Redundancies may also occur if both stock and flow data exists for the same subject.

2.2. Resources-Events-Agents (REA) model

The issues of the double entry accounting model in a computerized environment were already apparent to researchers in the late 1960s, despite the fact that the use of computers for accounting purposes was still in its infancy back then. Sorter (1969, p. 16) introduced the concept of events accounting which advocated a method for assembling financial statements in a way that makes it possible to reconstruct the original events from the aggregated information, which, in spite of being mainly a reporting method, already articulated the need for the storage of non-financial data in accounting systems (Dunn & McCarthy, 1997). The design and development of data models suitable for the semantic representation of business events was made possible by advances in database modeling theory in the 1970s, most notably the introduction of the relational database model (Codd, 1970). Everest and Weber (1977) subsequently applied the relational model to accounting and noted
that the traditional double entry accounting data model did not lend itself well to normalization (Dunn & McCarthy, 1997).

Building on previous research and using Chen’s (1976) entity-relationship (E-R) modeling approach, McCarthy developed and introduced the Resources-Events-Agents (REA) data model in his 1979 and 1982 publications. REA attempts to define a framework for the development of a completely new object model, based on Sorter’s research in events accounting, for capturing financial and non-financial data about economic events and resources of businesses without using any of the traditional double entry accounting concepts, and thus without subjecting the data model to the limitations of the traditional model. Therefore, unlike the double entry model where the most important, authoritative information is represented by entries in the general ledger and the business model is just an optional layer for storing the source data behind those ledger entries, the REA approach treats business layer entities as first-class citizens of the data model. As the concept of ledgers is missing from the model, redundancies arising from storing the same data in both the ledgers and the business layer are not present in REA (McCarthy, 1979).

It should be noted that REA is not really a specific, well-defined data model in a sense that it does not consist of specific predefined entities and relationships; it should rather be regarded as a generic theoretical framework for building data models using the entity-relationship model for representing business information; a “starting point for enterprise-wide database design”, as McCarthy and Dunn put it in a later publication about REA (Dunn & McCarthy, 1997, p. 34). This generic nature is what makes the model suitable for a wide range of applications, but it also requires an extensive design phase at the beginning of each implementation process, in which the entities, relationships and attributes specific to that use case are defined. Although sample adaptations of REA for specific use cases were later showcased (Geerts & McCarthy, 1997), implementing REA usually requires more design and development effort than building upon the traditional double entry accounting model, largely due to the lack of readily available model templates for a wide range of business use cases. This is probably the reason why REA has so few complete implementations, although subsets of its concepts are implemented in several accounting systems, including many of those from well-known vendors (Ferran & Salim, 2004; O’Leary, 2004).

2.2.1. Elements of the REA framework

As its name suggests, the REA accounting model represents business transactions by capturing their associated economic resources, the attributes of the event itself, as well as the agents involved in the transaction. Having introduced the basic concepts and properties of REA, in this section I present its basic architecture, its entity types and the relationships between them. Unless otherwise noted, the contents of the entire section are based on McCarthy’s original description of the REA model.
REA differentiates between stock objects and flow transactions by defining the concept of economic resources and economic events. *Economic resources* are defined as “objects that are scarce, have utility, and are under the control of the enterprise” (Ijiri, 1975, pp. 51-52), thus representing stock objects in the model. The definition of economic resources also roughly corresponds to the definition of assets in the traditional double-entry accounting model, with the intentional exclusion of accounts receivable and similar claims, accruals and deferrals, capitalized cost of establishment and reorganization, goodwill and other technical assets created out of necessity in double entry accounting. These assets are not treated as individual resources but are derived from the rest of the data instead, such as sales and payment records in the case of accounts receivable and payable. Recording liabilities, another class of stock objects, presents a problem with the original REA model, however, as they do not fit the definition of economic resources; this issue has been solved later by the introduction of *commitments* as an “agreement to execute an economic event in a well-defined future that will result in either an increase of resources or a decrease of resources” (Ijiri, 1975, p. 130).

*Economic events* are defined as a “class of phenomena which reflect changes in scarce means resulting from production, exchange, consumption and distribution” (Yu, 1976, p. 256), thereby capturing the flow of resources between the involved inside and outside parties (agents). The relationship between resources and events is therefore called a *stock-flow* relationship, as events cause either an inflow or an outflow of resources at a certain point in time. In addition to their stock-flow relations, events also need to be related to other events of opposite nature in order to represent the inherent duality of economic transactions, whereby a decrease in a certain economic resource must be accompanied by a related increase in another resource or an inflow of revenue. This dual nature of events, which is one of the core concepts behind double entry accounting, is captured by *duality* relationships in the REA model.

In addition to the economic resources and events, information about the participants involved in economic transactions must also be recorded in accounting systems. The REA model recognizes the traditional double entry model’s general inability to store this information without using subledgers or additional attributes in the business layer, and employs the concept of *economic agents* for assigning events to customers, employees or business units. Agents inside the company being accounted for are also called *economic units*. Economic agents are usually connected to events through *control* relationships, which are “3-way associations among a resource increment or decrement (event), an inside party (unit), and an outside party (agent)” (McCarthy, 1982, p. 563). In addition, they may also be related to each other through *responsibility* relationships, which signify the hierarchy of control and
accountability within the organization. The basic entity and relationship classes of the original REA model are illustrated in Figure 2-3 (McCarthy, 1982, p. 564).

2.2.2. Designing a REA-based data model

The basic entities and relationships described in the previous section serve merely as a framework for REA-based data models, as they are too generic to be viable for use in practical implementations. A design process is therefore needed for building a customized data model upon the REA framework, which meets the requirements of the accounting system being implemented. This section presents the steps of such a design process, as outlined in the original REA paper (McCarthy, 1979; Chen, 1976), by building a relational data model for a basic sales and inventory tracking system, an example which is frequently used to illustrate the principles behind REA.

The first step in designing a data model based on the REA framework is the “identification of entity sets such as classes of objects, agents and events that exist in the conceptual world and the relationship sets that connect those entities”, and the subsequent assembly of an entity-relationship diagram (McCarthy, 1979, p. 670). In our example, cash and inventory constitute economic resources, which are linked to sale and cash receipt events through stock-flow relationships: sales decrease inventory, while cash receipts increase the amount of cash available. This is the textbook case of a duality relationship between two events, where an increase and a corresponding decrease is caused by two related events in two distinct resource sets. Note that unlike the double entry model, in REA there are no separate transactions for recording sales revenue and the cost of goods sold; these attributes are consolidated in a single sale event instead (McCarthy, 1982, p. 561). The economic agents involved in the sale and cash receipt transactions are the customer on the outside, and the cashier and the sales department of the enterprise on the inside. Using these details, an entity-relationship diagram similar to the one in Figure 2-4 can be constructed.

The diagram is not yet complete, however, as the cardinalities of the relationships also need to be determined during the design phase of the entity-relationship model (Romney & Steinbart, 2014). Three basic cardinality types exist: one-to-one, one-to-many and many-to-many. In our example the stock-flow relationship between inventory and sales would have one-to-many cardinality, as a sale may involve multiple inventory items, but a single inventory item can only be sold once. Note, however, that a more sophisticated design would probably use a many-to-many relationship, as in practice inventory objects usually represent bulks of inventory, not unique items; in addition, inventory items may be returned for a refund and then sold again to a different buyer, thus participating in a second sale event. The cardinalities of the remaining relationships can be determined in a similar fashion, the details of which are not included here due to the length limitations of this thesis. Figure 2-5 displays the cardinalities of every relationship in our example model.
Armed with an entity-relationship model, the second step in building our data model is to define the attributes of entities and the relationships between them. The necessary set of attributes varies based on the requirements of the accounting system, which is why a clearly specified set of requirements can significantly increase the chances of a successful data model design phase. As an example, the attributes of sale events may include the date and time of the sale and the amount of revenue generated, and the attributes of inventory items may include their description and cost. It is not always clear, however, which attributes should be added to the entities and which to the relationships, especially if the relationship has one-to-many or one-to-one cardinality, where placing attributes on the relationship achieves the same effect from a relational point of view than adding them to the entity on the many side, or any one of the one sides. As the entity-relationship model will be converted to a relational database schema in the next step, this issue fortunately only matters in the case of many-to-many relationships whose attributes would be stored in separate tables in the database.

The third and final step of the design process is the conversion of the entity-relationship model into a relational database schema. Provided that the model has been designed correctly and all attributes have been placed on the correct entities and relationships, this step can be done rather mechanically by giving each entity and many-to-many relationship its own table with its own attributes. Each entity table should use an ID column as its primary key, whereas many-to-many join tables should contain composite primary keys consisting of foreign keys which refer to the primary keys of the entities connected by the corresponding relationship. Attributes of one-to-many relationships should be moved to the entity table on the many side, whereas attributes of one-to-one relationships should be placed on the entity on the foreign side. This concludes the design process of our REA-based data model (Romney & Steinbart, 2014).

2.2.3. Redundancies in REA

As I already mentioned, REA eliminates the possibility of redundancies arising from storing the same information in both the general ledger and the business layer, since the model only has a single layer of entities and relationships representing the relevant accounting information of the enterprise. However, the problem of capturing and storing both stock and flow data, in this case information about economic resources and events, is still apparent in REA models. This theoretical deficiency has not gone unnoticed by McCarthy who recognized the significance of using flow records as the primary source of information by stating that when REA “is considered in its maximum form of temporal generality, event descriptions would be maintained perpetually as base elements of the conceptual schema. That is, detailed descriptions of all transactions would be stored indefinitely in disaggregated, individual form” (McCarthy, 1982, p. 562). He also pointed out the unrealistic nature of storing only economic events in such a way for any practical purpose; an argument which was certainly valid at the time of the model’s initial publication.
The temporal nature of economic resources, events and agents also presents a challenge during the design and implementation of REA-based models; while events have a time dimension by definition, changes in the attributes of resources and agents over time need to be captured by the model as well. The solution to this issue is more or less the same as with the traditional double entry accounting model: each entity containing stock data, in this case economic resources and agents, has to be extended with two date attributes representing the beginning and the end of the validity of its information. This results in additional complexity in both the data model and the software implementation of the accounting system, as every query involving resources or agents needs to restrict its result set to records with a beginning time earlier and an ending time later than the date against which the database is being queried.

In conclusion, the Resources-Events-Agents (REA) model is a radically different approach to capturing and storing accounting information of enterprises. Instead of relying on the artifacts and concepts of the ledger-based traditional double entry accounting model, its original version defines three core entity classes for the representation of business information: economic resources, events and agents. Entities are in different kinds of relationships with each other; stock-flow relationships exist between resources and events, duality relationships are present between two related events, and agents are involved in control and responsibility relationships. Since REA is a generic framework, a design process is required before it can be implemented, in which the entities and relationships specific to a particular use case are defined. Although REA solves the problem of data redundancies on multiple levels of aggregation by having just a single layer of entities, redundancies may appear due to the storage of both stock and flow data. The next chapter contains a description of a purely event-based data model with which I attempt to provide a solution for getting rid of the redundancies arising from these stock-flow relationships.
3. Event-based data model for eliminating redundancies

Following the overview of existing accounting data models, this chapter introduces a purely event-based framework for designing models which, when implemented in an easily extensible fashion, should suit even the more complex accounting needs of enterprises. As I already mentioned in the introduction, this event-based model is not a refinement of an existing approach, but rather the result of starting the design process from scratch and questioning the validity of almost all underlying assumptions before incorporating any of them in the model. I acknowledge, however, that certain ideas and approaches have been adopted from both REA and the traditional double entry model. In some cases this adoption was the result of a conscious decision; in other situations it was purely incidental, as I arrived at the same conclusions as the researchers of earlier models. An example for such accidental use would be the idea of representing business objects in a semantic manner instead of using a ledger-based architecture, which was part of my concept even before I began researching existing models. In addition, some of the terminology used throughout this chapter has been adopted from Martin Fowler’s book on enterprise application architecture design (Fowler, 2002).

Despite their similarities, the event-based model presented in this chapter is fundamentally different from REA and the traditional model, in a sense that it attempts to completely eliminate redundancies arising from the permanent storage of both stock and flow data in a way that should also be realistically implementable in practice; something neither of those data models accomplish. The stock-flow redundancy is prevented by placing only flow records (events) into permanent storage, and defining a mechanism for dynamically calculating the current state of stock records using event attributes, as well as caching stock objects for enhanced performance. The details of these features are explained in the next section.

It should be noted that the primary purpose of this chapter is to provide a functional overview of the basic concepts and structure of the data model; therefore, technical considerations, such as choosing the right technologies for the implementation, translating the object structure into a relational schema or improving the performance of the system, will not be covered for the most part, with the possible exception of cases where including additional technical details contributes significantly to the thorough understanding of the model. The lack of technical details is mostly the result of the length limitations of this thesis, and by no means should it imply that the matters of implementation are not important. On the contrary, the usability and performance of the resulting software depends to a large extent not on the choice of data model itself but how the chosen model is implemented. To put this in concrete terms: designing a perfectly normalized data model is not a particularly challenging mental exercise as long as its performance issues can be conveniently disregarded; however, building one that performs well is much more difficult, as it requires walking the fine line between good
3.1. Basic features of the model

As I already noted, my main goal during the design process of this event-based framework was to eliminate redundancies arising from storing both stock and flow data, and to reach the highest level of normalization that is realistically attainable under normal circumstances. To achieve this goal, I first set out to make the model perfectly normalized, without any regard to performance implications, and then worked backwards from there until a system suitable for a wide range of practical use cases emerged. The normalization process started with the identification of critical pieces of information from which everything else can be derived in an accounting system; these turned out to be the economic events happening in the enterprise. Provided that these events contain all necessary attributes to reconstruct their effects on the financial position of the business, storing stock items is no longer required, and therefore stock-flow redundancies have no chance to occur in the first place.

The idea of only storing event information as permanent data is the cornerstone of the framework presented in this chapter; hence the name “event-based”. The following sections describe the most important features of the data model using object-oriented concepts and terminology, which are closely related to the entity-relationship model employed in REA.

3.1.1. Event classes and objects

The event-based framework captures event information in event objects, which represent the economic and non-economic events happening in the enterprise. Since the model has no general ledger or another universal way of representing any kind of business event, it is only able to accommodate transactional data and event data, depending on the specific use case. Since my event-based model has a low level of redundancy, it is therefore not recommended for use in special-purpose accounting systems with a very high transaction volume. The next section explains the basic building blocks of the event-based model, and the discussion continues by showcasing how business resources, events and other objects can be effectively translated into event and item classes. The third section presents a method for the automatic generation of general ledger entries based on existing event data in the business layer. The emulation of a general ledger is an optional feature, as it is not required for the operation of a purely event-based accounting system; however, regulations in certain countries such as Hungary require the use of a conventional ledger-based double entry system for keeping the books, so there needs to be a way to satisfy this requirement. A correctly implemented general ledger emulation feature also makes it possible to perform parallel accounting across multiple accounting standards. Finally, I conclude this chapter by discussing how implementations of the model can be made reusable and extensible using a modular architecture.
events that have a corresponding event class in the system. This limitation is a result of my conscious
decision not to include such a universal layer in the model, as it would reintroduce the practical
problem of eliminating the redundancies between the two different layers of aggregation. It should be
noted, however, that, although this setup results in the inability of the system to capture unforeseen
events for which no predefined event exists, this issue can be remedied by making the data model
extensible so that the necessary event classes can be defined in a relatively effortless manner when the
situation demands.

Each event class in the event-based model has a type, a set of attributes, and references which create
relations between the event and the referenced stock objects. The type of each event is determined by
the class the event object is instantiated from; “inventory purchase” or “cash payment” are examples
for such classes. In addition, event classes are also responsible for specifying the attribute and relation
fields of their respective event objects, which is why events of the same kind are guaranteed to be
homogeneous in terms of data structure. This structural sameness makes it possible to represent
event classes in a relational database schema; however, the complex attribute types described in the
next paragraph make document-oriented databases, which allow the use of complex data structures
within a single record, a better choice for the storage of data in event-based models.

Each event attribute has a name and a value; the former being a string such as “name”, “code” or
“amount”, and the latter being an object of any kind, irrespective of whether it can be natively stored
in a database column. The responsibility of figuring out how to store complex attribute values in the
database should lie with the database storage module, as the recommended method differs across
database systems. Not requiring attributes to have simple data types such as numbers, strings or dates,
enables the use of complex field structures in events.

Among these complex values are collection structures such as field groups, arrays or hashes. Field
groups represent a set of subfields packed into a single field; this concept is similar to embedding a
whole object into a single value. Field groups can be used to group related fields together under a
common name, which makes event classes more modular and extensible. Since subfields may contain
additional field groups or collections, there are no theoretical limits to the complexity of event data
structures. Array fields contain an ordered and numerically indexed collection of values of the same
type. Hashes are similar to arrays, except that their keys can be set to arbitrary values instead of just
numbers, and the order of their items may or may not be preserved depending on the
implementation. Figure 3-1 contains examples for the field structure of simple and more complex
event classes. Note that there are three required fields on every event: the date when the event
happened (effective date or simply date), the date when the event was booked (date of entry) and the
user who created the event in the system.
Event objects do not exist in isolation, however. Since it only makes sense to record events which somehow alter the state of a resource or some other aspect of the enterprise, almost every event will be related to at least one stock item in the database; for example, a sale event would be linked to the inventory objects that have been sold. These relations are captured by reference fields on the event classes; the equivalent of foreign keys in a relational database schema. As stock items can never be treated as persistent data, they cannot be stored alongside the events that reference them. Therefore, instead of referencing the actual stock items, reference fields contain placeholder objects with only a type and a unique ID which links them to the actual objects. Since references are just regular fields behind the scenes, arrays and hashes of references can also be defined, and field groups may contain references as well. In the previous example, our sale event would contain references to at least an array of product items, the customer record of the buyer, and the cash or receivable item booked. *Figure 3-2* presents a couple of examples for stock item references.

In order to provide an authoritative track record of business events, event objects need to be immutable, which means that once they are committed to the database, they can no longer be changed or deleted, only reverted by a subsequent event. This is in accordance with the double entry accounting model, where invalid entries should be canceled instead of changed or removed from the ledger. Revert events are special event objects that contain a reference to the event being reverted. They do not have to be backdated to the date of the original event; the software should be smart enough to take the original event and its effect on stock items into account when generating reports for before the date of reversion, but exclude these effects for reports created afterwards.

It has already been established that events are the only persistent pieces of information in an event-based data model, which should make it possible to reconstruct the financial position of a business based on nothing else but the information contained in its event objects. However, in order for the reconstruction to succeed, another key piece of knowledge is required: the details of the reconstruction process itself, as to how the event data should be applied to stock items. This knowledge is represented by the source code within the event classes, which encompasses the logic for applying the effects of those events to the stock items referenced by them. These pieces of code are not part of the data model in the strictest sense, but nevertheless measures should be taken to ensure that any change in them is properly tracked, as unexpected changes may leave stock items in an inconsistent state. These measures include the versioning of event classes, whereby backdated queries are executed using code that was effective at their respective query dates instead of using the most recent version every time. Code versioning is not a trivial feature to implement, however, as newer versions are required to maintain backward compatibility with both the earlier versions of the code and every event and stock item object the old code came in contact with, which can be a huge burden in frequently changing data models.
3.1.2. Caching stock items

Armed with both the data and the source code necessary for reconstructing the financial position of the enterprise from its event information, our event-based framework now seems complete. However, as McCarthy also realized during his research of the REA model, representing accounting information in such disaggregated form is not feasible in practical, real-world implementations (McCarthy, 1982, p. 562). Solving this issue would either require the denormalization of the data model by also storing certain kinds of stock records as persistent data, or the introduction of a caching layer capable of keeping track of stock items without committing them into permanent storage. I chose the second solution, as the introduction of redundancies into the event classes would have gone against the design goals of the model. As a result, stock items are automatically generated and stored in item caches that are responsible for keeping their contents consistent with the permanently stored event information.

The internal data structure of stock items is similar to that of events: they can define the same kinds of fields for their instances, including collections such as arrays or hashes, and they may also have references to other items. The main difference between events and stock items lies in their accepted level of redundancy: while it is important for events to be as normalized as possible, the same does not apply to cached stock items, as they can be cleared out and automatically reconstructed from scratch if the item cache is ever suspected to contain inconsistencies. The only cases where inconsistent data might sneak into permanent storage are situations in which information deduced from invalid stock object data is used as input for new events. Implementations of the data model should prevent the need for such feedback of stock data wherever possible.

Item caches are empty by default. Bringing them to a consistent state with the events in permanent storage requires constructing their stock items from those event records, which is carried out by ordering all event records by their value date, and then sequentially applying them on the item cache by executing the appropriate piece of code in their event classes. For example, an inventory purchase event would increase the available quantity on the inventory record of the product being procured, while simultaneously decreasing the amount of cash available or creating a new payable item to represent the obligation to pay. After all events have been applied to the item cache, it can be queried for stock information and balances just like the accounts in a ledger-based double entry system.

Several strategies exist for managing item caches and reducing the number of queries that result in a partial or a complete reconstruction of stock items, each of them representing a trade-off between code complexity and cache efficiency. The simplest and probably the least useful strategy for item cache management is to represent the state of stock items for only a single point in time, and regenerate the entire cache every time a query is executed with a different date parameter. This
method is trivially implementable and guarantees an up to date cache every time; however, its huge performance cost makes it only marginally better than not having an item cache at all.

A better solution would be to add a time dimension to stock items and store their entire history in the event cache, thereby making it possible to query the cache for any date without the need for a complete rebuild. This can be done by adding two date fields to stock items, which signify the beginning and ending dates of their validity period. Every time an item is altered during reconstruction, its current version is closed by ending its validity period, and a new version is created in the item cache. The cache keeps track of the date up to which its items have already been constructed; queries with an earlier date parameter can therefore be served straight from the cache. However, if a request comes in for a later date, any events between the current date of the cache and that later date have to be applied to the cache before the query can be performed. Furthermore, item caches also need to be notified every time a new event is saved to the database, as the addition of a backdated event should trigger the invalidation of all stock items dated after the new event. The large performance cost of backdating events is a major weakness of this otherwise quite reasonable cache management strategy. A more sophisticated approach would require the accurate tracking of dependencies between events and stock items so that the scope of the reconstruction can be limited to only those stock objects which are actually affected by the newly created event. As dependency detection and tracking in accounting systems is a complex subject which could easily be a thesis topic in itself, I am not including the details of this cache management strategy in this paper. The proof-of-concept software implementation accompanying this paper uses the previously discussed versioned approach for keeping its item cache up to date.

In order to demonstrate the potential of the item cache architecture presented in the previous paragraphs, I conclude this section by providing an example of a feature that would be easily implementable using the event-based framework, but rather troublesome with REA or the traditional double entry accounting model. This feature enables moving certain events into a separate event branch with its own item cache, thus making it possible to experiment with events or run what-if scenarios on the database without affecting the main data set in any way. As an example, suppose a business would like to change the valuation method of some of its fixed assets from the cost model to the revaluation model. In order to make sure the switch does not cause unintended side effects in the system, its event could initially be booked into a new event branch, separate from the main set of events. The new branch would have its own item cache, which, after its construction, would contain fixed assets valued using the revaluation model. Once the correctness of stock item information in this alternate cache would be verified, the new event branch could be merged into the main set of events, thereby finalizing the switch. In addition to creating event branches, the automatic generation
of stock items also makes correcting accounting errors much easier, as the side effects of erroneous events on stock items are automatically flushed out upon the next reconstruction of the item cache.

3.2. Capturing the business model

The event-based data model appears to be similar to REA in a sense that it is a generic framework rather than a specific data model ready to be used in software implementations. Having introduced the basic building blocks of the framework in the previous section, this section attempts to showcase the design process in which the model is adapted to the specific requirements of the implementation, by constructing the necessary event and stock item classes and determining their relations. This design process is always necessary; however, major parts of it can be eliminated or made much less time-consuming by taking advantage of the modular structure of the framework, which makes it possible to extract already developed components into their own modules and reuse them in other systems. Third-party modules may also be used in order to cut down on development time and avoid reinventing the wheel over and over again. Assuming a broad set of ready-made modules available, implementing a simple system may be easily reduced to the much simpler task of piecing together various vendor-supplied modules.

The design process of an event-based data model can be approached from two directions: either stock item classes are designed first and their related event classes second, or vice versa. The first method begins by defining stock item classes in order to represent a certain state of the business, and then adds the necessary event classes to be able to move from one state to another; in contrast, the second approach starts by constructing the event classes first and adding stock item classes later as they are needed. Although both methods are equally suitable for the task at hand and should produce approximately the same result, I personally prefer to design stock item classes first, while also making sure that I prevent any possible incompatibilities with events created afterwards.

An even more important question to decide before the commencement of the design process is to what extent the object model should adopt the concepts and structures used in the double entry accounting model. An earlier example mentioned the differences between REA and the traditional model regarding their representation of receivables: the traditional model treats them as regular assets, while in REA they are not stored as resources but dynamically calculated from the sale price and the amount of cash received in return. The REA approach might seem viable for simple data models; however, the dynamic calculation of receivable amounts can quickly become unmanageably complicated as these models are extended to accommodate additional features, such as allowing receivables to include a payment schedule with multiple installments, or booking impairment charges on degraded receivables; neither of which is trivially implementable without separate receivable records. This thought experiment demonstrates that in the end, a data model which disregards the
established concepts of double entry accounting usually creates more problems than it solves. But if that is true, it naturally leads to the question as to why I am advocating the use of an event-based framework instead of the traditional double entry model.

The answer is simple: abolishing the general ledger need not necessarily mean doing away with double entry accounting altogether. The problems of most implementations of the traditional model lie in the redundancies between their business layer and general ledger, not in the core concepts or the basic structure of the double entry model; in fact, it should be entirely possible to employ the concepts of double entry accounting without using a general ledger or a chart of accounts at all. Using double entry concepts as a basis for event-based models has at least two indisputable benefits: first, accountants are able to familiarize themselves with such a system much faster, thereby reducing training costs; and second, using assets, liabilities, income and expenses as base classes for stock items makes it painless to enforce the balance between assets, liabilities and equity, as well as making it easier to generate the necessary financial reports. Staying on the beaten path and employing the concepts of double entry accounting is therefore highly recommended in event-based models.

3.2.1. Designing stock item classes

*Figure 3-3* contains the stock item classes of a deliberately oversimplified model for a merchandising business, mainly concerning cash, inventory, receivables, payables, cost of goods sold and sales revenue, with the intentional exclusion of fixed assets, equity and other non-related items. Event classes are presented in a later diagram.

There are several base classes that constitute the foundation of event-based models. Defining a single `Value` class for storing monetary amounts along with their currencies is useful, as it makes currency conversion modules easier to implement. This class does not have to be a standalone stock item, as monetary values do not have any meaning on their own, so they are never stored as independent objects in the database. `Valuable` is a base class or interface for every stock item a financial value can be assigned to; by definition, this includes everything that would be represented in the general ledger under the traditional model, such as assets, liabilities, income and expenses, which all inherit from this base class.

Provided that instances of these stock item classes contain the necessary financial and non-financial attributes, their valuation can be automated using `Valuator` classes. Most models have several valuator classes, as valuation methods tend to vary across accounting systems, standards and policies, while also depending on the type of the stock item being evaluated. The model illustrated in *Figure 3-3* implements a simple valuation framework, where the accounting policy of the enterprise is responsible for determining which valuator should be used for which item. In order to make
automatic valuation possible, valuator classes need to implement and honor the rules of their respective accounting standards, and the relevant attributes of stock items need to be revised frequently, and updated when necessary. The simplest example of valuators is the cash valuator, which simply returns the value of the corresponding Cash item verbatim. The receivable valuator knows how to calculate the book value of receivables based on their original amount and impairment, whereas the inventory valuator returns either the original cost of the inventory or its market value, whichever is lower (Act C of 2000, 56. §).

Since stock items may contain attributes of arbitrary content and structure, it is possible to define stock item classes that contain no financial information whatsoever, thereby allowing these items to benefit from the automatic caching, reconstruction and versioning features of the event-based model. The model presented above contains several stock item classes of such non-financial nature. Instances of the Currency class represent the selection of currencies available in the accounting system. Exchange rates are not stored in the currency items because they change very frequently; instead, they are provided by an object or service external to the event-based data model. The Person class serves as a base class for storing information about related parties; these items are therefore the equivalents of agents in the REA model. Persons can be either legal or natural; these choices are implemented by the corresponding subclasses of the base class. An Entity is a special legal person which represents the enterprise being accounted for; most accounting systems therefore only contain one instance of this class. Multiple entities may be used in large, integrated systems with the capability of producing consolidated financial reports; however, such a feature would require a much more complex entity model, which is also able to capture the relationships between the entities. Entities may be linked to multiple Policy objects containing policy decisions for each accounting standard used by the business. Given that these decisions and their parameters may vary across standards, each standard is provided its own policy subclass. In addition to policies, entities may also have several associated Period items which represent the accounting periods of the enterprise.

As far as the basic elements of the double entry system are concerned, the model presented in Figure 3-3 defines three separate asset classes: cash, receivable and inventory; and a single kind of liability: accounts payable. Cash is a very simple class which only contains a single value object describing its amount and currency. Receivable objects contain attributes for the associated person, the initial value of the claim and the current realizable value of the receivable, which is updated every time an impairment charge is booked or reversed. Inventory batches are a bit more complex: their type denotes whether they are raw materials, unfinished or finished goods, purchased stock or other supplies, and they also have a single associated Product with a name and a description, which in turn implies that inventory batches may only represent homogeneous sets of the same product. Furthermore, the total cost of acquisition, the net realizable value and the market value of the batch can also be stored in the
inventory object; these are used by the inventory valuator to determine the current book value of the batch using the valuation method described earlier.

By design, inventory batches must be created from a single purchase or a single batch of production, since they only contain a single total cost attribute, implying that their items have the same unit cost. Multiple batches of inventory with potentially different unit costs can be represented together using inventory sets, which contain an array of associated inventory batches, and implement the necessary logic for determining the cost of sold inventory using the FIFO or the average cost method, whichever is configured in the accounting policy or on the inventory set itself. Batches that are members of an inventory set are excluded from inventory-related queries in order to prevent double-counting them, as the inventory set already aggregates and reports their book value as a part of its own value. Lastly, the model presented in Figure 3-3 is made complete by defining two expense stock items: impairment and cost of goods sold, and a single income item: sales revenue. Impairment expense items are created when an impairment charge is booked on a receivable or an inventory, and cost of goods sold and sales revenue items are generated when an inventory item is sold.

3.2.2. Constructing event classes

The next step of the design process involves the design of event classes which contain the logic and the data fields necessary for applying their corresponding event objects to their referenced stock items in an item cache. Figure 3-4 presents the event classes needed for the event-based data model in our current example.

As the state of stock items can only be altered by applying events on them, it is apparent that, in addition to the regular economic events, certain non-economic events need to be added for creating and managing non-financial stock items as well; the term “event” therefore has a meaning broader than just economic events in the event-based model. As an example, non-financial stock items such as currencies and persons are initially created in the system by their own “register currency” and “register person” events; other non-economic events are used to create entity items and their associated policies, register the products manufactured, sold or used by the business, and so on. The structure of these technical events is very similar to that of the corresponding stock items; as such, they are excluded from further discussion in this thesis.

Before the properties of economic events, such as inventory purchases, are described in detail, it is important to take notice of an important analogy between general ledger transactions in a double entry system and the corresponding economic events in an event-based model: in most cases, references from event objects to financial stock items are direct equivalents of the corresponding credit and debit entries of a matching general ledger transaction. This imposes a requirement on the
total value of credit-like financial references to be equal to the value of those corresponding to the debit side, in order to keep the balance sheet balanced. As an example, inventory purchase events would need to be validated in order to ensure that the initial cost of the purchased inventory (debit-like reference) is the same as the total value of all associated cash payments and payable items (credit-like references). This validation must be performed individually by each event class, as there is no universal way to determine the “likeness” of references in the event-based model. Most attributes of inventory purchase events are otherwise rather self-explanatory, with the possible exception of the cash payment and accounts payable arrays, which can be used to assign any number of cash payments and payable items to the event, in order to support payments in multiple installments, or even payments part in cash and part on credit.

Inventory sale events are similar to purchase events in terms of their field structure, with the exception of referencing receivables instead of payables, and providing two separate methods for specifying the inventory about to be sold: either an inventory batch is assigned and the inventory set and quantity parameters are left blank, or an inventory set and a quantity value are specified. In the first case the whole batch is sold and removed from inventory, while in the second case only the specified quantity of the product is removed from the referenced inventory set, measuring its cost using the configured costing method. Unlike purchases, inventory sale events involve two double entry transactions, usually with different amounts: the first transaction removes the specified inventory from the books and creates a corresponding cost of goods sold expense item; subsequently, the second transaction registers a sales revenue item along with the corresponding receivable or cash payment. Similar to purchase events, inventory sale events also need to validate the values of their referenced items: the cost of goods sold must be the same as the carrying value of the inventory being removed, and the sales revenue must be equal to the total value of all receivables and cash payments received in return. Note that although VAT is not included in this model, its implementation would not require significant changes in the structure of the inventory sale event.

There are two additional events that should be of interest in this particular discussion; these occur when an accounts receivable or a payable is paid either fully or partially. The attributes of the “incoming payment” and “outgoing payment” events are rather self-explanatory; although it should be noted that a more complete model would also need to include support for prepaid revenues and expenses, an event for netting down the outstanding payables and receivables of partners, as well as a way for assigning a single payment to multiple accounts receivable or payable items.

The data model demonstrated in this section provides a simple, yet elaborate example of how the event-based framework can be used to capture both the economic and non-economic events of a business, while also representing the resources and other stock items affected by those events. I conclude this section by emphasizing again the importance of designing event classes to be as
normalized as possible, since doing otherwise would mean leaving the most significant benefits of the event-based model on the table. Contrary to events, stock item classes are free to contain redundant attributes and references in order to improve the performance of the system, as long as great care is taken to ensure that event classes update these attributes in a consistent manner.

3.3. Emulating general ledgers

The previously discussed model of persistent events and automatically generated stock items should provide a flexible way for storing accounting information without the use of a general ledger, as long as the implementation fits the requirements of end users, the event classes are properly normalized, and the field structure of stock item classes allows queries to run in an efficient manner. Implementing the event model in its current form would not meet the legal requirements for accounting systems in certain countries, however, as countries such as Hungary require the use of proper double entry accounting systems with a general ledger and a strict double entry workflow for keeping the books of businesses (Act C of 2000, 20. §).

A possible solution for complying with this requirement would be to keep two sets of accounting data: one represented in an event-based system and another stored in a general ledger and its associated subledgers. This setup is unreasonable, however, as the whole point of employing an event-based system is to prevent such redundant storage of data. Fortunately, the cache-based architecture of the event-based model makes the emulation and automatic maintenance of a general ledger and a chart of accounts possible, as event objects already contain the information necessary for determining the appropriate ledger entries, thereby making manual involvement unnecessary. Since general ledger emulation is an optional feature of the model, is at the discretion of the implementers to decide whether their software requires such functionality.

3.3.1. The data model of general ledger emulation

The proper emulation of a general ledger and a chart of accounts does not warrant its separate data model, as these features can be implemented using the existing infrastructure of events and cached stock items. As general ledger entries represent information already captured in event objects, neither accounts nor ledger entries should be part of the permanent data set; storing them as stock items is therefore an obvious choice. Figure 3-5 presents a possible data model for the emulated general ledger, featuring its stock items and the events necessary for the management of those items.

The most important stock items of the general ledger emulation feature are accounts and transactions. Account objects have a number, a name, and a type denoting whether they represent asset, liability, income, cost or expense accounts. Since stock item classes are not part of the
permanent data set and thus do not need to be perfectly normalized, the current balance of accounts can also be stored on the account items themselves. The effect of account balance caching on the performance of the system needs to be carefully measured, however, as the advantages of faster balance lookups may be more than offset by the elevated risk of cache inconsistencies and a drastically increased number of account items, since new versions of them would need to be saved on every balance change.

Accounts may also be grouped using account group items, which, aside from a code, a name and perhaps a calculated balance, also contain two arrays referencing their associated accounts and subgroups. In Figure 3-5 groups are connected to their member items in a redundant manner, by using foreign keys on both ends of the relation, so that each member contains references to every parent group it either directly or indirectly belongs to; this redundancy makes it possible to fetch whole account trees in one query.

Despite their apparent “flow” nature, ledger entries are also captured as stock items in the event-based model, as storing them as persistent data using event objects would introduce redundancies between the event-based business layer and the emulated general ledger. Transaction objects contain a receipt number, a description, a reference to the business event that generated them, and their associated debit and credit accounts and amounts. Using separate stock items for capturing individual debit and credit entries is not necessary, as they can be stored within their respective transactions using array fields. This design supports ledger entries with more than two legs; it is the responsibility of transactions to validate the balance of their credit and debit legs before they are saved to the cache.

The object model presented in Figure 3-5 features three event classes for the setup and management of the chart of accounts. The “create account” event possesses the same attributes as account stock items, with the exception of the newly added “assign to group” attribute, which specifies the group the new account should be added to. A similar event exists for creating new account groups and assigning existing accounts to them. Finally, there is a complex event which enables the rapid initialization of the chart of accounts by creating the entire account tree at once. Although it would be tempting to include the possibility of setting the opening balances of accounts with these events, doing so would result in inconsistencies between the business layer and the emulated general ledger, as every increase or decrease in the account balances has to be explained by a corresponding event in the business layer. The correct way for setting the initial balances of accounts is therefore to implement and use the appropriate business layer events for registering the existing assets and liabilities of the enterprise in the system.
3.3.2. Posting ledger entries across accounting standards

The data model of the general ledger emulation feature contains no separate event for recording new transactions in the ledger, as the addition of new transactions is triggered by the existing economic events in the business layer, provided that general ledger emulation is enabled in the accounting system. Consequently, event classes causing new transactions to be posted in the general ledger need to incorporate the necessary logic to generate and book the appropriate ledger entries by creating their respective transaction items in the item cache. For example, the source code of the inventory sale event presented in the previous section would need to include a posting instruction for debiting the cost of goods sold account and crediting the appropriate inventory account, and another instruction for debiting the relevant cash or receivable account and crediting the sales revenue account. The relevant accounting policy object should be responsible for determining the exact account numbers to use for booking these transactions. Events of non-economic nature, such as the registration of a new currency or person, obviously do not need logic for emulating general ledger entries. In order to encourage code reuse and to prevent needless duplication, common elements of the transaction posting logic should be extracted into their own classes, and subsequently used by multiple similar events.

The issue of general ledger emulation is further complicated by the fact that the rules for recording specific transactions in the general ledger may vary across accounting standards. For this reason, designers of event-based accounting systems should consider moving rules related to each standard into their own module, and making accounting policy classes responsible for choosing the right posting rule for booking any given business event in compliance with their respective standards. Embracing such modularity also makes parallel accounting according to two or more separate standards possible: if an entity has policies for more than one accounting standard, each time an event is applied to an item cache, each of those policies should be queried for the appropriate posting rule to use for general ledger emulation. Parallel accounting also demands the use of separate instances of emulated general ledgers and charts of accounts for each standard, as regulations concerning the structure and the required elements of the chart of accounts, as well as the rules for booking certain transactions, may differ across standards. For example, Hungarian regulations require the gross amounts of both the expense and revenue arising from the sale of a fixed asset to be booked separately (Act C of 2000, 15. §, s. 9.), while IFRS allows the recording of just the net profit or loss of the sale (IAS 16, 67-71).

Equipped with a modular architecture for general ledger emulation in which each accounting standard has its own module incorporating its posting rules, enabling parallel accounting should be a matter of simply installing the appropriate modules and initializing their separate charts of accounts, with
everything else automated by the system. Since every piece of data related to the general ledger emulation feature is stored in item caches, redundancies arising from maintaining multiple sets of books do not affect the consistency of the permanent data in event objects. Nevertheless, extracting standard-specific posting rules into their own modules is just one of the several refactoring steps needed for the implementation of parallel accounting, as certain elements of the business layer, such as particular event and item attributes or valuation methods, may also vary depending on the accounting standard being used. Therefore, in addition to the general ledger emulation layer, the scope of standard-specific modules needs to include the business layer as well. The discussion of this topic continues in the next section.

3.4. Extensibility through a modular architecture

The event-based framework outlined in the previous sections is generic enough to be usable for the implementation of both simple and more complex accounting systems; at least in theory. However, due to the strict separation of stock and flow records in the data model, even simple components, such as the inventory model described two sections earlier, would need about twice as many classes as an equivalent ledger-based implementation would call for, making it more likely that designers of larger, more complex applications will run into issues concerning the manageability of the large number of event and stock item classes required for the operation of such systems. On its own, the large number of classes would not be a huge issue; however, lacking proper measures towards modularity and extensibility, these classes tend to be highly coupled to each other, resulting in a large, monolithic system, whose individual components cannot be extracted and reused in other implementations without major refactoring, or bringing over additional, unnecessary pieces of code. Two components are considered to be coupled if removing any one of them causes both to break.

Embracing modularity and implementing an extensible architecture is possible in almost every layer of an accounting system; for example, a data storage layer independent from any single database engine can be constructed by designing a common interface for database operations and moving classes responsible for interacting with specific database engines out into their own modules, thereby making it possible to switch database vendors by simply changing the database module used in the system. A similar idea can be implemented in the presentation layer driving the user interface of the application: factoring the interface code out into its own module enables the use of multiple presentation modules, such as one for the desktop and one for mobile users. Making the data model extensible is possible in an identical manner, by using object-oriented design patterns to eliminate direct references between classes implementing separate components. Two of these patterns, dependency injection and the observer pattern, are heavily used in the proof-of-concept software
implementation accompanying this paper; however, their detailed explanation is, outside the scope of this thesis.

Extracting related classes into their own separate modules has several advantages: first, independent modules can in most cases be reused in other systems without requiring much additional work; second, extending a well-structured, modular architecture is much easier than making changes to a monolithic system with a high degree of interdependence between various components. Without proper refactoring along the way, the complexity of implementations can reach a point where even the smallest changes are prohibitively expensive, as any single change may possibly break a large number of other features; this usually results in an “if it still works, don’t touch it” mentality. In contrast, adding new functionality to a decoupled system with distinct modules using well-defined interfaces to interact with each other is much less risky, as it is unlikely that a change in one module will severely affect the operation of components in other, unrelated modules. The third benefit of a modular architecture is the possibility of cherry picking features and including just the necessary components in specific implementations. Being able to install and uninstall modules at will, without affecting the operation of others, is a feature whose value should not be underestimated.

3.4.1. Managing modules

Following the decoupling of event and item classes and the extraction of related classes into their own modules, a module manager is needed for loading and keeping track of modules that are available for use in the system. Since the complete elimination of dependencies between various modules is unreasonable if not impossible, the module manager also needs to perform dependency management by taking the designated dependencies of modules into account when determining which modules to load and the order of their preparation. In the example of fixed asset accounting, the IFRS fixed asset module, which contains features for representing fixed assets according to the IFRS standards, would depend on the fixed asset base module, containing basic events and items for fixed asset accounting that are independent from specific standards. In turn, the fixed asset base module may depend on the core module of the system, holding the most important base classes used by every other module, such as Value, Asset, Person, and so on. In this example, loading the IFRS fixed asset module would trigger the loading of the fixed asset base module, which in turn would load the core module of the system.

In addition to the management of module dependencies, meticulous tracking of the separate versions of modules is also necessary, as module upgrades to newer versions may cause inconsistencies or inadvertent changes in the accounting data, especially if they reference objects created before the upgrade. In case of retrospectively applied upgrades, such as bug fixes, new modules must also maintain complete backward compatibility with their predecessors and seamlessly process events and
items created by those older versions; otherwise, various sorts of consistency and validation errors may occur. Module versioning adds another layer of complexity to dependency management, as certain modules are only compatible with specific versions of the modules they depend on. Since only a single version of a module can be loaded at a time, version collisions may occur in cases where two unrelated modules depend on separate versions of the same module at the same time. These features introduce several edge cases and thereby significantly increase the complexity of the module manager; however, true extensibility cannot realistically be achieved without them.

Besides making the source code of event-based accounting applications modular by breaking it apart into separate modules, the data model itself should be extensible as well, in order to let modules extend the fields of event and item classes in other modules. To understand the need for extensible field structures in events and stock items, consider the example of storing VAT rates and amounts in invoice-related records. Since VAT is not applicable in certain countries such as the United States, extracting VAT-related components into their separate module is advisable; however, the complete separation of VAT-related fields is impossible as long as the field structures of event and item classes are static and cannot be conditionally extended depending on whether the VAT module is loaded. Making field structures dynamically extendable lets the VAT module extend invoice event and item classes with the necessary VAT-related fields after it has been loaded by the module manager. If the VAT module is not loaded, classes related to invoice management are not extended, VAT-related invoice attributes returned from the database are disregarded, and the calculation of VAT is not performed.

It should be noted, however, that the implementation of dynamic event and stock item field structures carries the danger of separate modules attempting to insert multiple identically named fields to the same class unbeknownst to each other, and thereby causing name collisions. In order to avoid these situations, certain design rules must be established. As the event-based data model supports the notion of field groups, the simplest answer to the problem would be to create a field group inside the host class for each module that extends that class, and add the custom attributes of the module to that group. I call this technique property namespacing, as the module-specific field groups act as namespaces for the inserted attributes. If the host class needs to be extended with a more complex field structure, it is also possible to implement that structure in a separate model class, called extension class, and only extend the original class with a single field, specifying the extension class as its type. These techniques help shield the inserted attributes from name collisions, thereby partially alleviating the negative impact of dynamic event and stock item field structures on the reliability and robustness of event-based accounting systems.
3.4.2. Implementing parallel accounting

Presumably nothing benefits more from code and data modularity and extensibility than the implementation of parallel accounting: the simultaneous storage of booking entries and preparation of financial reports according to the regulations of multiple accounting standards. Parallel accounting is usually implemented in a highly redundant fashion in double-entry accounting systems, such as by storing a separate set of books for each standard, or in a more sophisticated way, by only duplicating accounts containing entries that differ across standards. Nevertheless, both solutions require frequent reconciliation in order to ensure that the same information is present in each set of books or accounts. In contrast, the caching architecture and the general ledger emulation feature of the event-based framework should make it possible to eliminate most redundancies in parallel accounting data models. The next few paragraphs provide a short overview of the functions and components required for recording the events and stock items of an enterprise in conformance with multiple separate accounting standards.

Adding support for parallel accounting in an event-based model requires the extraction of standard-specific code and data attributes into their own standard-specific modules, henceforth called standard modules, leaving only classes independent from specific standards in the base modules. Standard modules should contain at least a general ledger emulation component, their own set of valuator and policy classes, as well as logic for extending base events and items with the necessary standard-specific attributes.

General ledger emulation components usually contain their own posting rules for recording economic events in the ledger, as well as their own charts of accounts, since the naming and numbering of accounts may be regulated in certain countries. For example, a German company with a subsidiary in Hungary would need to use at least two different charts of accounts if a central event-based accounting system were to be used for keeping the books of both entities, or preparing the consolidated financial reports of the group.

Custom policy classes are also vital components of standard modules, as they contain among other things the necessary logic for deciding which valuation classes to use for certain types of assets, liabilities and other Valuable items. Since some valuation methods are used by multiple standards, only methods used by a single standard should be factored out into their standard module; the rest, such as the cash and the receivable valuators, should be left in the base modules. Both the general ledger emulator module and the standard-specific valuator classes might require additional attributes to be added to certain events or stock items, in order to be able to book them or determine their current book value; examples include the type of a cost item according to the Hungarian regulations, or the net realizable value of an inventory item, which is used by the IFRS inventory valuator class.
Fortunately these supplementary fields can be added to their respective events and items in a straightforward manner, by extending their classes when the standard module is loaded.

Using the techniques outlined in the previous paragraphs alongside with an event-based data model, the construction of a mostly redundancy-free data model is possible for parallel accounting: as general ledger emulation results are stored in the item cache and therefore do not constitute persistent data, redundancies between multiple sets of books are not a problem, as long as the separate posting rules of accounting standards are consistent and compatible with each other. Events remain the only permanent set of information in the accounting system, extended with additional attributes required by standard-specific general ledger emulation modules and valuator classes, and controlled by the accounting policy classes of the standards.

In this chapter I presented the basic concepts and design decisions behind an event-based approach for representing accounting information in a computerized environment, aiming to eliminate the redundancies between stock and flow records; a phenomenon apparent in most implementations of the traditional double-entry data model. Techniques for making event-based models modular, reusable and extensible were also covered, as they are fundamental for the construction of complex accounting systems with a diverse set of features. I leave drawing the final conclusions and discussing the benefits and disadvantages of the model for the upcoming last chapter.

4. Conclusion

This chapter concludes my thesis by providing a short summary of the previously discussed topics, presenting the most important benefits and disadvantages of the event-based data model featured in the previous chapter, and recommending additional areas and issues for further research.

The elimination of data model redundancies has been chosen as the central topic of discussion throughout the previous chapters, as the issue of redundancies severely affects many existing accounting system implementations, and their elimination is one of the main design goals of the event-based data model as well. For the sake of simplicity, this thesis distinguished between two kinds of redundancies: those present between various layers of aggregation and abstraction, and redundancies arising from storing both stock and flow records of an enterprise.

The second chapter provided an overview of the two most researched existing data models: the traditional double entry accounting model and the Resources-Events-Agents (REA) model. At the core of the double entry model lies the general ledger, which represents the main source of information in the accounting system. Constructing a simple ledger-based system is relatively straightforward using the minimal or the optimal model described by Hős and Lakatos (2008); care
should be taken, however, in order to ensure that the implementation meets the regulatory requirements of the target environment.

The next step in the evolution of double entry data models was the introduction of a business layer, containing entities responsible for the semantic representation of both financial and non-financial accounting information. This resulted in accounting environments where certain events were handled and automatically booked by the business layer, whereas events not supported by the business layer would have had to be manually added to the ledger. The existence of both a general ledger and a business layer makes the traditional double entry model susceptible to redundancies between the two layers of aggregation, as most entries in the general ledger are derived from information already existing in the business layer. Although techniques for preventing this kind of redundancy exist (database views, for example), the easier path of storing both instances of data separately is usually chosen in practice.

The Resources-Events-Agents (REA) model attempts to solve the issue of redundancies between different layers of aggregation by completely removing the general ledger from the equation, and defining concepts for the semantic representation of economic events, resources and related parties (agents) of a business using an entity-relationship model (McCarthy, 1982). Since REA is a generic framework in a sense that it only defines the core concepts of the data model, a design process is required, in which the entities and relationships of a business model suitable for a particular use case are created. The construction of REA-based data models has its own challenges, however, as the framework deviates on many counts from the basic ideas of double entry accounting, making achieving compliance with regulatory requirements troublesome if not impossible in more complex scenarios; not considering certain kinds of assets, such as receivables, as separate resources, but deriving them from other information would be an example for such a deviation (McCarthy, 1982).

These factors and the lack of readily available model templates for a wide range of business use cases together resulted in a lack of purely REA-based implementations; although certain concepts of the framework are used in some accounting systems (Ferran & Salim, 2004; O’Leary, 2004). Lacking a general ledger, REA solves the issue of redundancies between various levels of aggregation; however, it is prone to redundancies arising from the storage of both stock and flow data.

The event-based data model introduced in the second chapter attempts to address the issue of redundancies between stock and flow records by only storing the events of an enterprise as permanent data, but not the state of its resources or other information obtainable by aggregating the event data. Since the storage of accounting information in such disaggregated form would be unreasonable in terms of performance, the framework proposes the concept of item caches, which store non-persistent stock items, representing the state of the business at certain points of time. Item
caches are populated automatically by constructing their stock items using information available in the event records; the knowledge required for applying the effects of these events onto an item cache is incorporated in their corresponding event classes. Since item caches and their stock items do not constitute persistent data, they are free to contain redundancies for enhancing the performance of the accounting system; if they are ever found to be inconsistent, they can be automatically reconstructed from the event records at any time.

Despite the event-based framework not yet having been used in practical implementations, and the lack of evidence confirming its suitability in a wide range of real-world use cases, I believe that it is useful to enumerate some of its possible benefits and disadvantages. The strict separation of event and stock item records and the caching architecture present in the framework are clearly advantageous, as they make perfectly normalized data models technologically attainable, without significantly impacting the performance of the system. However, this division of concerns can result in a much higher number of event and stock item classes than what would be required in a double entry model with a comparable set of features, thereby increasing the complexity of the system, and articulating a greater demand for modular software architecture. Anticipating the need for the latter, several techniques and design patterns for extracting related components into their own modules are presented in the last section of the third chapter.

In addition to lending itself well to normalization, the caching architecture of the event-based model also makes the emulation of a general ledger possible, thereby satisfying the relevant requirements of certain countries and accounting standards. By utilizing item caches, parallel accounting can also be implemented in a more or less redundancy-free manner, provided that standard-specific data attributes and pieces of code are factored out into independent accounting standard modules.

Resulting from the design decisions made during its development, the event-based framework also has a couple of inherent disadvantages, which might make it unsuitable for use in certain situations. Probably the most significant drawback of the model is its similarity to REA in terms of its lack of a universal way of representing arbitrary business transactions, and therefore its reliance on predefined event classes. Lacking a general ledger, event-based systems are only able to represent a finite set of event types, for which event classes have already been implemented. This problem can be somewhat alleviated by making the data model easily extensible by providing privileged users a facility for constructing new event classes using a custom domain-specific language tailored to the definition of accounting events, instead of achieving the same outcome by writing new code.

In addition to its lack of a generic way to store financial information, other features of the event-based model might also turn out to be disadvantageous in certain cases; examples include the need for an extensive design process during the implementation of the model, the previously mentioned
increased number of event and item classes, or the added complexity resulting from the existence of a caching layer and having to choose from several different caching strategies, each with different properties, benefits and drawbacks. It is probably safe to assert that the implementation of an event-based data model generally involves a higher number of unfamiliar decisions and dilemmas than that of the traditional double entry accounting model. Whether the benefits of the model offset these disadvantages should be evaluated on a case by case basis.

As far as my suggestions for further research are concerned, the individual features of the event-based framework present ample opportunities for additional investigation and collection of practical experience, given that the model is in a rather rudimentary form at this time. One of the most important open questions is related to the implementation of the more advanced features of the model in relational database systems in a way that is not detrimental to the performance of the application; such features include the definition of complex attributes, such as arrays, hashes and field groups, and the extensibility of event and stock item classes. Fortunately, these issues of representation can be completely circumvented by using document-oriented database systems for storing event and stock item records; the properties of which should be carefully investigated before the decision to use this kind of database is made.

Another area suggested for further research is cache management, as the caching strategies described in this thesis are rather primitive and therefore offer much room for improvement. A significant leap in cache management would be the implementation of event dependency tracking, which could lead to a radically reduced number of necessary cache rebuild operations. However, a prerequisite for dependency tracking is the ability to perform static program analysis on the logic that applies the effects of events to an item cache; as previously mentioned, a domain-specific language for the static definition of accounting events is an important requirement for this feature. As a final suggestion, the viability of the event-based model for performing complex accounting tasks, such as preparing consolidated financial reports in multi-entity systems, should also be thoroughly researched, in order to get an idea of where the limits of the framework lie.

In the end, I argue that creating a working implementation of the event-based model, and collecting relevant experience by testing it in practical situations would be of much more help in determining its suitability in real-world scenarios than any amount of additional theoretical research could ever be. Testing the system in real-world conditions is not expected to be easy, however, as the introduction of new concepts to accounting systems is at least as much about changing the established practices and habits of accountants and finance departments, as it is about properly handling the inherent technological challenges of the resulting changes. We need to face the economic realities of the adoption of new software: unless the opportunity cost of not switching exceeds the costs and risks of adopting the new approach, companies are going to continue using their old, time-tested systems.
The cost of adoption can be greatly reduced by ensuring complete compatibility with existing systems, as well as making the user interface similar to that of traditional accounting software applications, and thereby increasing the confidence of users in the new system, improving productivity, and reducing training costs.

As the pace of change increases in both business and regulatory environments, problems existing in many existing accounting information systems are going to become more and more pressing. In turn, these systems are going to need to become more robust, more integrated, more adaptive and easier to customize or extend with new features, so that they are an asset of enterprises, rather than a liability. The event-based data framework presented in this thesis aims to support the development of systems with these qualities; however, only real-world implementations of the model will confirm or deny its suitability for such purposes.
Appendices

Appendix A: Figures and diagrams

Figure 2-1

This chart presents the core table structure of the minimal model, based on the work of Hős and Lakatos (2008, p. 151). Primary keys are set in bold, foreign keys are shown in italic.
Figure 2-2

As an extension to Figure 2-1, this chart presents the complete table structure of the minimal model, based on the work of Hős and Lakatos (2008, p. 151). Although not discussed in this thesis, the Ledger View and View / Account tables are also part of the model, making it possible to create custom views of the general ledger. The Entry Relation table is used for linking negative cancellation entries with their original counterparts. Primary keys are set in **bold**, foreign keys are shown in *italic*. 
**Figure 2-3**

This diagram presents the basic entities and relationships of the Resources-Events-Agents (REA) model, as originally published by McCarthy (1982, p. 564). Entities are displayed in rectangles, relationships are shown in diamonds.

**Figure 2-4**

As a concretization of Figure 2-3, this diagram presents the entities and relationships of an inventory sale transaction using the REA model, based on a similar chart published by McCarthy (1982, p. 566). Entities are displayed in rectangles, relationships are shown in diamonds.
Figure 2-5

This diagram extends Figure 2-4 with the necessary cardinalities of the relationships in the model (McCarthy, 1982, p. 566). The many side of relationships is represented with the letter N, the one side is identified with the number 1.
Figure 3-1

This chart presents a couple of examples for the field structure of economic (received payment, purchase inventory, sell inventory) and non-economic (register currency, register address) event classes. The columns on the left side contain field names, while the columns on the right designate the types of the fields. Reference fields are marked with an arrow (►) and specify the types of the referenced stock items. *Value* is a composite type containing a monetary amount and its currency.

<table>
<thead>
<tr>
<th>Register Currency</th>
<th>Purchase Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Date</td>
</tr>
<tr>
<td>Creator User</td>
<td>Creator User</td>
</tr>
<tr>
<td>Date of Entry</td>
<td>Date of Entry</td>
</tr>
<tr>
<td>Name</td>
<td>Partner</td>
</tr>
<tr>
<td>Code</td>
<td>Assign Invoice</td>
</tr>
<tr>
<td>Currency</td>
<td>Assign Inventory Set</td>
</tr>
<tr>
<td></td>
<td>Product</td>
</tr>
<tr>
<td></td>
<td>Quantity</td>
</tr>
<tr>
<td></td>
<td>Price</td>
</tr>
<tr>
<td></td>
<td>Inventory</td>
</tr>
<tr>
<td></td>
<td>Cash Payments [ ]</td>
</tr>
<tr>
<td></td>
<td>Accounts Payables [ ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Sell Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Date</td>
</tr>
<tr>
<td>Creator User</td>
<td>Creator User</td>
</tr>
<tr>
<td>Date of Entry</td>
<td>Date of Entry</td>
</tr>
<tr>
<td>Country</td>
<td>Partner</td>
</tr>
<tr>
<td>Postal Code</td>
<td>Inventory Batch</td>
</tr>
<tr>
<td>City</td>
<td>Inventory Set</td>
</tr>
<tr>
<td>Street</td>
<td>Assign Invoice</td>
</tr>
<tr>
<td>Address</td>
<td>Quantity</td>
</tr>
<tr>
<td></td>
<td>Price</td>
</tr>
<tr>
<td></td>
<td>Cost of Goods Sold</td>
</tr>
<tr>
<td></td>
<td>Cash Payments [ ]</td>
</tr>
<tr>
<td></td>
<td>Accounts Receivables [ ]</td>
</tr>
<tr>
<td></td>
<td>Receivables [ ]</td>
</tr>
<tr>
<td></td>
<td>Receivable Amount</td>
</tr>
<tr>
<td></td>
<td>Receivable Value</td>
</tr>
<tr>
<td></td>
<td>Receivable (array)</td>
</tr>
<tr>
<td></td>
<td>Receivable (array)</td>
</tr>
<tr>
<td></td>
<td>Receivable (array)</td>
</tr>
<tr>
<td></td>
<td>Receivable (array)</td>
</tr>
<tr>
<td></td>
<td>Sales Revenue</td>
</tr>
<tr>
<td></td>
<td>Sales Revenue</td>
</tr>
</tbody>
</table>
Figure 3-2

This diagram presents the stock item references of two example events involved in an inventory sale transaction and the subsequent collection of payment. The gray rectangles represent stock items, with their type displayed in bold. The columns on the left side of the tables contain field names, while the columns on the right designate the types of the fields. Reference fields are marked with an arrow (►).
Figure 3-3

This chart presents the stock item classes of the example data model for inventory management. Inheritance is signified by the lines between the classes. The columns on the left side of the tables contain field names, while the columns on the right designate the types of the fields. Reference fields are marked with an arrow (►).
Figure 3-4

This diagram presents the event classes of the data model shown in Figure 3-3. For the sake of simplicity, certain less interesting classes have been omitted. Inheritance is signified by the lines between the classes. The columns on the left side of the tables contain field names, while the columns on the right designate the types of the fields. Reference fields are marked with an arrow (►). Event acts as a base class for every event class and contains fields required by all events.
This chart presents the event classes and stock items required for the implementation of general ledger emulation. Inheritance is signified by the lines between the classes. The columns on the left side of the tables contain field names, while the columns on the right designate the types of the fields. Reference fields are marked with an arrow (►).
Appendix B: Proof-of-concept implementation

This thesis is accompanied by a proof-of-concept implementation of the event-based model in the form of a simple fixed asset accounting software. Upon the completion of its development, the application is going to be capable of performing parallel accounting for a limited set of business events in conformance with both the IFRS and the Hungarian accounting regulations. The primary goal of the implementation is to serve as a tool for the testing and further development of the event-based model; as such, it completely lacks a usable presentation layer and any other convenience features. Therefore, it is not expected to be suitable for practical accounting purposes.

The development notes, the latest available version of the source code, and instructions for the installation of the application are available at the following address:

http://github.com/solymosi/xas
References


Nyilatkozat saját munkáról

Név: Solymosi Máté János
Email cím: mate@solymo.si
Neptun kód: ET8I2Q

A szakdolgozat címe magyarul:
Eseményorientált adatmodell bővíthető számviteli rendszerek fejlesztéséhez

A szakdolgozat címe angolul:
An Event-Based Data Model for Extensible Accounting Information Systems

Szakszeminarium-vezető neve:
Dr. Lakatos László Péter

Én, Solymosi Máté János, teljes felelősségem tudatában kijelentem, hogy a jelen szakdolgozatban szereplő minden szövegrész, ábra és táblázat – az előírt szabályoknak megfelelően hivatkozott részek kivételével – eredeti és kizárólag a saját munkám eredménye, más dokumentumra vagy közreműködőre nem támaszkodik.

Záróvizsgát a 2013/14. tanév június-július záróvizsga-időszakában kívánok tenni.


................................................
Solymosi Máté János
Témavezetői nyilatkozat

Alulírott, Dr. Lakatos László Péter konzulens kijelentem, hogy a fent megjelölt hallgató fentiek szerinti szakdolgozata benyújtásra alkalmas és védésre ajánlom.


................................................

Dr. Lakatos László Péter
Nyilatkozat a szakdolgozat nyilvánosságáról

Név: Solymosi Máté János
Szak neve: Pénzügy és Számvitel (alapszak)
Szakirány: Számvitel

Dolgozatom elektronikus változatának – PDF dokumentum, a megtekintés, a mentés és a nyomtatás engedélyezett, a szerkesztés nem – nyilvánosságáról az alábbi lehetőségek közül kiválasztott hozzáférési szabályzat szerint rendelkezem.

Teljes nyilvánossággal

A könyvtári honlapon keresztül elérhető a Szakdolgozatok/TDK adatbázisban (http://szd.lib.uni-corvinus.hu), a világháló bármely pontjáról hozzáférhető, a fentebb jellemzett PDF dokumentum formájában.


................................................
Solymosi Máté János