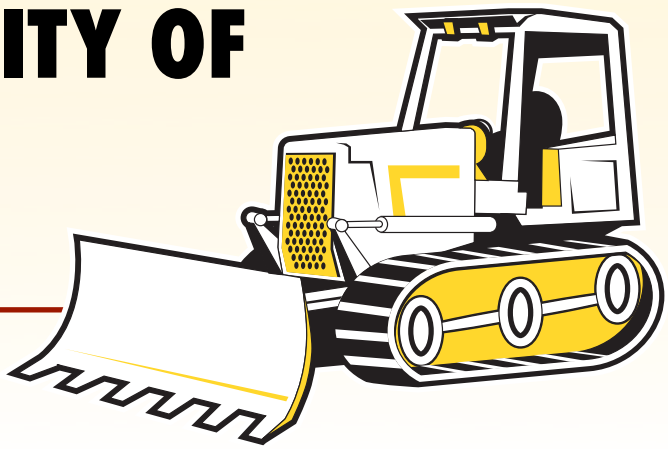
APPLICABILITY OF SOFTWARE FACTORIES



Master's Thesis Computer Science (Management & Technology)
Radboud University Nijmegen
May 2007

Name: Vincent Driessen
Thesis number: **566**

Supervisors:

- Prof.dr.ir. M.J. Plasmeijer *(Computer Science)*
- Drs.ing. P.M. Vos *(Management & Technology)*
- Drs. T.J. Zeeman *(Sogyo Information Engineering)*
- Dr. P.W.M. Koopman *(Computer Science, reader)*

Document typeset with LaTeX. Thesis version was last compiled on June 4, 2007.

Abstract

Developing software is getting increasingly more important, but complex at the same time. Challenges arise for businesses to deal with this. Both science and IT companies are searching for ways of making the production process of software more efficient. Recent investments have been made by Microsoft regarding the software factories movement. This young movement is concerned with the automation of the software production process by enabling generation of software.

This thesis describes an attempt to evaluate the practical applicability of software factories within the software development department of Sogyo, a Dutch IT company. Specifically, the research identifies two major business challenges that Sogyo faces—the ability to deal with requirements that change during development and enabling structural reuse of software that has been created. This research aims at the assessment of the applicability of software factories for tackling those challenges by subsequently:

- Considering a typical application, subject to both business problems and developed by Sogyo. First, this clarifies the business challenges by exemplification. Next, its analysis forms the basis for a re-implementation. Finally, it lays the foundation for a qualitative comparison between the methods;
- Creating a theoretical comparison framework to facilitate the comparison of the application implementations;
- Applying this framework to compare the traditional development of software to the software factory approach.

From the comparison, we establish that software factories are very well suited to tackle the structural reusability challenge of Sogyo, but they unfortunately contribute very negatively to the flexibility in dealing with changing requirements. Sogyo is thus advised to take on a sceptical attitude towards the current possibilities of software factories.

The main contributions of this research are (1) the practical demonstration of how software factories may be put to use, relating concepts to theory, (2) the construction of the comparison framework for the assessment of software products—not only this specific framework, but it may serve as a blueprint for the creation of such frameworks based on other business challenges, and (3) the recommendations for Sogyo.

Keywords: software factories, automation, domain-specific languages, software development, CRUD

Contents

List of Tables

List of Figures

Listings

1 Introduction

This chapter's purpose is twofold. Firstly, it is introductory. The context of this research is described in order to understand the business setting and the software development process at Sogyo. Secondly, we describe the research itself by showing the research objective, the specific goals and the approach to these goals. Furthermore, the motivation for this research and its relevance is pointed out.

We will start with a brief introduction to the software development department of the company Sogyo. We will simultaneously take the opportunity to define what we mean by software development.

Next, we will point out two major business challenges that Sogyo faces and will analyse the possible factors that cause these challenges. We will then see how the need for research arose and how this research will contribute to that. In particular, we will point out which of the causes this research will try to tackle, and we will propose that software factories might assist in the overcoming of these challenges. The latter forms the topic for the remainder of this research.

1.1 Sogyo

Sogyo[†] is a young, medium-sized software engineering company in The Netherlands, situated in the city of De Bilt. Its core business is software application and web development—mostly for customers in the financial sector. Sogyo invests heavily in developing applications for the Microsoft .NET framework. In its business software development process, Sogyo uses RAD tools (like Visual Studio .NET) that aid their development speed at less effort.

Since its foundation in 1995, Sogyo has developed its organisational structure along three pillars. The first is *Sogyo Academy*, which focuses on education and coaching. Next, *Sogyo Professionals* is responsible for the IT Professionals Program (secondment) and consultancy. Finally, *Sogyo Development* is the in-house software engineering department. The hierarchical organisation of Sogyo is depicted in Figure 1.1. The specific context in which this research project will take place, is the department Sogyo Development. However, the research will

[†]Sogyo is named after a 17th century Japanese monk that made the most delicious tea, because he always added a bit of dedication to the tea water.

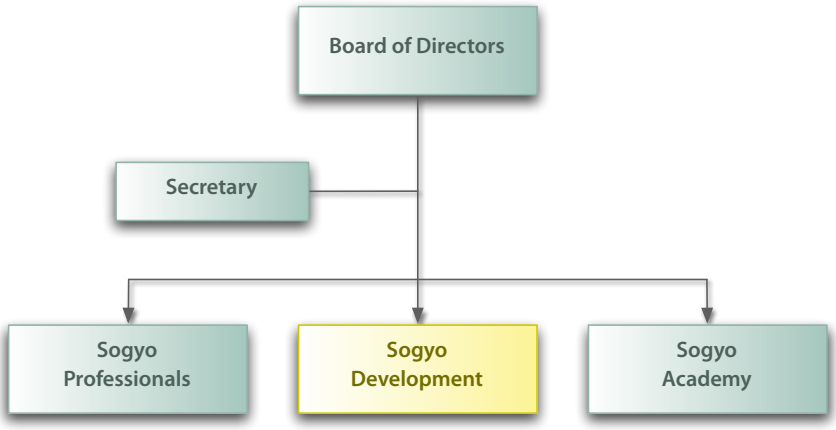


Figure 1.1: The Sogyo organisation hierarchy.

be relevant for Sogyo in general, because software factories play an upfront role in Sogyo's vision of future software development.

Software Development

An important notion that is used during this whole research, is the notion of *software development*. A lot of literature has been devoted to the concept of software development [57, 63, 73] although none of them provides a true comprehensive overview of the field. As a consequence, many definitions of the concept hang around. The definition that will be used throughout the rest of this research is the following.

Definition 1. *Software development*—sometimes referred to as *application development*, or *software engineering*—is the systematic, disciplined, quantifiable, integral process of analysis, design, construction, deployment and maintenance involved with respect to the production of (large) software applications.

Software development in most theories consists of a phased model in which at least analysis, design, construction, deployment and maintenance can be recognised. This research's focus is the design and implementation phases a typical development process at Sogyo Development. Other phases might be mentioned briefly, but will not be considered in detail.

1.2 Problem Analysis

One of the foremost trouble with the software engineering process is the dynamic and baroque environment in which it takes place. At Sogyo, this becomes most visible from the following two main challenges software engineers face. These will be discussed and elaborated on in the following sections.

1.2.1 Dealing with Changing Requirements

Dealing with *requirement shifts* by customers forms a problem that occurs at almost any project. It is a well-known problem in the world of software development and even the most

experienced software developers are not always able to cope with this. Customers, and in particular technically unaware customers, rarely have a clue about the choices underlying a technical solution. Not seldom it occurs that during a project, even although the rationale behind technical design decisions is fully consistent with the requirement analysis and reasoning of customers, the requirements change—sometimes even last-minute. This might be both due to a changing customer's wish, or due to new insights that the software designer gained during the course of the project, for a deep understanding of the problem is never available *a priori*.

Requirement shifts may imply that important pieces of the software suddenly become useless, while small details turn out to be more important than initially thought about. What this means is that developers are required to deal with these new requirements in a flexible manner. Especially when the requirement changes require a full redraw of major components of the existing design, heavy tension is put on the developers. Of course, a good software architect will try to shape his design as elegantly as possible to increase the robustness of the design, but sometimes changes will come from an unexpected angle nonetheless. The development of these designs and implementations therefore puts a burden on the necessary dose of the developer's mental effort.

Perhaps even more important, project duration will develop a tendency to delay. Besides that, especially since customers are typically unaware of the extent of the change in their requirements, let alone the impact of them on the software's design, they will not understand why the project delays, nor why such changes can not be easily dealt with. The consequence of this is that customers might judge the project's outcome to be rather unsatisfactory.

1.2.2 Enabling Reuse

Reinvention of the wheel poses a problem on overall efficiency of software development. A large amount of the produced software shares more or less the same functionality, only applied differently in each case. Engineers realise they are reinventing the wheel over and over again, but at the same time they find themselves incapable of building an infrastructure that would allow for effective reuse of software solutions in order to increase long term business efficiency.

An illustrative example of this problem is the fact that during the last years, several CRUD-structured (create, read, update, destroy) applications have been developed from scratch or with a lot of duplication. The same thing goes for tracking applications, of which, during the last years, three different implementations have been built from scratch at Sogyo. Sogyo has been actively developing libraries that deal with this overlapping functionality. Although these libraries form a practical solution to deal with these reuse problems, a structural solution for it is desirable.

1.3 Causes

The causes to these symptoms form a complex web of related factors. Discovering these causes may require a whole new research subject, which obviously is not our intention. The following potential causes are likely to be the most important ones—at least according to

the author and agreed upon by Sogyo management and personnel and by the research supervisors. Despite a missing scientific justification, the causes as we have laid them out here are sufficiently provided with an argumentation, showing the rationale behind the choice. Nonetheless, they remain hypothesis. It is left to follow-up research projects to assess their validity.

1.3.1 Lack of Anticipation Skills

A vital skill for software designers is the ability to predict the future, at least in terms of risks. This applies to both challenges simultaneously:

- When a designer is unable to anticipate on what customer's next step is likely to be, the design will not be flexible enough to deal with these environment changes to come;
- Lack of anticipation in the long run might cause the second challenge. If a designer is busy making the design of the first customer, he or she might already consider setting up the project's core in a more general fashion in anticipation of a system that might be reused for future similar customers.

1.3.2 Lack of Architectural Skills

Apart from the fact whether or not the software designer is able to predict the future, he must be skilled enough to deal with these anticipations. This means that if the creative or technical skills lack to come up with an elegant, orthogonal design, both challenges will boil up:

- A good design is one that, although satisfying the system's requirements, also treats these requirements as very unstable. A designer must therefore follow a modest and defensive strategy in order to let the design be flexible enough to be quickly adapted to these environmental changes. Risks lures when a designer is too much focused on following the given specs too tightly, not being able to respond to changes;
- Another consequence of an architect following specifications too tightly is that he or she will not be able to separate the prominent from the side issues. Omitting this important architectural task will make reuse virtually impossible.

1.3.3 Communication Problems

Trouble in human communication poses threats on the first problem mostly, but not significantly to the second.

- As the problem description made clear already, a vast problem in the whole requirement analysis phase is the description of the problems in a way that both the customer agrees with it and the software designer may distill a software design in order to compose a solution. Often, even when both parties initially agree on these requirements, they remain to have a different look on the matter. Most likely, software designers will take on a technical view, while customers reason from their own domain logic and take no interest in the technology and solution strategy used.

Next to that, keeping the customer informed with what is happening and how their business challenge will be tackled is a process which requires communication effort. When this communication is disturbed or not present at all, it will lead to the customer having no clue about the technical decisions made and, as a consequence, this poses threats at the acceptance and satisfaction.

- Communication problems do not contribute to the existence of the structural reuse problem. Although bad communication might imply that the core concepts of a software solution are not exposed well enough, complexity is more of a villain than communication issues are.

1.3.4 Programming Paradigm Limitations

The choice of the programming paradigm (the underlying scientific discipline) determines the view that a programmer has over the software he or she creates. The paradigm influences, or even dictates, the way particular problems are solved.

A good software designer (at least theoretically) decides on the appropriate programming paradigm for each individual problem he or she intends to solve. In practice, however, the most influential factors on the paradigm decision are familiarity, reliability, predictability and finance. In fact, often businesses invest in only one development environment and develop all software using that tool, instead of considering alternatives. As a consequence, they are stuck with the accompanying paradigm.

This is also visible at Sogyo, where investments are made in Microsoft's .NET tools and Java. All projects are simply developed using these tools—which means being stuck with object oriented design, without alternative. As an important effect, their customer's problems might be tackled in a suboptimal fashion.

This has implications for both business challenges:

- Because each paradigm individually dictates what *is* and what *is not* allowed, this influences the relation between the domain elements from the problem domain and their representations in the software model. The more natural a representation is able to match the problem domain elements, the more flexible a software model will become when it comes to matching new requirements.
- The more natural the representation of a given domain element, the more effective such a representation (a software object) might be reused. This is due to the characteristics of natural representations, since they describe the elements at their core and model details as details.

1.3.5 Software Engineering Methodology Limitations

Besides the technical programming paradigm to use, there must also be decided on the software engineering methodology. The methodology can be seen as a paradigmatic way of furnishing the software engineering process.

Scientific methodologies are basically concerned with process organisation to support the creation of software models that technically reflect reality as much as possible. The waterfall methodology is the most famous and notorious trivial interpretation of a methodology.

Iterative development is its natural successor, generally accepted as the traditional way of developing software.

Critics of the traditional waterfall and iterative development methods argue that these methodologies are too much based on an idealised situation and should instead take a more practical focus. A typical manifestation of these critics can be found in the field of so called *agile development*, in which the *dynamic systems development method* (DSDM) [11, 57] and *extreme programming* (XP) [3] are the best-known methodologies. DSDM explicitly takes time and resource limitations into account, arguing that absolute quality is not achievable in a commercial setting. XP is primarily focused on the result of the process, not as much on the process itself.

As with the programming paradigm factor, the choice should theoretically be funded on the most appropriate approach to tackle the problem, but the methodology decision is often greatly influenced by practical factors like familiarity and predictability as well. Not too surprisingly, the choice of methodology and programming paradigms are always made simultaneously and fit one another.

As the choice of methodology follows the choice of the programming paradigm and *vice versa*, both business challenges will be influenced by this cause:

- ▷ First and foremost, the choice of software engineering methodology influences the flexibility of the software's design. Differences in methodology rest in the way that the software engineering process is looked against after all. When a software project is less restricted in terms of available resources or deadline pressure, there will be more time to analyse the problem and nuance the software design. In particular, this implies that the flexibility of the software's design can get extra attention, which is a must in order to deal with shifts in requirements.

- ▷ For the same reasons, the reusability aspect might become troublesome. Refactoring parts of software requires explicit effort, since a structural generic solution is much harder to achieve than an *ad hoc* one. Therefore, how a methodology values the importance of a software's design will greatly influence the opportunities for reuse that will sprout afterwards.

1.3.6 Development Environment Limitations

Modern *integrated development environments* (IDE), like Microsoft's Visual Studio or Eclipse, provide a full spectrum of tools for the software designers and engineers, coupling together more small, focused, primitive tooling that are specialised at doing one task (e.g. type checkers, parsers, compilers, linkers and the like) and extend that with intelligent code writing assistants [24, 53, 56]. This eliminates the necessity of learning technical details that stand in the way of the development process. Although generally modern IDE's indeed support a more efficient production of software (especially the *rapid application development* (RAD) tools) and there also is a downside to too much of that support.

The first is that these IDE's pose limitations on architectural creativity. Increasingly, tooling becomes the main reference point of the programmer, which shows from the fact that functionality that is directly visible to the programmer is more often used than functionality

that is hidden at a deeper level. The explicit availability of that functionality might therefore falsely imply that it is the *only* functionality.

Also, such IDE's encourage software development in an experimental fashion—just clicking together some software components, write some code and see what happens might increasingly guide the development process. Structurally, this is the wrong approach. The path walked easiest is not the best path in most circumstances. Increasing luxury apparently comes at the risk of increased laziness.

This affects both of Sogyo's business challenges to a degree:

- While technically an IDE eases the writing of code and the creation of *graphical user interfaces* (GUI), they do not contribute to the development of better software models. Attention, too often, is not given to the core of the software: the model.

  The risk of this is that, when a requirement shift occurs, the designers and developers will try to adapt the code to satisfy the new requirements. While doing that, they get assistance of the IDE, which perceptually contributes largely to the speed of that process (i.e. the requirement shift). However, typically the programmer will not realise that the cause of the requirement shift might be the lack of attention to the model, caused by the IDE in the first place.

- The RAD philosophy might threaten a good reuse model. Although RAD tools use reusable components extensively, this is not necessarily a guarantee for developing reusable software. As we already saw, RAD tools automate code writing and interface design in order to guide the programming process and to make this as much of a trivial task as possible. And although the intentions are good—by automating programming, there can be more attention to the modeling—this does not work out like that.

  Since there is no support for the modeling tasks, in combination with the extreme guidance during the programming process, the developers will become more and more dependant on this automation. As a consequence, pure modeling will become less important, since the automation of the programming process will be able to deal with problems that arise from that in an *ad hoc* fashion.

  This approach leads to a process of *unintended* favouring of patching over curing, which is fatal when the creation of reusable components is important.

1.4 Relevance to Research

In order for Sogyo to tackle the business challenges it encounters, at least these causes must be cured. Table 1.1 shows an overview of the causes discussed. The human causes are pretty influential to the problems. In order to cure them, for example education and experience gaining will be required. And although this cure is anything but trivial, it is outside the scope of this technical thesis. From a computer science perspective, it would be more interesting to look at the technical causes, and that is exactly what will be done in this thesis.

A very important note that should be made by now is that, since we explicitly do not try to tackle these human causes, Sogyo's main challenges will probably remain, unless all of the causes are targeted. It is therefore not the intent of this thesis to solve the main challenges, but to contribute to a part of the solution. By approaching the technical causes, however, we

Influential factors	Requirement Shifts	Enabling Reuse	Selected
Human			
Lack of anticipation skills	++	+	–
Lack of architectural skills	+	++	–
Communication	+	–	–
Technical			
Programming paradigm	++	++	✓
Software engineering methodology	++	+	✓
IDE limitations	+	++	✓

Table 1.1: A simplified, schematic overview of the selected causes.

might sideways contribute to the cure of the human causes, too. For example, if a technical solution is able to create a qualitative infrastructure for software developers, this might lower the imposed human creativity and experience levels. As a consequence, the level of necessary human training will lower, accordingly.

1.5 Software Factories

A specific software development methodology that claims that it can be used to effectively weapon oneself against the selected causes as Sogyo encounters them is *software factories*. It is a movement lead by developers at Microsoft. The key concept of a software factory is that it can generate a whole product line of similar domain-specific applications, based on a dedicated software model.

To facilitate that, the software factories philosophy suggests the use of the *model-driven design* (MDD) technique [20, 22, 30]. One of the goals that MDD aims at is constructing a high-level domain model, which poses a visual language for modeling the target application and generating it to a large extent from the model information.

Ideally, software specifications can be given in a domain-specific language, from which code and abstract services (think of themes like security, persistency, etc.) can be automatically generated. Of course, this is the vision of the advocates of model-driven design.

With the release of the latest Visual Studio 2005 SDK's, Microsoft has shipped *DSL Tools*[†] as part of the package. DSL Tools is a Visual Studio.NET 2005 plug in that aims at the easy creation of visual domain-specific languages that can be used to model software. Microsoft claims that designing a separate, domain-specific language, and use that to generate software would put back the focus to the core of the process: the model.

At Sogyo, interest in DSL finds its roots in work floor enthusiasts that share Microsoft's

[†] http://msdn.microsoft.com/vstudio/DSLTools/—visited December 21, 2006

vision. Yet budget limitations impede R&D on the very subject. If Microsoft's claim is true, this might contribute to the cure of the causes, and by doing so, form a partial solution to Sogyo's business challenges.

1.6 Objective

Since expectations at Sogyo are that software factories and the generation of software might be an improvement to the development process and that it might help in guarding the process against the two main business challenges, the general research objective will be to validate that.

The goal of this research can then be stated as follows:

> **Goal.** To determine to what extent software factories may contribute in practice to Sogyo's current development process, specifically regarding the ability to deal with changing requirements and to enable structural reuse.

How this research goal will be achieved is elaborated on in Section 1.7.

Considering that Sogyo Development currently has no running projects involving software factories, nor has had any in the past, this research might sideways touch the surface of pointing out opportunities and risks that might be expected when engaging development using software factories.

To achieve the research goal, the following questions need to be answered. Each main question is answered by answering the corresponding sub questions.

1. **To what extent do software factories change the way applications are developed?**
   - ▷ What activities should be taken in the process of creating software with software factories?
   - ▷ What contextual requirements can be identified when using software factories?
2. **To what extent will software factories have a positive effect on the ability to deal with changing requirements, compared to Sogyo's current way of developing?**
   - ▷ What attributes are influential to the effects on dealing with changing requirements?
   - ▷ To what extent do software factories influence those attributes positively or negatively?
3. **To what extent will software factories have a positive effect on the ability to enable structural reuse, compared to Sogyo's current way of developing?**
   - ▷ What attributes are influential in determining the effects on enabling structural reuse?
   - ▷ To what extent do software factories influence those attributes either positively or negatively?

1.7 Research Approach

We will propose a research model, forming the followed strategy to answer the research questions posed in the previous section. This is depicted in Figure 1.2.

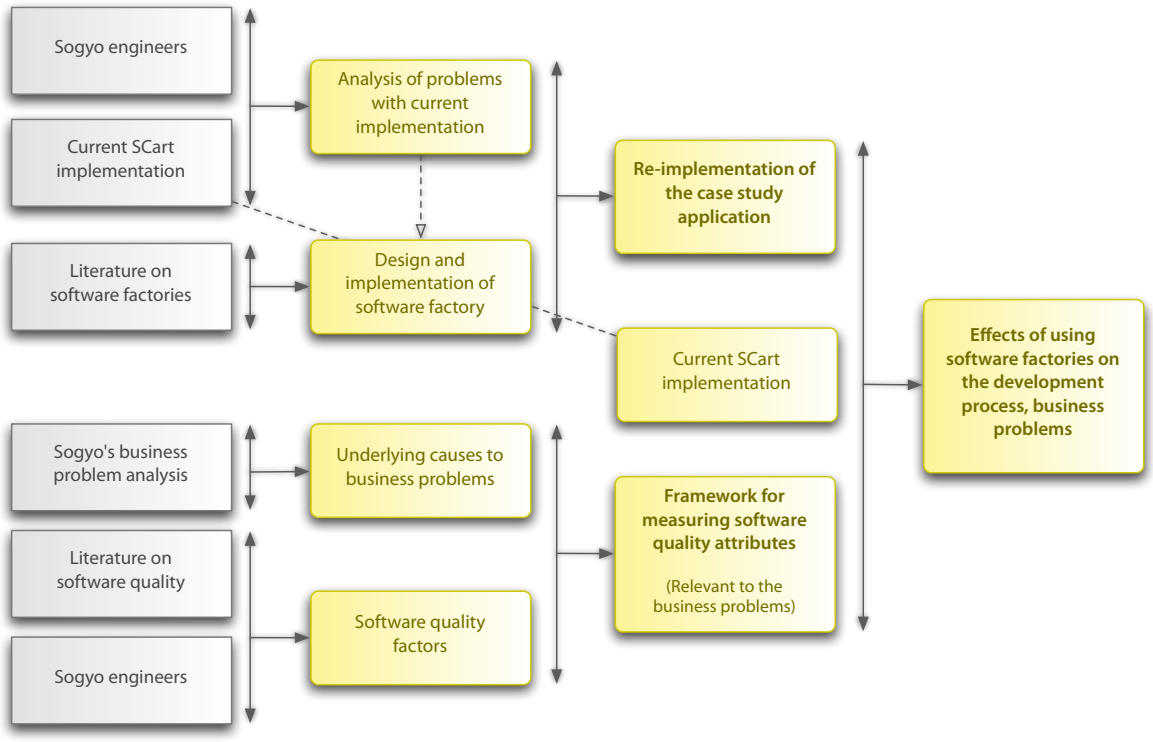


Figure 1.2: The research framework.

To visualise the problems that occur or have occured in the development process, we will do a case study of SCart, a case which is affected by both business challenges. We will analyse the case, pose an example software factory and re-implement the case using this software factory, in order to show that software factories can indeed be used. This is not a qualitative judgement, but rather an argument that software factories can be used to implement the application.

Next, we will develop a framework for judging software quality, by selecting software quality attributes from a thorough study of software engineering literature. The selection of relevant attributes will be based on the applicability and relevant of the assumed causes of Sogyo's business challenges. A framework using indicators then is proposed to measure the quality of software attributes affecting the specific business challenges.

Finally, we will compare the ways of developing the application, the original one with the re-implementation. This comparison will both be based on our own experience developing the solution, and based on technical criteria, which are offered by the research framework we have developed.

1.8 Chapter Structure

This research follows a chapter structure that resembles the proclaimed research approach. It is depicted in Figure 1.3 below.

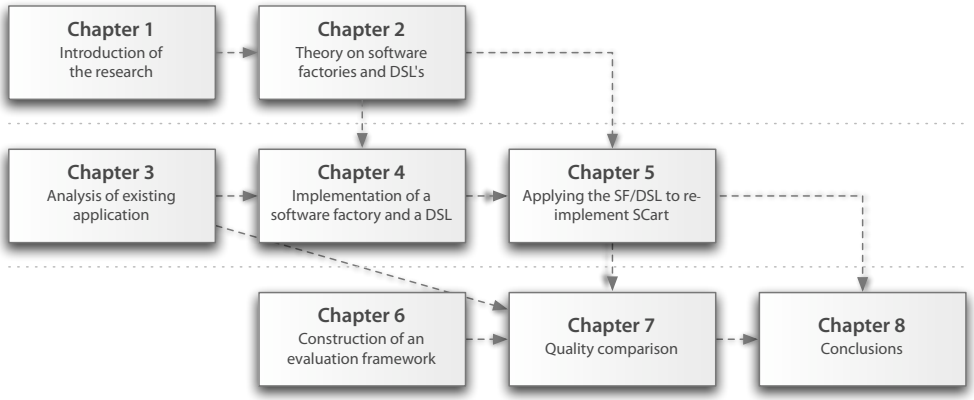


Figure 1.3: The structure of chapters throughout the thesis.

- **Chapter 2** will introduce the concept of what a software factories is, discuss its motivation and explain the rationale behind the philosophy. It will explain the connection between software factories, (visual) modeling and domain-specific languages;
- This research is built up around a case study. **Chapter 3** will introduce the CRUD application SCart, analyse it and illustrate how its development is subject to the business challenges;
- In **Chapter 4** we will propose a software factory solution based on the analysis from Chapter 3, designed to generate CRUD applications with minimal amount of manual work;
- In **Chapter 5**, the software factory is used to actually make a re-implementation of the application. It will also point out specific implementation details and practical implications;
- **Chapter 6** will discuss the factors that define the quality of software. A focus is taken on the criteria that are considered relevant to the business challenges. A framework for objectively comparing the quality is constructed;
- Then, in **Chapter 7** this framework is put to use to make the actual comparison;
- Finally, in **Chapter 8**, conclusions of this research are drawn from the comparison results. The research questions are explicitly answered, recommendations are made and further research subjects are suggested.

1.9 Contributions

The main contribution of this research thus are the following:

- Identification of business challenges, an argumentation of why a new development paradigm is necessary, and the expression of an expectation for a possible solution;

- ▷ An introduction to the concept of software factories, and how these could be used to the concrete challenges of Sogyo;
- ▷ A case-study of an applications with typical business challenges symptoms (SCart), pointing out the problems, including a product-line analysis identifying commonalities and variabilities for CRUD applications;
- ▷ A design and implementation of a software factory (CRUDE) for specifying CRUD applications from their underlying database designs, respecting the product-line analysis;
- ▷ An example re-implementation of the SCart application using CRUDE, thereby showing how software factories can be used in practice to solve the typical problems around CRUD applications;
- ▷ The creation of a reusable comparison framework, based on literature and specifically tailored to the business challenges, for the comparison of the software quality of the original application and the re-implementation;
- ▷ A quality comparison of the original and re-implemented SCart applications, using the comparison framework. Based on that comparison, concluded that software factories indeed are a useful tool for the development of CRUD applications.

Summary

We have introduced the company Sogyo, their development department and have outlined that the main challenges that the developers face consists of

- ▷ the ability to deal with (last-minute) requirement shifts; and
- ▷ the enabling of structural reuse.

We have shown what the causes are that are rooted behind these symptoms and have proposed a methodology that might be used to tackle these problems: software factories. Based on that proposal, we have formulated a sequence of research questions and a strategy as it will be followed in the remainder of this research.

In the following chapter, we will introduce the theoretical concept behind software factories, its rationale and its core foundation: domain-specific languages.

2 Software Factories

To determine whether software factories could be a solution to Sogyo's business challenges, as laid out in the previous chapter, we should first explain what a software factory is, where it origins from, and how it contributes to exactly what problem. The purpose of this chapter is to provide the theoretical foundation for the research. A vocabulary is introduced that is used throughout the research, providing a solid base for the understanding of the approach and related technical concepts.

We will throw a light on the challenges that are encountered with traditional software development. This is done by showing that traditional software development is hard because it does not scale up well when project sizes increase. We will show that the cause of that problem is the bridging of the abstraction gap.

Next, we will discuss software factories as another way of approaching software development. The philosophy behind software factories is discussed and we will introduce its core concept, domain-specific languages (DSL).

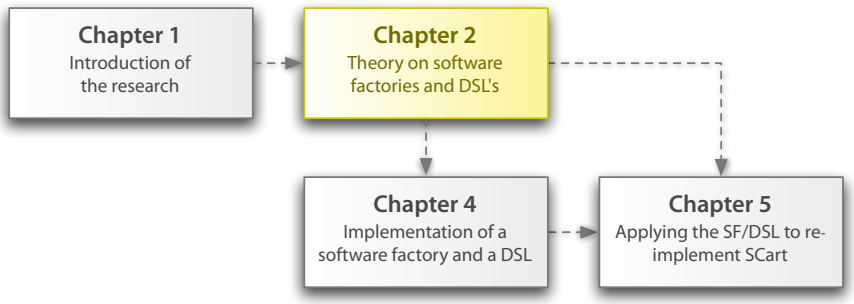


Figure 2.1: Chapter 2 in its context.

Much of this chapter builds on the work of Greenfield and Short [30]. For a more thorough discussion of software factories, their work is much richer and we encourage the reader to consult that work.

2.1 Traditional Software Development

Software development is notorious for its reputation to be carried out inefficiently, with crossing deadlines, cost limits and its tendency to fail to achieve, let alone exceed, expectations. It has been for years and the image is probably here to stay for a few more. A lot of software users do not trust software, yet have become very dependent of it.

Recent studies [13, 31, 48] have pointed out that in 2004, only 30% of all software was valued "successfully executed and delivered". About half of all software projects were "challenged", which by definition of the research group meant that the project balanced at the edge of failure, or that customer satisfaction was low. Typically, these projects exceeded their budgets and deadlines. Finally, a remaining 20% was delivered by far exceeding project limits, or projects being cancelled. In early reports [32], numbers were even more pessimistic. In about 10 years, the industry has been able to increase the success and decrease failure rates by about 10%. But still, half of all software projects are challenged, so optimism should be approached modestly.

In essence, software has a great potential to solve business challenges. But clearly, the software industry has not been able to mature fast enough. What characterises the industry is that individual technology advances, but the structural process of how programming problems are solved has not, or only limited. Any software technology introduced since assembly programming, was put into the market with huge pretensions, but failed to leverage.

Greenfield indicates that stakeholder expectations increase as platform technologies advances. It is the industry's task to keep up with these expectations. Yet, most software nowadays is still developed by hand, from scratch, using labour-intensive methods instead [30]. Looking at the industry's progress over the last few years, the enhancements regarding process efficiency are scarce. Marginal improvements are not enough to catch up with major technology advancements.

This is in line with the theories by people like Kuhn and Christensen [10, 45], who argue that disruptive innovations take place when settled innovations are matured until the point where they outgrow their own technical or physical limits and new innovations are invented to solve the existing problems. Every so many years, new innovations take over the settled market and lift production to a new level, making the old innovations virtually obsolete.

Given that the software industry will only grow in the next decades, the current way of working will simply not suffice to obey the demand. Instead, the industry should take a look at the lessons learnt in other industries that were dominated by craftsmanship once. While craftsmanship works well enough in relative small and immature scenarios, it does not scale up well. More mature businesses should therefore embrace process automation in their development where possible [30].

While the question as to *what* is to be done to lift the maturity level of software development remains unclear, so much is clear that some dramatic changes have to take place in the current process. Process automation once was a key concept in the industrialisation of factories, where production efficiency was achieved by machines and product-lines and the amount of manual work decreased incredibly.

All of this is easier said than done. However, historical patterns from the industrialisation

era might be more similar to present day software development than might be expected *a priori*. The industry will probably not be able to understand how the industrialisation should be achieved, but it will be recognisable once it is there. Now that society is familiar with the concept of automated factories, looking back we know that adding more labour capacity would not have solved the efficiency problems back then. Instead, it is the *tooling* that should be improved to make people more productive.

Having established that we will only fully understand how we should shape software development only once we have done so already, this might seem a useless undertaking. Nonetheless, searching for this holy grail will positively change the way we think about how software should be constructed.

2.2 Simulations

Software creates solutions to problems[†]. There are two important notions involved in that process that need to be strictly separated to keep the discussion clear: the *problem domain* and the *solution domain*. As Lenz and Wienands [48] argue, the problem domain should be described in business language, whereas the solution domain should be described in technical terms in order to solve the problems. A *domain* is a specific subject area, or an object family. It is typical to the viewpoint of an actor or field of focus and therefore has a specific problem set.

Definition 2. A *problem domain* is the domain related to a specific problem that an actor has. Here, "problem" refers to typically a business or a technical problem, experienced by an actor, for which a solution is wishful.

Definition 3. A *solution domain* is the domain related to a specific, technical solution that might be able to solve (part of the) problems in one or more given problem domains.

Lenz and Wienands provide a simple but illustrative example. Suppose a person wants to travel from point A to point B within a city. The key problem for this person is how to get there. This problem belongs to the *problem domain*. In order to solve his problem, this person might use a street map. The knowledge that this map provides is an oversimplification of reality, which makes the problem easy to solve. The route from A to B can be distilled from the map, and mapping that information back to reality leads to the way to walk. That is the *solution domain*.

An important thing to realise is that both the problem and the solution domain exist in reality. Software usually represents concepts from reality, but computational objects have no meaning in reality, until they are interpreted. This is similar to the street map.

Their relationship therefore might be defined in terms of *domain* and *mappings* between them (see Figure 2.2 on the next page). Resolving these mappings requires intellectual efforts, but once these mappings are known, they are relatively easy to implement in software [30].

[†]Although some would argue that software is the cause to many problems.

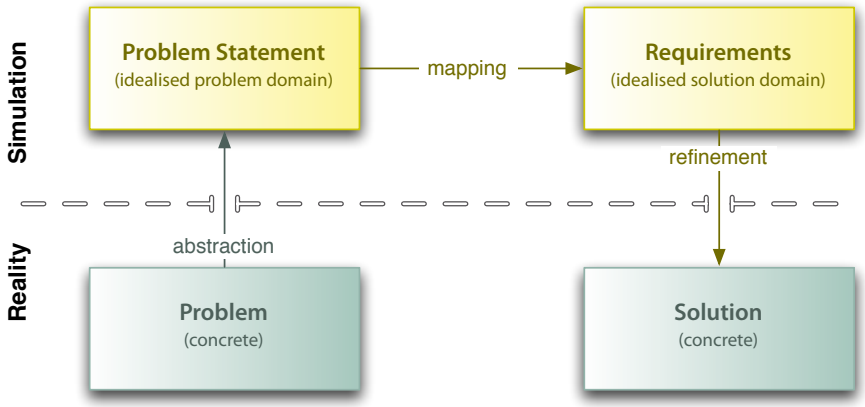


Figure 2.2: The process of coming to a solution starting from the problem.

2.3 Abstraction in Software Development

All key innovations in software development can be related to the act of increasing the level of abstraction. Abstraction is the process of describing concepts at a more general level by removing selective information from a description with the intent to focus on the remainder. The street map from the example above is a typical abstraction.

The inverse of abstraction is refinement. Refinement concerns the addition of details to an abstract concept, with the intent to create a concrete implementation, for example to create executables. Relating this to the street map example, a refinement would be the interpretation of the route on the map to the actual way to walk in reality.

Abstractions capture and encapsulate knowledge, which makes using the abstractions easy because details are not involved. Making abstractions is only useful if they can be used to solve more than one problem—they are always used to solve *a family* of problems.

The fundamental concept behind using abstractions and refinements is the intensity and localisation of knowledge. The more knowledge an abstraction contains, the less the programmer using that abstraction has to provide. However, the more knowledge is contained in the abstraction, the narrower the scope of an abstraction becomes.

Figure 2.3 on the facing page shows some examples of abstractions, indicating that the narrower the scope, the more value an abstraction offers and vice versa. Generality indicates the *scope* of an abstraction, while specificity is a measure for its *value*. General abstractions encapsulates much knowledge, while specific abstractions have a reduced scope, but have more control over it. The value of an abstraction increases with its control.

2.3.1 The Abstraction Gap

There is a huge difference between the problem domain and the actual code of an application [40]. Abstractions are used to manage the complexity involved with closing the gap between the requirements (specification) of an application and the final implementation using software thereof. Linking this back to Figure 2.2, the challenging part concerning software factories is the refinement step from requirements to the concrete solution.

This abstraction gap must be bridged, which is done by simultaneous top-down refinement

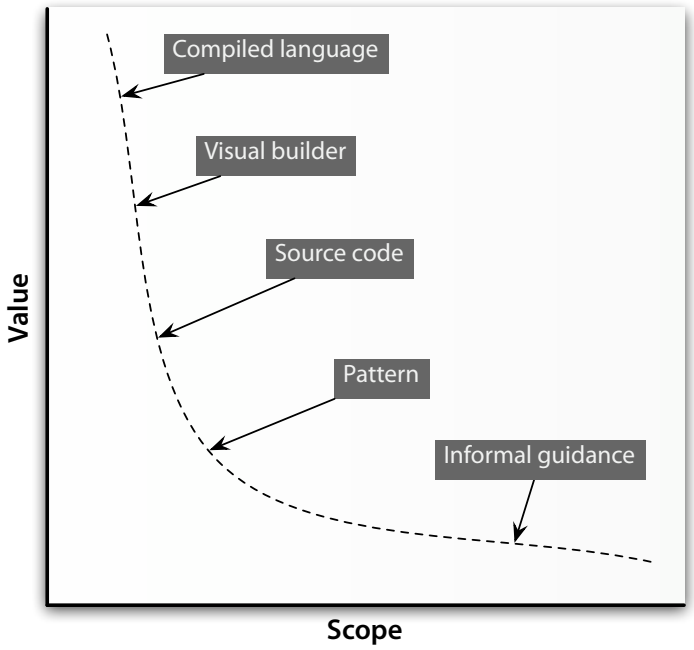


Figure 2.3: Scope versus value for abstractions—adapted from [30].

and bottom-up abstractions. Much of this work is still carried out by humans, using labour-intensive efforts. Typical efforts include making design decisions and writing source code that captures them. After that, source code is compiled and the resulting executable is actually the only real implementation of the requirements. In other words, although the source code often is considered the implementation, it actually is a specification used as the input for a compiler.

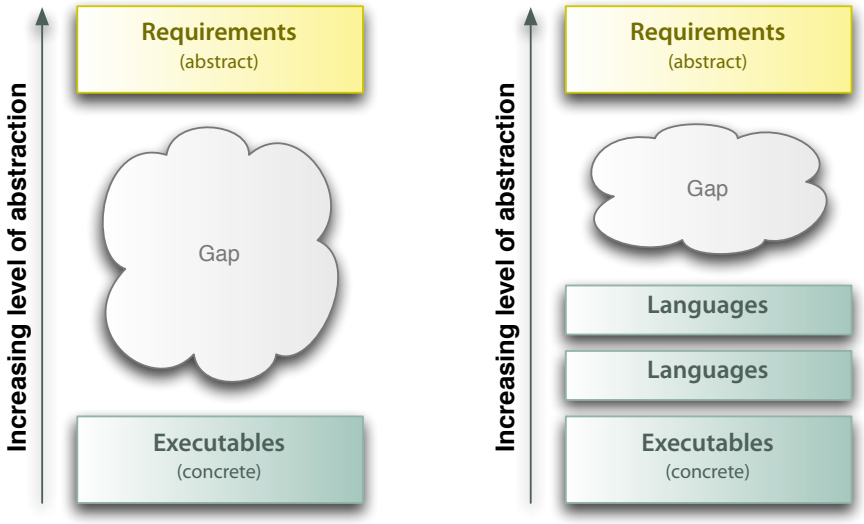


Figure 2.4: The abstraction gap on the left is large. Using languages (i.e. tools) can decrease the gap that must otherwise be closed by human labour.

From that, we may conclude that closing the gap is partly done by human labour and partly by tools (i.e. languages, compilers, etc.). This is shown in Figure 2.4 above. Exactly this observation drives the software factory philosophy, which argues that more aspects of the

development process can be automated. For example, if the right tooling is available, one might be able to generate (part of) the source code directly from the model. These kind of automation opportunities tend to show up in the less creative parts of the development process. Tools are simply better at performing tedious, straightforward, clearly defined jobs than humans are. On the contrary, programmers are capable of abstract reasoning while tools are not, because of the creative nature of that process.

As [30] emphasises, a lot of programmers are not even aware of the fact that they are writing specifications for compilers. Instead, they believe that they are writing the actual software. If tooling would be capable of generating source code from specific models, programmers would be able to focus on the intellectual modeling job, instead of performing the tedious job of writing code, much like how compilers have eliminated the necessity of writing machine instructions.

2.3.2 Manifestations of Abstractions

Greenfield and Short argue that abstractions can be manifested in two forms, either through platforms or through formal languages.

Abstractions through the platform work by making the platform more abstract. Using the .NET Framework, there are dedicated classes for controlling web services easily—something that had to be done manually on the Win32 platform. It is important to realise that these design decisions are bound at platform development time.

Next, languages can hold abstractions too. Formal abstractions are defined by language constructs. Through these language constructs the actual implementation details (compiled or interpreted) are hidden. Informal abstractions might be manifested in patterns, which are defined in documentation. Patterns are based on experience. Because of their informal nature, they are less rigid and susceptible for alternative implementations, because their implementations are not prescribed.

Where abstractions reside tells something about their maturity. Typically, abstractions will start out as isolated practices by individual programmers. Once this common practice becomes clear, they are described in patterns informally. Next, they might be incorporated in platforms formally and finally find their way into languages. A typical example of that is event handling, a mechanism that was used by programmers pretty unstructured[†] and first described as the OBSERVER pattern by the Gang of Four [25]. After that, events where incorporated in platforms like the Java VM through listeners. Only more recently, events where structurally promoted to native language constructs like for example in C# through delegates.

2.4 Fundamentals

Intents to formalise software factories have not been very successful, because of the involved complexity in describing development processes formally. To come to the level of systematic reuse instead of *ad hoc* reuse, at least the technical aspects of such processes should be

[†]Not even realising the concept of “event handling”.

formalised. According to [30], the three key concepts that each software factory should contain are:

1. A software factory **schema**, which is a definition of assets (inputs) and production artefacts (outputs), including a specification of how to build the production artefacts from assets to build those;

2. A software factory **template**, configuring extensible tools, processes and content to form a production facility for the product family. Whereas the schema provides a description on paper, the template is used to actually implement the schema using tools such as languages, patterns and frameworks. The template therefore is the collection of all assets for the factory;

3. An extensible **development environment** provides the tooling for the cook to handle the preparations easily and automate menial tasks.

An analogy to a software factory is a kitchen[†]:

1. The software factory **schema** contains all the recipes of all the meals in a restaurant—it describes which ingredients are required for which meals and how to prepare them;

2. The software factory **template** is like a bag of groceries that contains all of the ingredients necessary to cook the meals;

3. The extensible **development environment** is like the kitchen where the cooking takes place.

Schematically, such a software factory is shown in Figure 2.5. The development of software using software factories involves two stages. In the first, the software factory itself must be developed. From that results a software factory schema and variable assets, that together are utilised to create specific tooling that can optimally be used in the process of creating individual software products.

This implies that software factories are only useful for the creation of many individual, but similar products. This can be compared to economies of scope, such as car manufacturers that make many similar cars that each are unique, but share a lot of the same parts and design concepts. Once the factory is built for a specific domain, structural reuse may take place in the development of the different instances of software applications. This means that the factory is developed *once*, and software products reuse the facilities that the factory offers (domain models, tooling).

Building software products using software factories has aspects common to the traditional process of creating software, like problem analysis, specification, design, implementation, testing and deployment. The difference is that software factories requires some of these aspects to be performed formally. For example, the design phase will yield a model of the software that is considered an asset, not documentation for human purposes. It is a specific, formal, first-class product in the whole process, used as input for machines later. Generally, the main difference is that software factories describes a framework for applying each step in a structured way.

[†] http://blogs.msdn.com/askburton/articles/232021.aspx

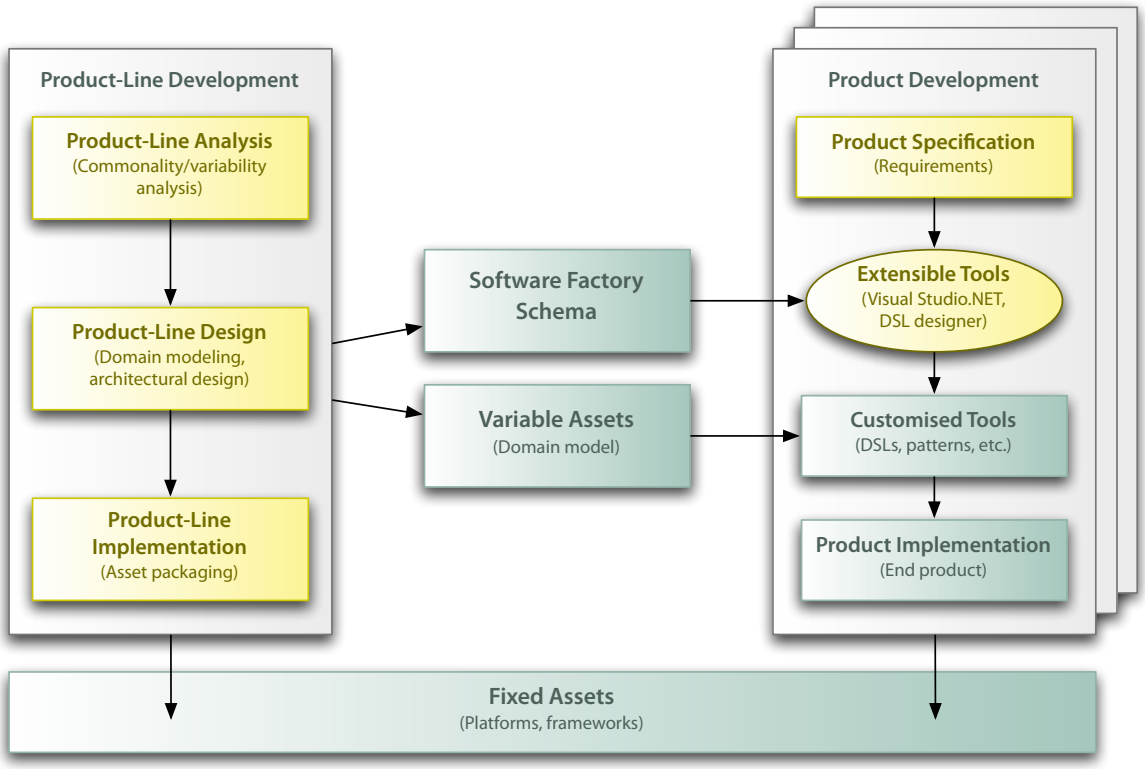


Figure 2.5: The process of building a software factory—adapted from [30] and modified for this research.

2.4.1 Towards Product-Line Thinking

By now, the message should be clear that software factories are aimed at the facilitation of structured reuse of software. A major consequence of this approach is that the mindset of developers should make a shift. They should not be thinking of individual software products anymore, but move towards the idea of developing product-lines—families of similar but individual products.

This implies, among others, that the whole analysis and design phases should reflect exactly that shift, too. In the late nineties, Coplien, Hoffman and Weiss from Bell Labs, made a case for performing structured analysis and created a method called the SCV-analysis, for scope, commonality and variablility analysis [14, 75]. This might be regarded one of the first attempts to formalise the concepts of product-line thinking. The SCV-analysis consists of five steps. From [14]:

1. Establish the scope: the collection of objects under consideration;
2. Identify commonalities and variabilities;
3. Bound the variabilities by placing specific limits—such as maximum values—on each variability;
4. Exploit the commonalities;
5. Accommodate the variabilities.

Phases 1 and 2 are the key phases and are the hardest ones to carry out. Coplien et al. provide a simple yet interesting example to grasp the kind of complexity involved with commonality analysis in general. Imagine that the following two equations express two program family specifications:

$$(x - y)^2 \tag{2.1}$$

$$(x^2 + y)(x - y) \tag{2.2}$$

In this scenario, obviously the factor $x - y$ can be identified as the families' commonality, while the remainders $x - y$ for (2.1) and $x^2 + y$ for (2.2) are the variabilities. Using long-lasting principles from software construction, we should implement $x - y$ separately as component X and re-write these implementations as X^2 and $(x^2 + y)X$ respectively.

However, to show the complexity in software, suppose the formulas where somewhat less compact and written as:

$$x(x - 2y) + y^2 \tag{2.3}$$

$$x^3 + (x - x^2 - y)y \tag{2.4}$$

These implementations are algebraically equivalent to the first two, respectively. However, it would be much harder to distil the factor $x - y$ from them. Of course, software is even much harder to refactor and this is exactly the challenge that software engineers face. Finding these commonalities is the key job of the software developer and will be remain a matter of human intellectual effort for a large part.

The previous steps might sound trivial and are nothing new in computer science field. Early work by Dijkstra and Parnas already indicated a strong correlation between design decisions and program families [16, 55] and of course familiar concepts like inheritance derive from the same practice. However, they did not *invent* it. Concepts where formalised and scientist became more aware of what they were actually doing. This is similar to the abstraction maturity discussion on page on page 18. In that light, SCV-analysis is not some *new* method, but provides a framework for performing those steps orderly, which becomes increasingly important when moving to product-line thinking.

2.4.2 Managing Generalisation

Finding the right balance in how far to pursue the extraction of commonalities is a matter of taste and business priority. Software manufacturers in general will not have a clear view of the domain they are developing for from the start. Knowledge is gained by experience. Therefore, they are not able to directly incorporate all future wishes that arise from the market.

What virtually always starts out as a product for a specific customer will be used for the development of software for new customers that demand a similar solution. First, it takes time for the software developer to recognise the repetition pattern. This mostly arises when developers encounter déjà vus when modeling or writing lines of code. The next step is to formalise the concept, which Greenfield and Short identify as *ad hoc* reuse. We will see a specific example of this in Section 3.5, where the SCart application is studied.

Although there is nothing wrong with *ad hoc* reuse, it does not scale up well. As more and more similar software is demanded from the market, businesses will soon not be able

to deliver. Structural reuse is when the process or solution architecture is fundamentally refactored to support the concept. Dikel et al. provide a visualisation of that process. It indicates a typical evolvement of product-line thinking.

Essentially, the risk involved in not being able to scale up the development process to meet the need of the market (either in numbers or in time) will invoke the need of widening the architectural scope of the application. However, the risk in that is that there will be over-generalised. Not only is there massive labour involved in the development of such a wide architecture (many details are required), but to apply this architecture to an actual application will be as much work and will require specialists that are familiar or experienced with the architecture.

Effectively, managing this complexity comes down to knowing when to stop and find a balance in specificity and reusability. Working with software factories means accepting and respecting these insights and keeping them in mind at all times.

2.5 Model-Driven Design

In the process of shifting from one-off thinking to product-line thinking, the first question one should ask oneself is *what* part of the manual processes are possible candidates for automation.

2.5.1 Capturing Intent

The base rationale for the structural incorporation of formal models is the capturing of intent. Intent is what developers intend to create when involved in the development of software. In order for intent to be usable, people will develop mental images of important solution concepts (like CUSTOMER, PROJECT, ORDER and the like), which should be captured into software one or the other way. Source code, however, is not a suitable vehicle to carry intent [30]. Reading source code, the original intent is not clear and cluttered with details that disguise the core concepts. Source code *can* be—and in an ideal situation *is*—an implementation of the intent, but not the other way around. Therefore, other mechanisms are required to capture the intent of software.

Once done with the development of software, the intent is often lost either immediately or incrementally over the passage of time. Some knowledge related to the intent might be conserved informally in requirements or design documentation. But we have seen already that these documents are victim to quick depreciation.

A way to capture the intent as much as possible, is to require the software to be specified in a far more direct and explicit fashion. The key problem to source code is that is blurs the core application logic with implementation details. We need to specify software in the language of the intent, instead of the implementation thereof.

2.5.2 Modeling

Modeling is a special case of creating abstractions. Models are powerful abstractions for several reasons. First of all, like any abstraction, they leave away irrelevant, trivial, or

complex details while putting an emphasis on others. Exactly because of this, models have the potential to reveal intent just by looking at it, while the same thing would be complex from source code alone. Documentation must either be promoted to a source artefact, or replaced by an intuitive model, effectively "replacing" the intent. By source artefact, we mean that models are used to generate (partial) implementations. This is exactly what *model-driven development* (MDD) is about [65, 77][†].

As we saw, software product-lines are defined by their commonalities. We also saw that commonalities must form the base for the software factory. These commonalities reflect the *domain* of the product family. Evans argues that the domain concepts should be encapsulated in a separate layer—the so called *domain layer* [20]. The domain layer contains all of the application's core knowledge concerning the problem domain. The layer may only contain information on the state and behavioural aspects of the domain. The main purpose of such design is the separation of the application's core logic from the supporting logic that is necessary to create a low-level implementation.

The domain does not necessarily have to be a black-box implementation. In essence, objects from the domain model may be directly interacted with and events occurring within the domain can be caught by software components outside the domain layer. However, the domain by itself may strictly never know of objects outside the domain. Each of the services should be connected with the domain through adaptors. Note that these adaptor objects also know of the domain objects and may interact with them. The service connected to the adaptor should ideally not talk to the domain objects instead.

2.6 Domain-Specific Languages

In order to automate the rote work in the development process, the model must be persisted in a format that tools are able to handle. Therefore, the model must be formalised. General-purpose modeling languages, of which UML is the most well-known example, are unsuitable for generation of code, as *computer-aided software engineering* (CASE) tools from the eighties have shown. These model can describe anything generally and therefore nothing specific at the same time.

Greenfield and Short argue that while there are always wrinkles that make each project unique, the vast majority of the work does not change much from project to project [30]. They propose that each project indeed might require Turing-complete, general purpose languages for a small part, but many specific details might as well be implemented using much more abstract languages, even though these do not offer Turing-completeness.

What is important is that automatic refinement is possible from these models. Recall from Section 2.3 that value decreases with the scope of abstractions. The more specific a language is therefore, the more value it carries. Therefore, we need formal domain-specific modeling languages to support automation.

Domain-specific languages have been discussed in literature for quite some time, but few authors have really tried to define the term. The definition that will be adopted for the purpose of this research is the one posed in [72]:

[†] http://www.voelter.de/services/mdsd-tutorial.html—visited February 19th, 2007.

Definition 4. A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Because DSLs are typically relatively *small* languages, in literature they are sometimes referred to as *little languages* [4, 71] or *micro-languages*. Although it is a matter of taste, the name *domain-specific languages* arguably is a better name, because many DSL implementations will be built upon an already existing language, which in most cases is a GPL. Consequently, the DSL inherits all GPL functionality in addition to offering domain-specific power. In these cases, clearly the name *little* is at least misleading.

DSLs come in a vast majority of flavours, which will be briefly discussed here. Primarily, DSLs can be classified based on language properties, for example be it a declarative or an imperative language, or by their representation, be it a textual versus a visual language. A third classification is proposed by Fowler [22, 23], based on implementation details, for example they are created from scratch or implemented inside another language. We will not focus on this language property, since it is not an essential property of the language, but merely an implementation detail.

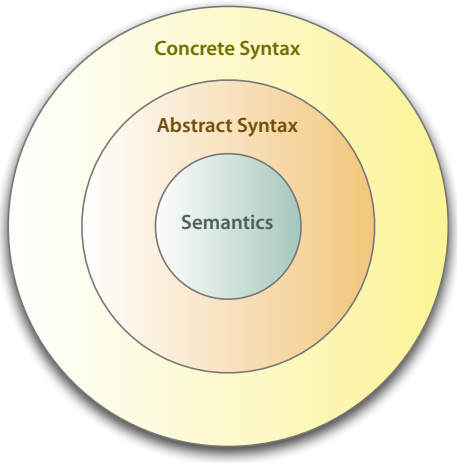


Figure 2.6: The elements of a formal language—adapted from [30].

The rationale behind the categorisation has to do with the variations possible within the elements of which each formal language is built [23, 30, 59, 76]. Consider Figure 2.6. At the inner core reside the language *semantics*, which concerns the meaning of the individual and the composition of language elements, as comparable to the meaning of words as they appear in a dictionary. On top of that, the *abstract syntax* is built, which concerns the language constructs that are available in the language. In English, these would be language constructs like verbs, nouns, syllables, and how they can be combined to form sentences. Lastly, the *concrete syntax* is a concrete manifestation through which the abstract syntax can be represented and communicated. Examples are textual, spoken or drawn.

The language variation space of DSLs is defined by the variations on the syntactic level, combining both abstract and concrete syntax variety. This is displayed in Figure 2.7 on the facing page, with some examples positioned in it.

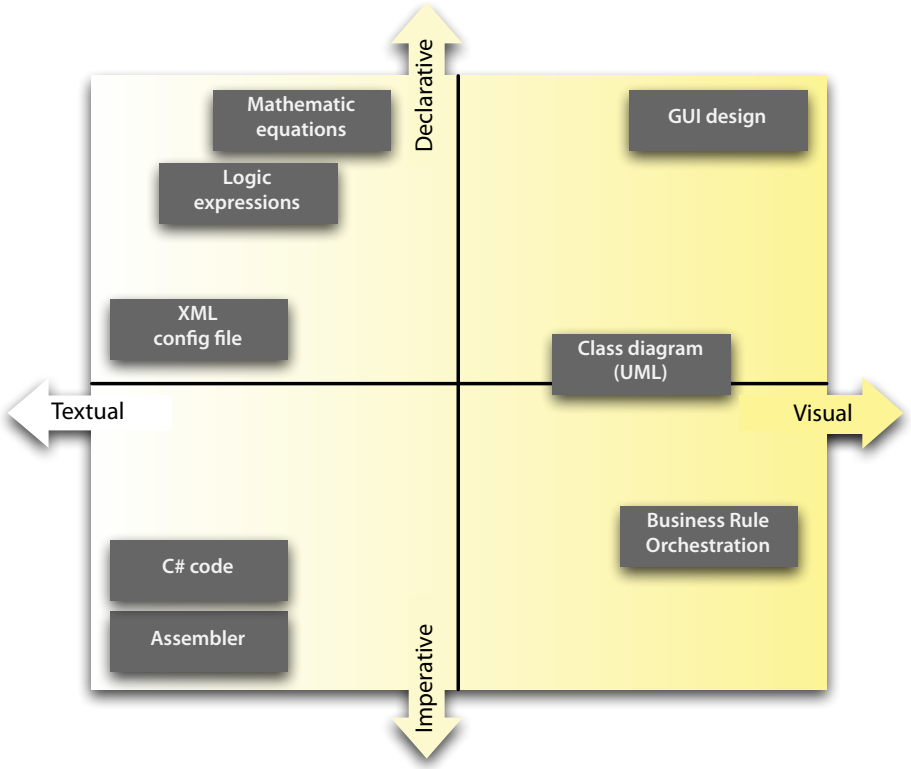


Figure 2.7: Variety of languages on a syntactic level.

2.6.1 Declarative vs. Imperative

This variation concerns the abstract syntax and is determined by how computations are specified. The notational nature of languages can be either declarative or imperative. Languages may have elements of both, but mostly languages can be categorised without too much trouble to either of the two.

Declarative Languages. Declarative languages are written in declarations, rather than assignments or control flow statements. This abstracts away from control flow and leaves it to the framework or compiler to deal with correctness and efficiency [59]. Moreover, declarative languages are considerably simpler than their imperative counterparts. The meaning of a statement in a declarative language is clear from the very statement itself, because it comes close to the expression of its intent, rather than its implementation. Most declarative languages are therefore great candidates for *specification*.

Declarative languages can be so concise because the tooling that handles them contains a lot of specific knowledge for the problem. The refinement that has been carried out at *tool development time* has reduced scope and added value. Declarative languages therefore score higher on the specificity scale.

Imperative languages. On the contrary to declarative languages, imperative languages are procedural languages. They are used for step-wise execution of instructive commands, in order to get to a computational result. Imperative languages are aimed at control, thus often low-level, thus far less abstract, which causes that the intent of a given piece of imperative

code is less obvious. Of course, giving the user full control over the program's control flow is exactly the advantage of an imperative language. However, from a DSL perspective, this is also the main drawback.

The pros and cons of declarative and imperative languages are thus exactly the ones that are imposed by the level of abstraction.

2.6.2 Textual vs. Visual

A second way of classifying languages is based on the concrete syntax. In other words, we may look at the presentation of the syntax elements of the language.

Textual Languages. Textual languages are represented in strict series of characters. Names are responsible for the identification of language elements. This is the most well-known way of specifying formal languages. Backus-Naur form (BNF) grammars for example use a textual format for the definition of the abstract language syntax.

Visual Languages. Visual languages are yet another representation of the same language elements that are possible to construct using textual languages. Visual languages offer graphical constructs that are not limited to patterns like words and sentences. Examples are boxes, lines, balloons, or icons, for the same language elements that textual languages use textual identifiers for. Elements do not have to be defined by their names, but can be identified by their graphical position in the composition, their shape or other visual elements, too.

Visual languages are mostly richer in the way that a picture can be clearer than a thousand words. The human brain is very visually oriented and the reason why a visual language can be clearer, especially for modeling (i.e. capturing intent) is because models are graphs in a mathematical sense. This contrasts with source code, which is tree-structured. This allows models to reflect and describe relationships among abstractions in a richer way than hierarchical media would [30]. As a consequence, visual languages are visually appealing to humans and the user gets a good intuitive notion of the semantics of the language elements relatively quickly [53].

The major disadvantage of visual languages is that they require specific tooling. Text is easy to handle for a machine, but graphics require a specific tool focused at the expressiveness of the language.

Summary

Industrialisation requires the development of tools to make people more productive. Software factories are that tooling and its use is concerned with the maturity shift of software development, lifting the development process of manual software *creation* into *generation*, analogous to manufacturing.

As we have seen, visual languages are good at expressing structure and relationships between abstractions. Given that implementations are renderings of intent through code, defined and limited by implementation technologies instead of natural intent, models must

be used for that purpose. Visual modeling can clearly reveal natural intent. Having established that the main drawback of models historically has been that they were too general and lacked formalisation, we may use DSLs to bridge the gap between the domain model and specific program implementations. Software factories use these DSLs because they are built on collections of domain-specific models that offer abstractions targeted to specific product families.

We have introduced the concepts of abstraction and refinement and their manifestations. Furthermore, we have shown how the abstraction gap arises and how software factories might overcome part of that gap by generating a large part of the software.

As an important part of software factories, we have defined the concept of domain-specific languages and shown their relationship: a DSL is the tangible part (like a machine) inside an intangible concept of a factory (which is an approach).

In the next series of chapters we will conduct a case study in order to assess the practical use and applicability of software factories to Sogyo. The theoretical foundations as laid out in this chapter are used to link practical insights to.

3 SCart: A Case Study

From the last chapter, we know what a software factory and a domain-specific language is. In the upcoming three chapters, we will demonstrate that such a software factory can be used in practice and can assist in the production of a series of similar applications.

We will demonstrate that by making an analysis of an existing case that typically is affected by Sogyo's business challenges by way of a case study. Subsequently, in Chapter 4, we will define a software factory capable of targeting the problem, based on that analysis. Finally, in Chapter 5 we will apply the software factory to create a re-implementation of the SCart application.

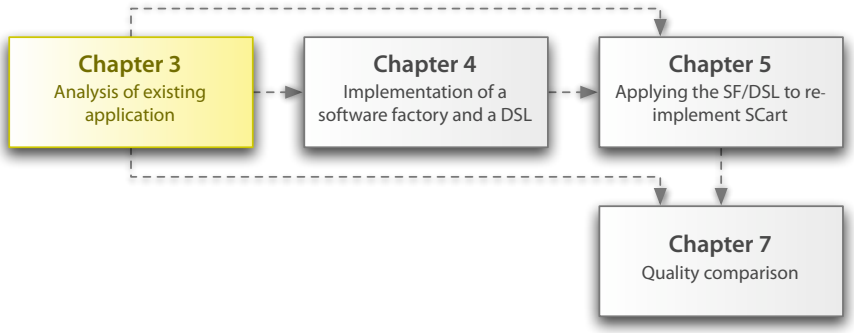


Figure 3.1: Chapter 3 in its context.

This chapter will introduce and analyse the SCart application, discuss the issues with its development and maintenance. Furthermore, we will point out several inconsistencies in the program and show that these are related to Sogyo's business challenges. Finally, we will argue why a software factory approach would fit the development of this application. We will conclude the chapter with a product-line analysis.

3.1 Introduction

In 2005, Sogyo built the software application SCart for the company Smitshoek Coördinatieprojekten B.V.[†], or "Smitshoek" for short. Smitshoek, a coordinating firm for construc-

[†] http://www.scp-a.nl

tion projects, being an intermediary between building contractors, real estate brokers, architects, lawyers, municipality, project developers, suppliers of goods and buyers of houses. Formerly, they used an Access application to manage all of the information involved in such projects. However, as the firm grew, performance trouble arose and the application became unmanageable from a maintenance perspective. Sogyo recreated a data-intensive software application for Smitshoek, called *SCart*, which was intended to be an application redesigned and rebuilt, but with an identical front-end interface, since Smitshoek was satisfied with the interface.

At first, the project was built using Microsoft's .NET framework 1.x, which had regretful support for data access and manipulation. It was especially incapable of managing large amounts of data in a production setting and it took seven months to deliver the first stable version of it. When the 2.0 version of the .NET framework was released, Sogyo rebuilt SCart in 2006 from scratch again in only three months.

SCart is used to administer any information related to the planning, coordinating and execution of construction projects, like buildings in a new neighborhood. This encompasses not only the technical implementation details of the actual construction projects, but also administrative data like company contact information, and the like. Therefore, both people involved in these projects and the Smitshoek secretary make intensive use of this application. For example, the application forms the central point for sending mass mailings or individual letters to companies or contacts, too.

Primarily, SCart is an application for use of Smitshoek personnel themselves to support their business processes, project data management and administer business relations. Secondly, Smitshoek sells the construction project's data by means of subscription to interested parties (typically building constructors). Business data is thus kept up-to-date in a central fashion and available to any party interested. Thirdly, Smitshoek might sell the application framework (without the actual business data) to competitors.

When houses are going to be built in a new municipal district, Smitshoek will create so called BOUWNUMMERS—construction numbers[†]. Each BOUWNUMMER represents a house or a building, either under construction or completed. A PROJECT is a bundle of BOUWNUMMERS, for example an office building or a whole neighborhood of buildings.

Each BOUWNUMMER may have OPTIES—options. OPTIES represent specific wishes or customizations. Typical OPTIES may be an extra bath room, a larger hall or a different color of electric socket.

3.2 The Usage Cycle

Since the SCart application is so data-intensive, it has a clear and intuitive usage cycle. At SCart's first run, it will connect to the remote web service that exposes the data from the central database. This involves a huge amount of data to be down-streamed to the local client, where it will be cached. The database's revision is recorded so that future invocations of the client will be able to download only the records that are outdated, which

[†]Unfortunately for this thesis, the project was created entirely in the Dutch language, which makes terminology somewhat unclear occasionally.

saves substantial network traffic.

After that, the main application's interface is shown, which is illustrated in Figure 3.2. There are such overview screens for BOUWNUMMERS (the default opening screen), OPTIES, PROJECTS, PERSONEN (contacts), BEDRIJVEN (companies), and ACTIVITEITEN (activities). These are shown on the left.

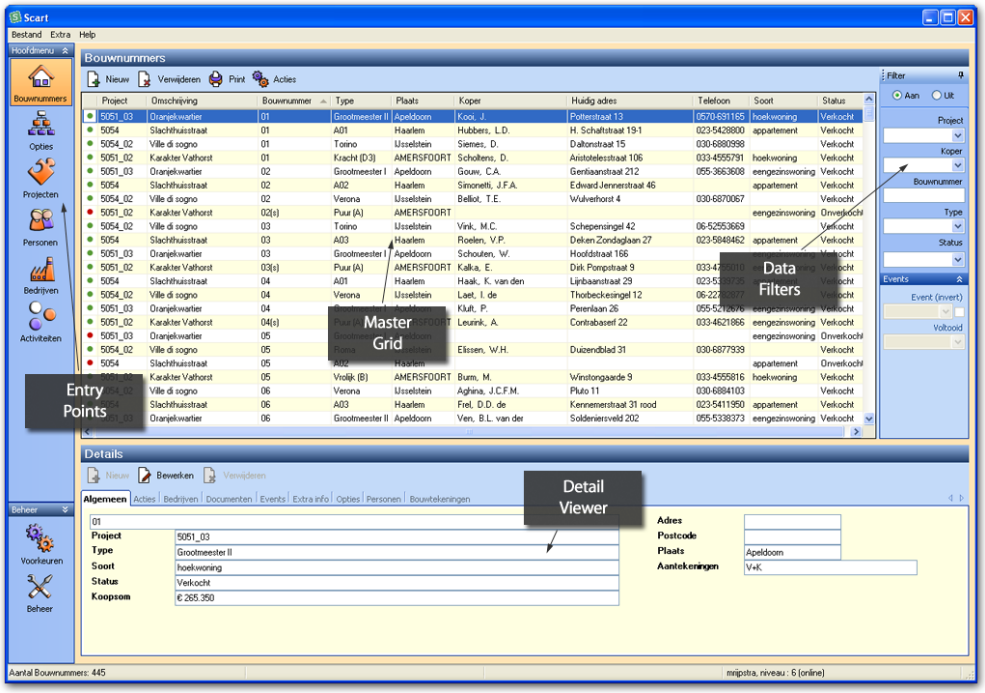


Figure 3.2: The SCart main interface.

At these overview screens, the data-intensity of the application becomes very clear immediately. Data grids show information which has a straightforward, similar representation in the underlying database. Each overview screen essentially represents one table. On the left, the user selects the overview screen he or she wants to read or edit. All rows are shown by default, but filters may be applied by the filter controls on the right side.

By selecting a row, the details of that row are shown in the detail window (on the bottom side). The detail windows consists of tabs that expose both the row's properties and indirect coupled information that is relevant to it.

The system's data can not be edited directly here. Modification of data is available only through the new and edit windows. Also, properties of relations can be edited in the various tabs of the detail screen. After each modification, the data is written back to the central database system so other client may directly visualize the changes.

Only Smitshoek themselves have write-access to the data and coordinates the input of the system. Data is read-only for each of the system's subscribers.

3.3 Design Characteristics

Due to the data-intensive nature of the application, the *table-driven design* (TDD) methodology guided the development of the application's base model. In TDD, the structure of the database design will be used as the basis for the model of the application. It typically implies that the model is derived from the database schema and extended with extra model information. This makes the model classes actual representations of database tables, the class' properties representations of table columns and the domain relationships representations of PK/FK-relations. Each object instance then represents a table row. The model classes and relationships even hold more information relevant to the generation of application logic.

3.4 Technical Architecture

The technical architecture of SCart is fully based on the underlying data model, which is quite a straightforward and intuitive notion when designing CRUD-intensive applications. However, once stuck with the database schema and the model derived thereof, the application is quite static and poor in a behavioural sense. In essence, the application core comes down to providing an interface that exposes properties from the database. From that point, semantics of concepts like projects or construction numbers have become irrelevant, since the application does not care what properties *mean*, but *which* properties exist. There is no such thing as behaviour anymore.

Nevertheless, the main application window exposes six views of the primary system data—one for each major application concept. These tables are interrelated through foreign key relations. For example, looking at the table `tblbouwnummers`, we see there is a column `IDproject`, which is relationally mapped to `tblprojecten`'s primary key.

This relation is reflected in the application's interface through the detail window. Each of the master views has an according detail window, exposing all of the relationships belonging to the selected master record through tabs. Following the example, selecting a row in the `Project`'s master grid, the detail window will expose, among others, the bouwnummers belonging to it (Figure 3.3).

Each of the master views works in this way, which lets us describe the behaviour easily. Although the actual data differs in each case, the behavioural structure of the application is quite simple in essence:

1. Master grids expose selective columns, but are actually simple views of tables. Adding new rows, editing existing rows and deleting[†] rows is possible using the action buttons on top;

2. Filters are predicates that expose or hide rows from the master grids. There are two types of filters:
   (a) Simple filters are property-value based predicates. They check whether a specific property adheres to the given value;
   (b) Custom filters are predicates that apply to multiple properties, or even to certain other, context-related information.

[†]Which, as we know, technically comes down to marking rows as deleted.

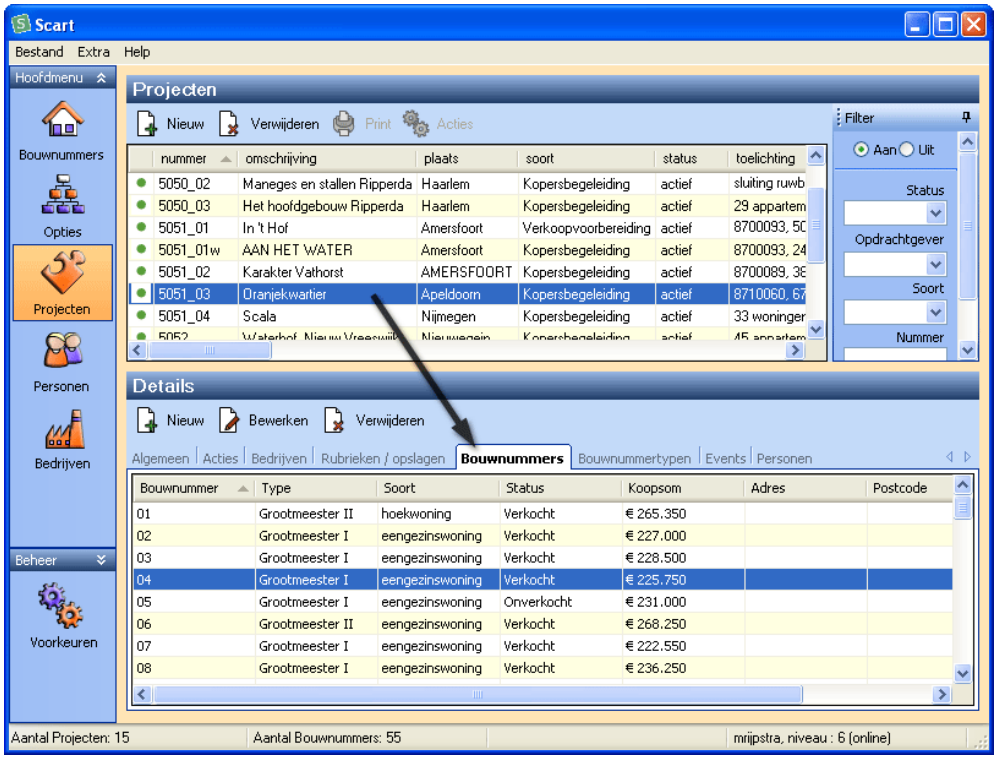


Figure 3.3: The relationship that links projects to construction numbers is visualised through the detail window. Other tabs show different relations.

3. Detail windows expose:
   (a) A specific view of the currently selected row in the master grid. This view might be a repeated view of the data that was already visible in the grid, but may expose more information (a property hidden in the master view, for example);
   (b) Other tabs that contain grids exposing related data. For each relation, there is a tab that allows for the modification of the relation. In some cases, this behaviour is not consistent, which is further discussed in Section 3.5.2.
4. Edit windows are dialogs that are used for modification of rows, in two ways: editing existing and creating new rows;
5. Validation constraints are used on the data entered from the edit windows to safeguard the field's individual correctness, as well as the correctness of the object data as a whole (constraints may span multiple object properties).

We will discuss each of these core application concepts in the following sections in somewhat more detail.

3.5 Implementation Details

Although the functionality of the application is easy to describe in a general fashion, the actual implementation is quite a bit rougher. Through the stapling of historical changes and changing wishes of Smitshoek, each of the application elements has been tweaked numerous

times, sometimes even being implemented as “exceptions to the rule”, breaking uniformity. All of this makes the application more and more unmanageable as time elapsed.

Not only did this render the application’s interface and behaviour less uniform, it also required a lot of manual programming and deep knowledge of how the application’s elements are connected. The following sections explain the technical implementation details that show how this design is troublesome.

3.5.1 Application Element Hierarchies

The application elements are all essentially created manually, deriving from base parental classes. This mechanism was invented to bundle trivial implementation aspects like handling the dialog GUI construction on the one hand and on the other hand to remain flexible at points in the implementation where custom behaviour was required.

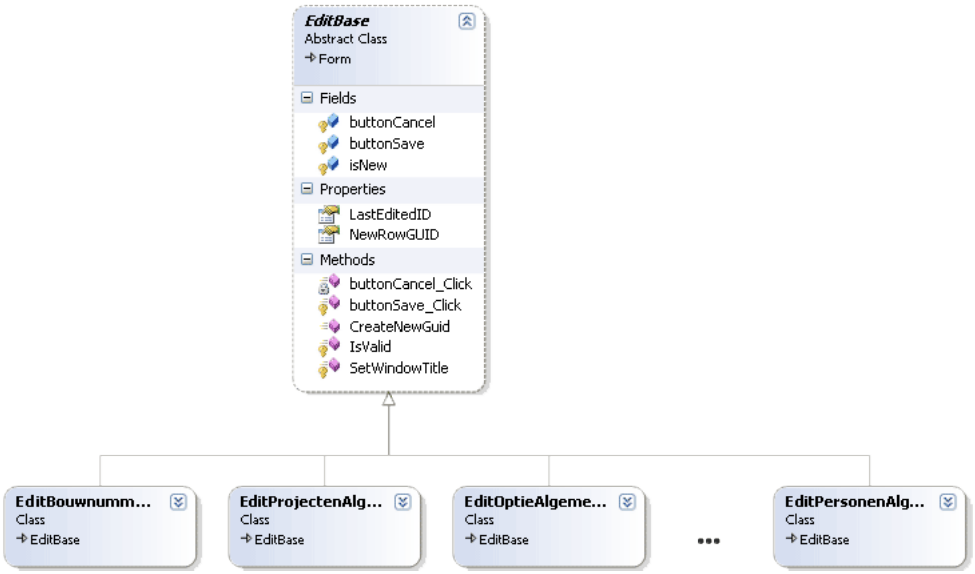


Figure 3.4: Inheritance of edit dialog windows.

Each edit dialog inherits from their parent class `EditBase` as shown in Figure 3.4. As visible, the `EditBase` class implements the common logic for the dialog window’s visualisations (dialog buttons and show/hide logic). Also, it incorporates a common body of logic to bind data generally. This is done using methods that support existing row modification (`LastEditedID`) or a new row creation (`NewRowGUID` and `CreateNewGuid()`). How the edit windows are invoked, is stored in the `isNew` boolean property.

Third and last, the `EditBase` class implements a common framework for validation, available through the `IsValid()` method, which prevents conceptually invalid records to be added to the database.

Although basic form state and behaviour is inherited smartly, the controls on the form’s surface pose the actual work that is to be done. Important in the current version of SCart is that all of these labels, controls, databinding and validation logic has to be written manually, which is an extensively labor-intensive job, considering that 58 of such forms exist in the current application.

Essentially analogous, the designs of the master panes, the detail panes and the filters are set up. They inherit from the classes `MasterBase`, `DetailBase` and `FilterBase` respectively. The basic functionality of these base classes is also visible through their properties and methods.

3.5.2 Inconsistencies

The project's core problem is the huge amount of redundancy. Where table-driven design might be the intended approach, clearly it has not been carried through thoroughly enough. Consequently, the application is inconsistent at several points. This typically is the case in situations where semantics exceed the naive implementation of plain table data. A few manifestations of such behaviour are:

- ▷ DOCUMENTS are read-only related to BOUWNUMMERS and PERSONS. Such relations are only possible to add through the sending of a mail and attaching a document;
- ▷ The manual character of the filters in the detail window allow for outrageous extensions. The tab "Contactpersonen" in the detail window of BEDRIJVEN shows multiple level relationships (Figure 3.6);
- ▷ The overview screen for ACTIVITEITEN does not show a detail window;
- ▷ When editing records from the detail window, *what* is edited is not always clear:
  - When editing a row BOUWNUMMERS in the detail grids of master rows from PROJECTS or OPTION, the *relationship* between the bouwnummer and the project/option is edited;
  - However, when editing ADDRESSES in via the detail grids of PERSONS, the properties of the addresses themselves are edited;
- ▷ The checkbox "voltooid" (completed) on the tab "Events" at the BOUWNUMMERS detail page can be changed inline—this was a specific requirement of Smitshoek to quickly be able to access this property since it is used very frequently. This is extreme inconsistent behaviour, since all other columns in the application are read-only.

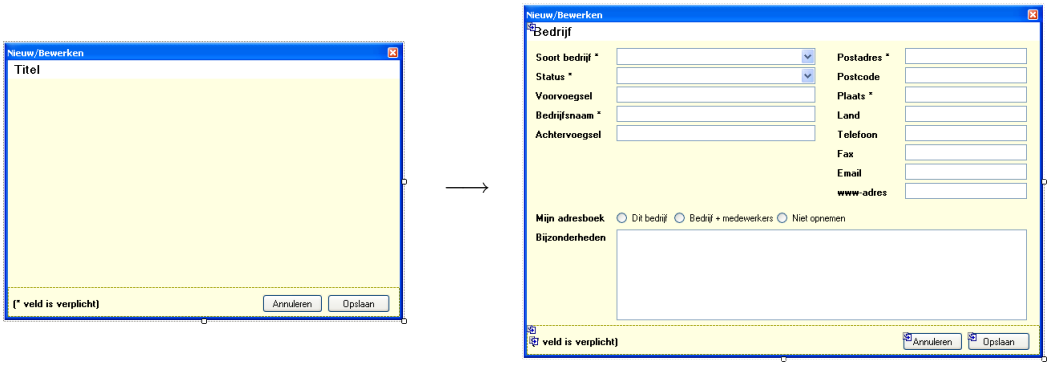


Figure 3.5: Inheritance is supported in the Visual Studio.NET dialog designer. On the left, the `EditBase` is shown. On the right, `EditBedrijvenAlgemeen` (EditCompaniesGeneral) is shown. All of `EditBase`'s properties are inherited, and fields are added on the container surface.

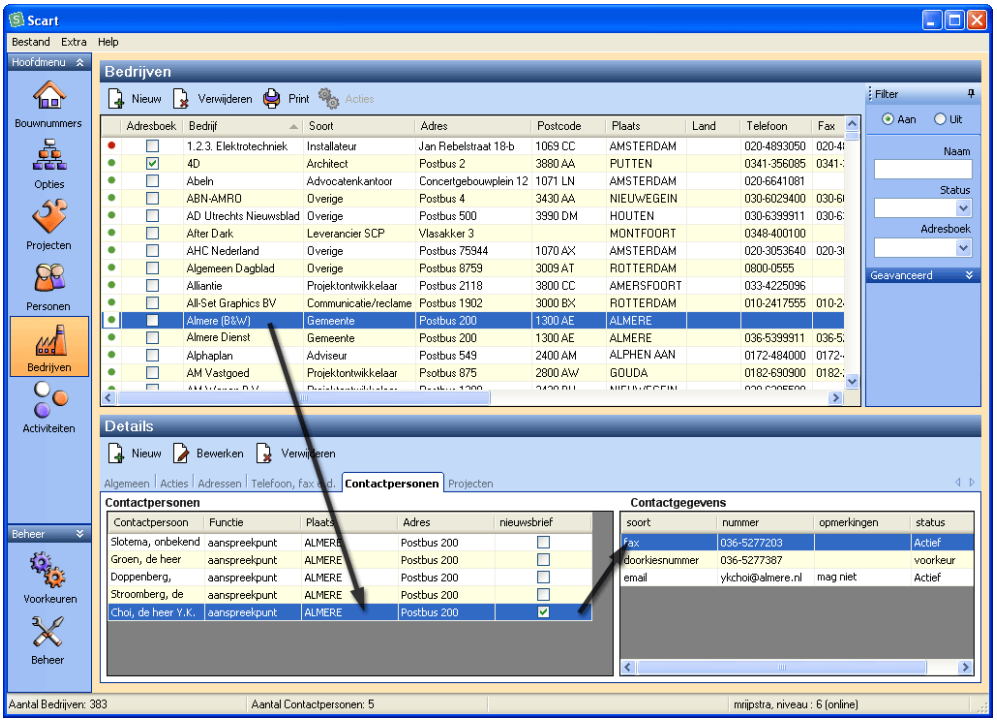


Figure 3.6: Inconsistent layout and behaviour of the detail window when viewing contact information.

3.6 Product-Line Analysis

Apparently, this calls for a different approach. The core problem of this case is the terribly unmanageable amount of labour involved in extending the application. The creation or modification of the GUI takes an extensive amount of effort, because specific accessing logic must be developed for each table manually. To visualise this, imagine that we want to add a table to the database, for example to add an employee administration. The application then has to be extended manually in the following aspects:

- Data set helper objects (`DataTables` and `TableAdapters`) to access the table through .NET proxy objects[†];
- A master grid showing the most common column data;
- A filter implementation for employee data, to search and filter data from the new table;
- A GUI filter pane for the master grid, to access the filter logic;
- A detail pane exposing a selection of the selected row's properties and allowing editing of that data;
- Filter logic and GUI for the detail pane;
- Add specific logic to implement relationships:
  - Parameterised filler function definitions (`FillBy<prop>()`) to the typed `TableAdapters`;

[†]This step does not take too much effort, since these can be generated by Microsoft's data set generator.

  - Logic that, based on user behaviour uses those filler functions to expose the data at run time.

- ▷ As part of the detail page, tab pages that expose any data relationships with the currently selected row;
- ▷ A GUI dialog and accompanying logic to:
  - create new data rows;
  - edit existing data rows.

  Fields that are exposed in this GUI dialog may differ for these actions—modification may expose other fields than creation.
- ▷ For each related table (e.g. `Company`, `Project`, etc.), extend their detail pane with a tab page to expose their relation to `Employee`.
- ▷ A data access entry point for the visualisation of the `Employee` table (the big icons on the left side of the GUI);

Furthermore, since SCart is available for reselling purposes, the framework will be used for other customers as well, making reusability increasingly important. Software factories form a great candidate solution to this problem.

Because of the specific, technical nature of the table-oriented object model that underlies the application, these tasks are well-suited to be implemented by automated processes. The specific and technical nature leads to applications that have a large common base and show only minor varieties. The next two sections will explicitly point out the commonalities and variabilities in an analysis that will be used in Chapter 4, where we will build a software factory that can be used to generate most part of CRUD applications.

3.6.1 Commonalities

Commonalities that have an internal character are commonalities that occur within an application more than once, while external commonalities are the ones that occur among a family of similar applications [48, 75].

In our case, since SCart is the only application, the internal commonalities can be recognised through repeating similarities, both functional and behavioural. For external commonalities, a common base among CRUD applications must be extrapolated, as far as that is possible based on one case.

The commonalities of CRUD applications can be described in three categories: *data entities*, *data relationships* and *data operations*.

Data Entities

The first commonality concerns the data entities that are available in the application. Given that, due to the table-oriented nature of the application's design, we have (flat) database tables as our main data sources, we will reflect that flat data projection using data grids in the visualisation of the data entities on-screen.

Data entities are visualised using views. Such a view is essentially constructed using subset selection or combinations of tables. This is what might be familiar from the SQL language (`SELECT` and `JOIN`). Such operations can either result in a request for all the table's data, a specific constraint-based selection, or a constraint-based combination with other tables (an SQL join).

There exist four types of these basic view creations: selections of subsets or combinations with other tables, both in either horizontal or vertical orientations. Examples of all four in SQL pseudo notation are:

- **Horizontal selection** or *column selection*:

$$\texttt{SELECT } t(c_1), t(c_2), \ldots, t(c_N) \texttt{ FROM } t$$

- **Vertical selection** or *row filtering*:

$$\texttt{SELECT } \forall_c.\, t(c) \texttt{ FROM } t \texttt{ WHERE } condition$$

- **Horizontal join**:

$$\texttt{SELECT } \forall_c.\, t_1(c),\ \forall_d.\, t_2(d) \texttt{ FROM } t_1, t_2 \texttt{ WHERE } condition$$

- **Vertical join** or *unioning*:

$$\begin{gathered}\texttt{SELECT } t_1(c_1), t_1(c_2), \ldots, t_1(c_N) \texttt{ FROM } t_1 \\ \texttt{UNION} \\ \texttt{SELECT } t_2(d_1), t_2(d_2), \ldots, t_2(d_N) \texttt{ FROM } t_2\end{gathered}$$

  where:

$$\begin{aligned}\text{Type}(t_1(c_1)) &= \text{Type}(t_2(d_1)) \\ \text{Type}(t_1(c_2)) &= \text{Type}(t_2(d_2)) \\ &\vdots \\ \text{Type}(t_1(c_N)) &= \text{Type}(t_2(d_N))\end{aligned}$$

With $t, t_1, t_2 \in \mathbb{T}$, the set of all tables. Also, $t(c)$ represents the specifier of column c in table t, denoted in pure SQL as `t.c`. The notation $\forall_c.\, t(c)$ is the representation of SQL's `t.*`. Finally, $condition$ is a predicate expression, which may (and typically will) use variables defined in prior statements like `SELECT` or `FROM`.

Views are the main visible elements in the application. Each view is basically the result of one of these four operations, with either tables or other views as operands.

Data Relationships

Because of the ability of relational database systems to link tables via primary/foreign key relations, there also is a specific need to visualise those relationships on-screen. Essentially, there are four kinds of relationships possible cardinality wise.

- ▷ **Many-to-one.** One row field of the source table references a single row field of another table (the target table). Multiple rows in the source table may thus reference the same target row. An example would be that each `Project` has exactly one executive `Company`. Multiple `Projects` may thus have the same executive `Company`;

- ▷ **One-to-many.** The same, but inverse relationship. The example is also the inverse: each `Company` may be the executive company for a series of `Projects`. However only one company may ever be involved in execution of each `Project`. Any one-to-many relationship can be expressed in terms of a many-to-one relationship.

- ▷ **One-to-one.** This is a specific relationship which is not useful conceptually, since any one-to-one relationship should be normalised into a single table.

- ▷ **Many-to-many.** In a many-to-many relationship multiple row fields of the source table may reference multiple rows in the target table. This always requires an intermediate table. Example: a `Company` can have multiple `Employees`, but `Employees` may work at multiple `Companies`.

Hence, with data entities, a many-to-one relationship, and a many-to-many relationship, we can express any relational data model.

Data Operations

The essential functionality of a CRUD application is its exposure of the four core CRUD actions. For each of these, we will discuss how SCart handles them, and what are general commonalities for such applications.

Create. Creation of new rows can be invoked in SCart via the "New" button, which is available for all data entities (in both the master and detail grids). Typical creation procedures have the following logic:

1. Generate a new, unique identifier, a GUID;

2. Show a dialog window with editable fields, representing fields from the table structure;

3. When pressed OK, insert a new row into the table, using the new GUID for the primary key field and the data from the dialog in the other columns.

This logic makes a few assumptions, which we will make explicit here:

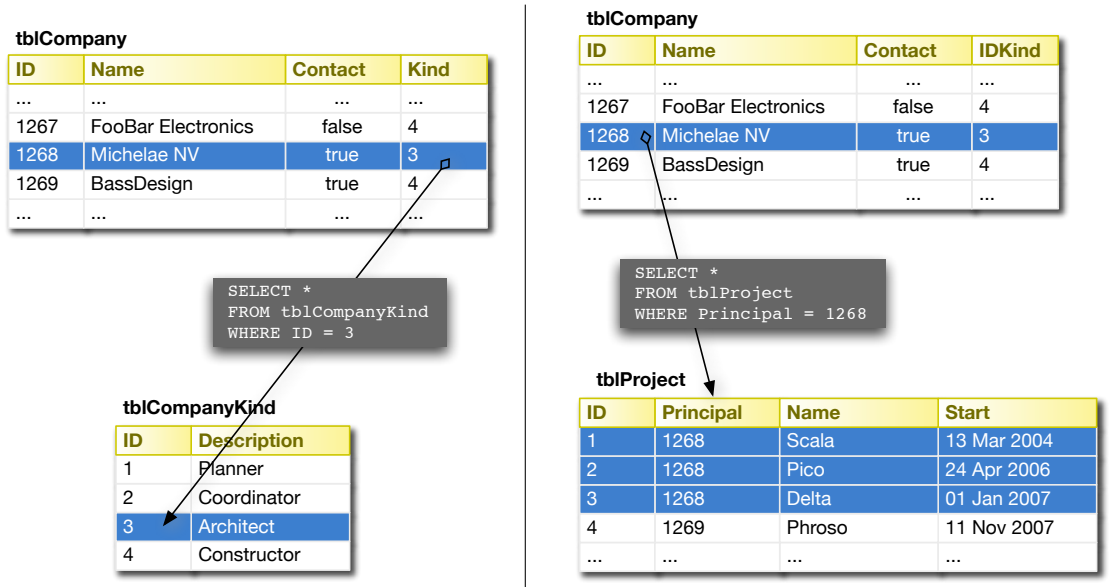
1. Each table has a single column that serves as the primary key column. Primary keys can therefore not be declared over multiple columns;

2. For each such table, for each column that is *NotNull* (i.e. is a mandatory column) the dialog must provide an input control (or a default value).

Read. There are several situations in which reads are performed. First of all, the master grid that is visible in the main SCart interface is a simply projection of flat data from the according view. In most cases, the data is just a simple view, namely the table itself or a horizontal subset (some system columns may be hidden from the end user). But, although this does not occur in SCart, in essence any view can be shown, including joins of tables.

Next, there is filtering support. When the end user specifies some (column-based) search queries on the right hand of the master or detail grids, a row filter applies, which is nothing more than a simple implementation of a vertical sub selection.

The third situation is the relational selection to expose the related data. Each of the data relationships needs to be visualised during runtime. This is done in SCart by dynamically changing the detail pane in the application, based on the currently made selection in the master grid. Based on the specific relationships that exist with the row, different behaviour is desired. For both many-to-one and one-to-one relationships (we will use the notation *-to-one for that from now on), this might mean that data is either represented as properties that directly belong in the master grid's data entity structure (i.e. "hiding" the relation). For *-to-many relationships, a grid is used to visualise the look up that is performed to gather the related information. This requires a different look up structure (see Figure 3.7).



a. *-to-one relations. **b.** *-to-many relations.

Figure 3.7: Look up related data. With *-to-one relations, selection of the target table occurs on the primary key. With *-to-many relations, selection happens on the referencing column (in this case "Principal"), based on the source table's primary key column.

Update. Updating (editing) of existing rows can be invoked in SCart in two ways. First, selecting a row in the master grid, the detail pane with editable controls for the row's fields will become visible. Secondly, double clicking on a row will open a dedicated edit dialog window, which (also) includes controls to edit the data. Although both actions lead to an edit action conceptually, technically the behaviour might be ambiguous due to a non-overlapping code base. In other words: the editable fields that are available through both options differ.

In essence, this is not desirable, because next to creating an ambiguous interface for editing,

both options must be written manually. Currently, SCart also lacks a central edit routine for each data entity and this functionality is implemented twice for each edit action. We will pose a solution to this problem in our software factory.

Typical update procedures have the following logic:

1. Read the existing data from the row and read the primary key's value;
2. For each column except the primary key, create a specific control that can edit the value for that control;
3. Initialize the control's value;
4. Let the user edit the data using the controls;
5. Validate the new values;
6. If passed validation, commit the changes.

This logic makes the assumption that data entities are representable using a specific control. In fact, this is only the case with primary data, not when data resides in more complex structures like lists or structures, or when data is generally not representable using only one control.

Destroy. Finally, data rows may be destroyed (deleted from the table). A destroy action within the table is not interesting, because that merely comes down to the removal of the row. However, there are complicating factors:

- ▷ other fields may refer to the row—these relations become invalid;
- ▷ the row may reference other fields—the other fields may be left dangling.

Both are tackled by specifying correct delete behaviour for each situation in the database's relation specification.

Furthermore, like most CRUD applications nowadays, SCart is designed to never remove a row from the database physically, but to mark the row as deleted, in order to satisfy the popular "never delete" business rule. The "never delete" philosophy has a few advantages, mainly (but not exclusively) in a non-technical sense. Auditing, making business decisions, reporting to management and legal issues are a few. The rule itself poses only a small penalty to the system, because each read action must make an explicit vertical selection based on the row's alive mark.

3.6.2 Variabilities

Given the common factors of a CRUD application as pointed out in the previous section, we may distinguish the variabilities that identify the product family's members. The variabilities that are identifying for each CRUD application are all related to *what* data is managed, not *how* [48, 75]. We distinguish a few properties in this discussion: entity signature, column types, field visibility, relationship visibility, and custom validation.

Entity signature. The primary identifying variability of CRUD applications concerns *what* data is managed in the CRUD life cycle. This data is defined by its signature. What is meant by that, is the structure of the data. In essence, such data entity signature is defined by the database structure that the application operates on. Such an entity signature is defined by:

- ▷ **Tables.** Tables have a unique, identifying name, and a set of one or more columns that defines its structure.
- ▷ **Columns.** Column have a unique, identifying name, a data type and a column type. Their data types and column types are elements from sets of predefined values.

Besides the signature of these entities, the specification of which tables serve as entry points for the application to start managing the data is also a variability.

Relationship signature. Another key defining factor is the relationship signature. Each relationship is a match between values of column data. Relationships are directed and defined using foreign key / primary key references (FK/PK-references) to columns, which are the relationship roles.

Field accessibility. Field accessibility is the variability that defines which of the columns in the data entity definition are accessible through the application. Accessibility can either be defined in terms of visibility or representation of data. The latter comes down to the representation of data on screen rather than by their default database types—for example, although a column is of type string, its value may be well picked in the GUI by a date picker.

Field accessibility is defined per CRUD action—creation, updating and reading actions may each require different field access.

Relationship representations. Like field visibility, relationship representations must be defined for each relation. Besides visibility, the representation of relationships may take different forms in case of *-to-many relationships. In that case, the related data can be conceptually regarded as belonging to the referencing row. A visual representation matching that point-of-view is to incorporate the referenced data in the master grid as a series of extra columns joined to the master table. Alternatively, representation of the referenced data can be shown in a secondary grid.

Custom logic. CRUD applications may require additional logic apart from the core logic facilitating CRUD management. A typical example would be the provision of checks on data values that would otherwise form data that conceptually has no semantics. Validation may therefore be required at all CRUD actions. Validation checks on data can take so many forms that any additional validation checks should be implementable by hand.

In fact, any custom logic that is foreseeable *a priori* should be implementable based on predefined events throughout the application's architecture.

Summary

In this chapter, we have introduced the SCart application that Sogyo has developed and still maintains. We have structurally analysed the program and have made clear that there is a tremendous amount of tedious manual work to be carried out in order to create the application in the first place and to extend it.

Furthermore, we have pointed out several inconsistencies in the program, which are mainly due to the manual *ad hoc* reuse that is used when constructing the application. We have also seen that these tasks may well be candidates for automation.

Finally, we have made a case as to why a software factory approach would fit automation of these tasks and made a product-line analysis of the application. In the next chapter, we will define a software factory schema and define a DSL that is capable of specifying CRUD-based applications.

4 The CRUDE Software Factory

Based on the product-line analysis we concluded with in the previous chapter, we will in this chapter develop, a software factory and an according DSL that is capable of specifying CRUD-applications. It is called CRUDE, as a pun on CRUD.

There are several purposes to this chapter. First of all, we want to give an introduction to the workings of Microsoft DSL Tools, simply to provide some background information on the metalanguage that is used to construct DSL's with. Secondly, we describe how our language CRUDE is constructed and how it is used to generate code from. Thirdly, we show how CRUDE satisfies the specification from the product-line analysis from the previous chapter.

In Chapter 5, we will actually apply the DSL we create in this chapter to show that it can be used to implement CRUD-applications with.

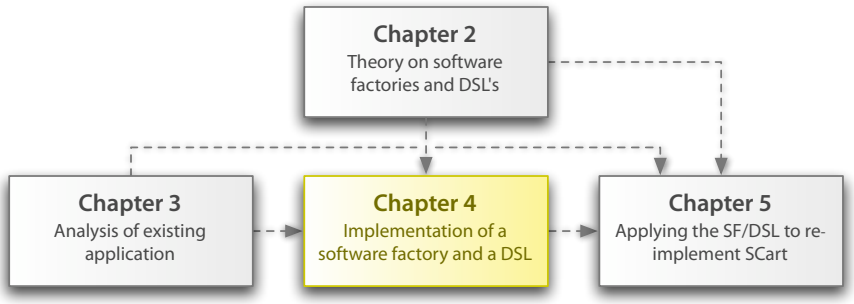


Figure 4.1: Chapter 4 in its context.

4.1 Factory Design

This chapter is devoted to the construction of the CRUDE software factory. Before we go into the details, it is important to realise what the exact goal of our software factory is.

The software factory is designed to ease the process of coming from a specific database to a graphical user interface (GUI) that allows a user to edit that database in a user-friendly fashion. In other words, we want to automate the process of constructing a GUI for CRUD applications based on the database model.

In this section, we will give an overview of the factory. Structuring down the design, we will initiate the discussion with some starting points. Next, we will show the input and the output of the factory, followed by the factory internal operations that achieve that transformation.

4.1.1 Starting Point

The context from where the software factories philosophy became necessary, is the amount of tedious work to be done to come from an existing database model to a working CRUD application that manages that database. The effort lies in the step from the database to the application. The intellectual modeling effort for the application has actually been carried out during the database structure's design.

The software factories philosophy assumes a domain model that forms the source for the development process. To this domain model, services can be "attached" that use specific domain logic to implement their functionality. A typical example is that both a persistency service and a GUI can be generated from such a domain model (see Figure 4.2).

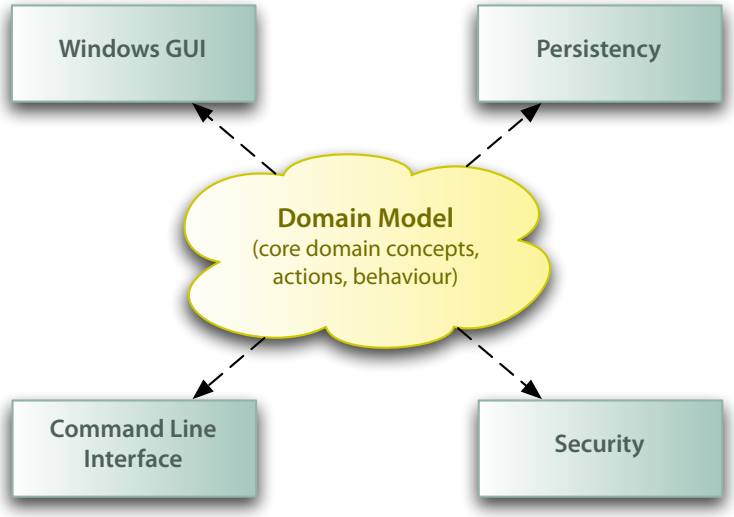


Figure 4.2: Model-driven design.

However, one can ask himself whether the use of software factories fits the table-driven approach SCart is developed with. Following Figure 4.2, we must come to the conclusion that the way to implement the factory much at least have a central domain model, where the database is simply considered to be a persistency service. We will argue here for several reasons why such reasoning might in fact be too shortsighted:

- The persistency *is* in fact the core of CRUD applications. There might be an interface, but that simply functions as a gateway to the application's core actions. Persistency must not be seen as the product to generate, but as the spindle around which the application is built;
- The database design exists already. To simply ignore this fact and centralise a new domain model in which the same database design has to be generated means that nearly all functionality of the database development environments must be implemented. This not only means that year-long work by specialists must be re-implemented, but eventually this step would not leverage the desired gain in productivity.

▷ The database model is interwoven in a CRUD application's nature. Everything is about the database model. A domain model for such applications will be a model that will not differ much from a table-oriented model. Why not simply start from that anyway?

This might prove to be a confusing element in this case study, because the domain model does not concern an actual end-user domain, but a technical domain—the database domain from the point-of-view of a software developer.

4.1.2 Factory Input and Output

The previous discussion made clear that most ideally, the input of the software factory is a domain model, but that we may suffice with a model based on a database structure as the source for the generation of services for good reasons in practice.

The input for the factory consequently is a database structure. Depending on the concrete table information from the database, a GUI (i.e. a service) must be generated that enables the management of the data and satisfies the requirements as specified in the previous chapter.

The output is a functional CRUD application, or better, the *source code* for a CRUD application. The advantage of the delivery of source code over a compiled application is of course that manual tweaking can be conducted in the form of extensions of the programmatic logic, before it is compiled. Besides graphical Windows Forms components that take care of the correct display of on-screen data, standard CRUD related programmatic logic is generated for every data entity.

4.1.3 Factory Operations

Based on the input and output of the factory, we may have a more in-depth look at the realisation of that output. We shall design a domain-specific language that makes it possible to specify CRUD applications, given an existing database. The software factory schema then looks as in Figure 4.3.

The numbered arrows in this schema show the semi-automated steps in the process—tasks where part of the work must still be done by the human programmer. We shall detail each step below:

1. From the database structure, a representation must be derived that can serve as a source for the generation of application functionality. This representation is the application model and is described using a DSL (CRUDE). More details are given in the following section[†].

   In essence, there is no mechanism that safeguards the consistency between the database structure and the application model. Hence, this forms the fundamental objection to this approach. Theoretically, we would rather walk the path the other way around—this is marked as "desired implementation" in Figure 4.3. In that approach, the database structure would be generated from the application model instead. The

[†]This approach resembles the MDA approach [22, 44, 52]. A striking difference between the approaches, though, is that MDA follows a broader philosophy, namely the generation of software from specifications given in a general-purpose language (MDA uses UML, for example). We have already made an argument why the domain-specific approach is much more manageable in Chapter 2.

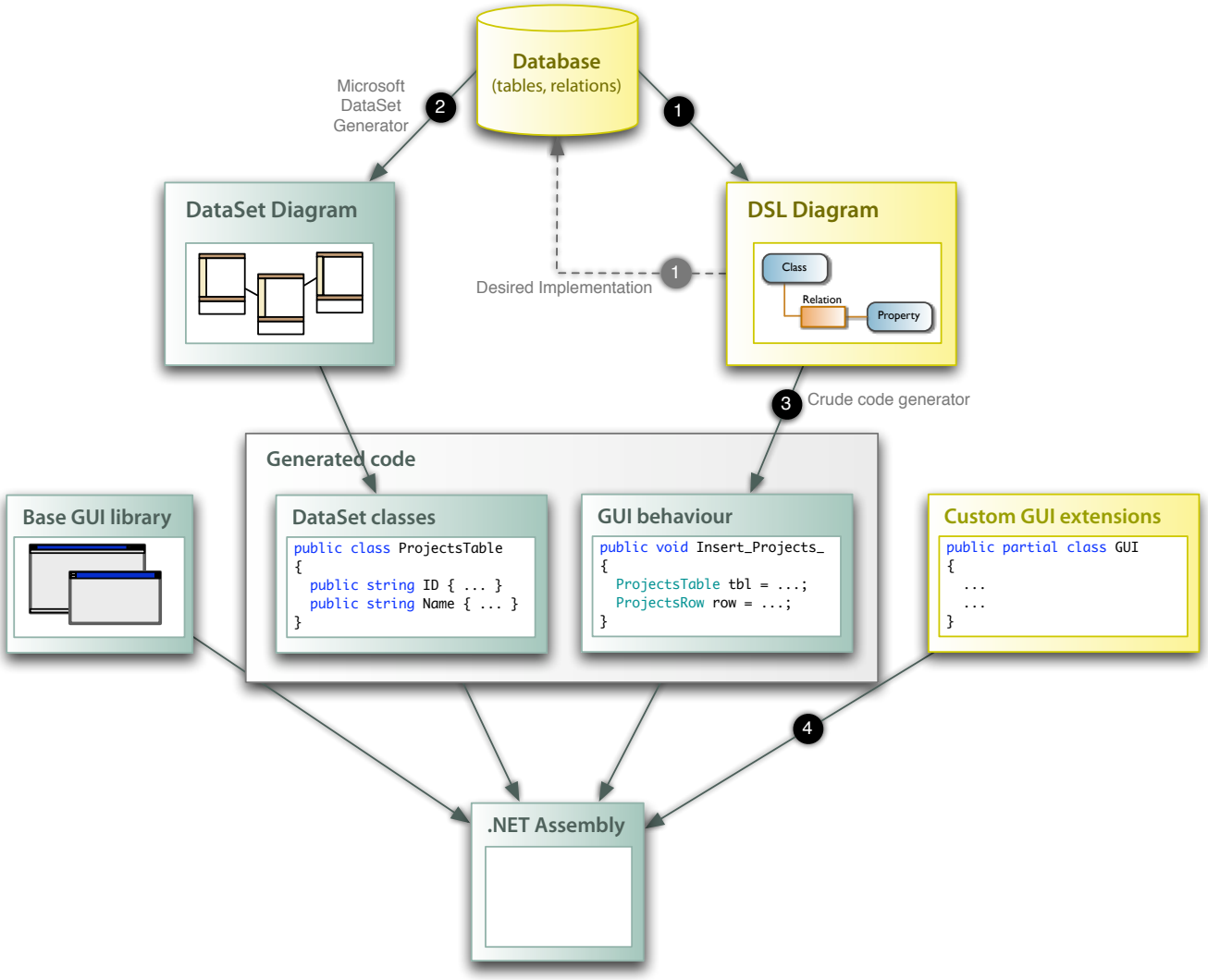


Figure 4.3: The software factory schema, showing the process of using CRUDE to construct a CRUD application from a database.

reason why we would prefer this is that there are two sources that must be kept in sync. This must now be done manually. Duplication of this kind of data leads to redundancy quickly—something that should always be expelled.

Now, when we would choose this alternative implementation (i.e. the application model is leading), there is enough information available in one source (the model) too both generate the database structure as well as the application logic[†]. However, making that implementation has technical consequences. The database structure contains a lot more information, like stored procedures, views, diagrams, delete behaviour rules and custom validation guards. Many exist in the current SCart application even. The model should be extended with information of these properties of the database, while the properties are essentially irrelevant to the application logic. These details would obscure the model instead of clarifying.

[†]By the way, arrow (2) can be left as is. It does not have to originate directly from the application model. After all, arrow (2) is reachable from arrow (1) in a top-down descending path, so all information can be generated sequentially.

2. From the database, a DataSet is generated—a proxy layer which can be addressed to access the database. The proxy support for DataSets stems from the Microsoft .NET 2.0 Framework. Its generation is fully automated, but is to be invoked manually and, although separately from the creation of the application model, must be based on exactly the same database source as the model is based on. The step itself is trivial and requires no more than clicking the "Finish" button of the data set generation wizard. Exactly for that reason, we value the separated generation acceptable.

3. The next step encompasses the generation of application code from the application model. This happens through the CRUDE code generator, which will be fully laid out in Section 4.3.2. This step is completely automated and requires no more than a key press. After the application model gets saves, the code generator is automatically invokes to directly reflect the model changes in the application code.

4. In the final step, we can extend the generated application where possible. This step concerns the addition of programmatic logic of work that can not be attributed to a standard CRUD application, but typically belongs to a specific target application—a special case of a CRUD application.

   When the implementation of the custom logic is finished, the combined source may be compiled into a .NET assembly.

4.2 Domain-Specific Language Design

To realise the goal to come from a database towards a working application, we shall create a DSL that makes it possible to:

- ▷ express specification for CRUD applications;
- ▷ automate the implementation through code generation.

Before diving into the details of the language itself, first we have to select the meta-DSL in which we will describe our own DSL. In our case, of course, we have used Microsoft's DSL Tools. DSL Tools offers the creation of visual, declarative models. So given Figure 2.7 on page 25, DSLs created with DSL Tools will be located somewhere in the upper-right corner.

Recall the elements of a formal language from Figure 2.6 on page 24. Each language consists of an abstract syntax, a concrete syntax and semantics. The next sections will describe the CRUDE language in these terms.

4.2.1 Abstract Syntax

This basis for the domain-specific language grammar is formed by the product-line analysis from the previous chapter. The commonalities are translated into model objects, while the variabilities are implemented using parameters of these model elements.

The foremost commonalities are modelled in domain classes and domain relationships, which we will discuss below. Part of the framework for this software factory is derived from

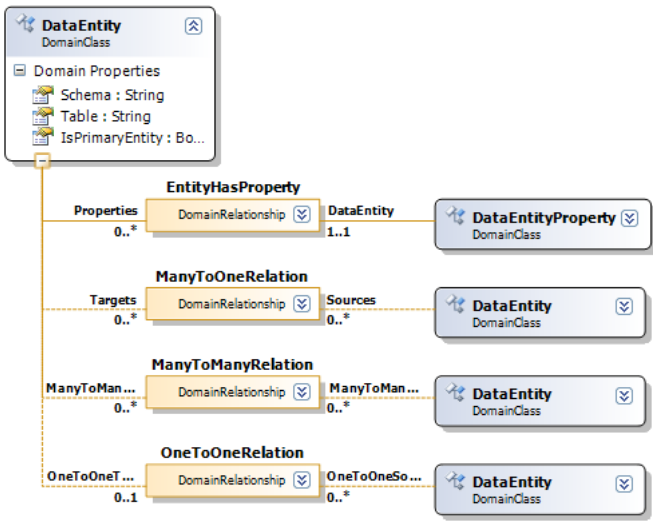


Figure 4.4: The abstract syntax elements for the CRUDE DSL.

the open source tool ActiveWriter[†], a DSL for generating NHibernate proxy classes based on a database model by Gokhan Altinoren.

Domain Classes `DataEntity` and `DataEntityProperty` are the most important domain classes. The link that exists between these classes is an embedding relationship (`EntityHasProperty`) with two roles (`DataEntity` and `Properties`). Each domain class has its own signature. In `DataEntity` it is the following:

$$\texttt{DataEntity} ::= \langle \quad \textbf{Schema} : \text{String}, \\ \textbf{Table} : \text{String}, \\ \textbf{IsPrimaryEntity} : \text{Boolean} \rangle$$

The properties `Schema` and `Table` are used to realise the coupling with the underlying database. The combination of these properties allows for addressing the database for the execution of actions. The property `IsPrimaryEntity` is used to specify whether a `DataEntity` functions as an entry point for the CRUD application (recall Figure 3.2).

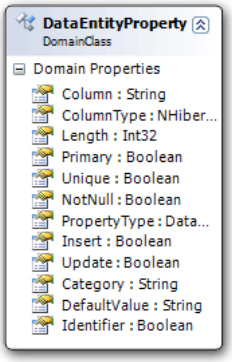


Figure 4.5: The `DataEntityProperty` domain class.

[†] `http://altinoren.com/activewriter/`—visited April 5th, 2007.

The representation of properties of data entities is provided through `DataEntityProperty` (see also Figure 4.5):

$$
\begin{aligned}
\texttt{DataEntityProperty} ::= \langle \quad & \textbf{Column} : \text{String}, \\
& \textbf{ColumnType} : \text{NHibernateType}, \\
& \textbf{Length} : \text{Int32}, \\
& \textbf{Primary} : \text{Boolean}, \\
& \textbf{Unique} : \text{Boolean}, \\
& \textbf{NotNull} : \text{Boolean}, \\
& \textbf{PropertyType} : \text{DataEntityPropertyType}, \\
& \textbf{Insert} : \text{Boolean}, \\
& \textbf{Update} : \text{Boolean}, \\
& \textbf{Category} : \text{String}, \\
& \textbf{DefaultValue} : \text{String}, \\
& \textbf{Identifier} : \text{Boolean}\rangle
\end{aligned}
$$

The properties `Column` through `NotNull` are actually the relevant properties from the database table. We will copy these properties into the model (recall that there is no tight coupling between the table and the model). Important with that is that the exact structure of the database is reflected in the model accurately.

The properties `Insert` and `Update` are used to specify the accessibility of the column property in the application. A property that has `Insert` specified, is visible in the edit window during insertion of new data rows to the database. Similarly, `Update` specifies visibility during edit actions.

Next, the property `PropertyType` specifies the kind of column. We need a supportive construct for this—the so called domain types. Domain types are references to externally defined complex data types or domain enumerations. Some columns in the database do not serve a public purpose and therefore do not have to be visible in the application. System columns, for example a typical `ID` column that makes rows uniquely addressable, typically fall under that category. Such columns are not relevant to the end user of the application. Hence, we make a distinction between three property types through a domain enumeration:

$$\texttt{DataEntityPropertyType} ::= \quad \textbf{Hidden} \mid \textbf{System} \mid \textbf{Visible}$$

We will briefly clarify the alternatives:

- ▷ **Hidden.** The column will always be hidden in the interface and will be ignored as far as the application is concerned;
- ▷ **System.** The column will always be hidden in the interface, but will not be ignored in the application logic. This might be because the column requires system logic or should adhere to validity rules;
- ▷ **Visible.** The column is a regular, visible column in the interface.

Finally, the properties `Category` through `Identifier` determine the way data is shown in the application. `Category` is used to group properties over tab pages in edit mode and

`Identifier` will visually format the column data in a different layout, to make it stand out visually (in large tables, finding the natural identifying column may be rather inconvenient).

Domain Relationships Three relationship types exists between data entities, as argued in Section 3.6. These are modelled in CRUDE as reference relationships. They represent the relations that may exist between data entities: one-to-one, one-to-many, many-to-one and many-to-many. Also, we have already seen that one-to-many and many-to-one relations only differ in direction, so the design may suffice providing merely a many-to-one relation.

- ▷ **One-to-one.** This is the most straightforward relationship and nothing more than a reference. The relationship holds only four properties:

  `OneToOneRelation` ::= ⟨ **SourceDescription** : String,
  SourceRelationDisplayName : String,
  TargetDescription : String,
  TargetRelationDisplayName : String⟩

  The `*Description` properties are purely informative and may be used to describe the nature of the roles informally, as a hint to the users of the DSL. The `*RelationDisplayName` properties are used in the GUI to indicate the relations with a user-friendly name.

- ▷ **Many-to-one.** With the many-to-one relationship, there exists the notion of a source and a target table. The source table is always the table that plays the primary key role in the PK/FK relation, while the target table is the one playing the foreign key role.

  `ManyToOneRelation` ::= ⟨ **TargetColumn** : String,
  TargetDescription : String,
  TargetRelationDisplayName : String,
  SourceColumn : String,
  SourceDescription : String,
  SourceRelationDisplayName : String⟩

  The properties are mainly identical to the one-to-one relation, except that there are two extra properties, `*Column`, specifying between which columns the PK/FK relation is defined.

- ▷ **Many-to-many.** With the many-to-many relation, there is an intermediate table in the database (holding all the primary key–foreign key combinations). The many-to-many relation is very identical to the many-to-one relation, but contains some additional

fields `Schema` and `Table` to address the intermediate table:

`ManyToManyRelation` ::= ⟨ **Schema** : String,
Table : String,
TargetColumn : String,
TargetDescription : String,
TargetRelationDisplayName : String,
SourceColumn : String,
SourceDescription : String,
SourceRelationDisplayName : String⟩

Finally, all the data entities are "held together" by the master model object through the embedded relationship `CrudeModelHasEntity`, which is nothing other than a list of all the data entities that are contained in the model. This makes is possible to address any domain object. This model object is the entry point for access to all model elements (classes, relationships). Besides access to these objects, the model itself also contains some properties that can not be attributed to any of the classes or relationships:

`CrudeModel` ::= ⟨ **Language** : CodeLanguage,
MainGuiTitle : String,
MainGuiClassName : String,
TypedDataSetClassName : String,
TypedDataSetNameSpace : String⟩

The property `Language` makes it possible to choose a programming language in which the application source code should be generated. Essentially, any .NET language may be outputted. CRUDE currently provides a choice between C# and Visual Basic.NET.

The properties `MainGuiTitle` and `MainGuiClassName` offer respectively an appetising title for the application's main GUI window and an internal .NET class name for the main window's class to be generated.

Recall the discussion from the beginning of this chapter. The application to be generated assumes an available typed DataSet proxy system to communicate with the database. We already saw that this typed DataSet can be automatically generated by an excellent Visual Studio wizard. CRUDE is build to work with a plain typed DataSet (e.g. without any customisation in the wizard). Exactly this makes the generation so trivial. What remains a human matter, though, is providing the typed DataSet class location to CRUDE through the properties `TypedDataSetClassName` and `TypedDataSetNameSpace`.

4.2.2 Concrete Syntax

The concrete syntax of DSLs in DSL Tools is given by so called Shapes and Connectors. These define the visual elements in the model designer as they are shown on the model diagram surface. The underlying classes and their relationships are modelled by editing the shapes and connectors. There is a direct relation between domain classes and shapes, and between

relationships and connectors. These relations are specified in the DSL definition using so called diagram element maps.

Shapes and connectors are actually descriptions of how classes and relationships should be represented visually. They are descriptions of which class information should be shown or hidden in the model diagram, and how that information should be formatted visually.

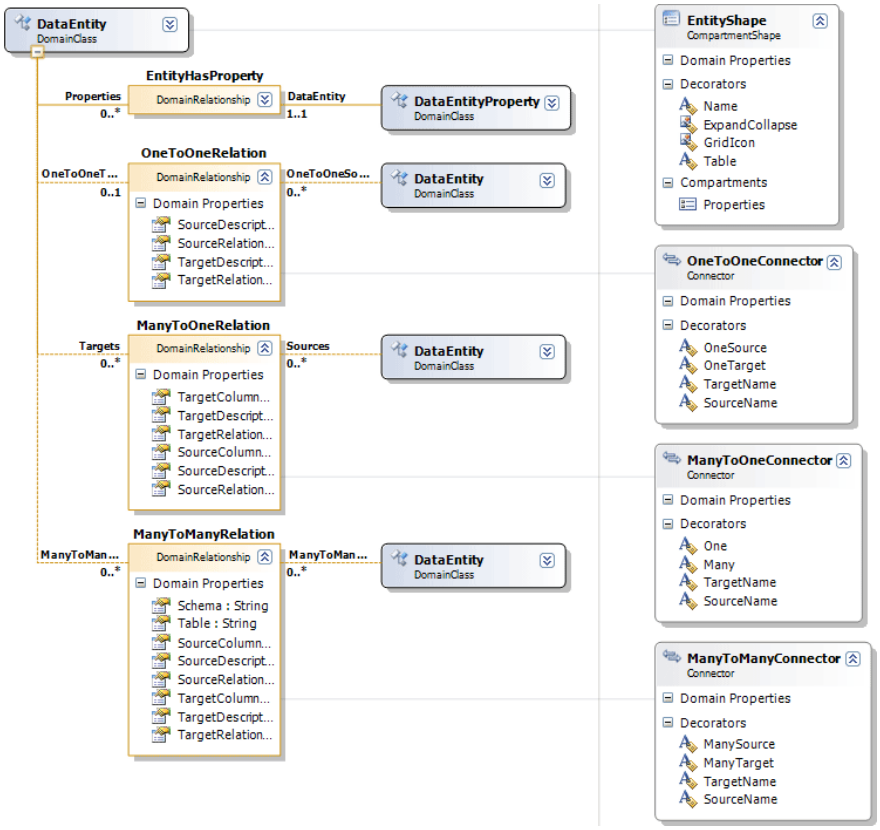


Figure 4.6: The abstract and concrete syntax elements get related through diagram element maps.

Despite the must to specify shapes and connectors to create a working DSL, they remain helper objects and are not very interesting conceptually. We will therefore not go into too much details regarding this, but we will touch decorators briefly.

Besides properties of the shapes or connectors, decorators may be defined for these diagram elements. There are several different decorators. In the current DSL Tools release, there are two relevant ones: textual and graphical. These decorators allow the designer of the DSL to add labels or icons to the diagram elements, exposing property values or state, optionally guarded by conditionals. Icons are currently limited to static, pre-defined images, but textual labels can show any property desirable. To do that, for each diagram element map, a decorator map must be specified using a path language[†].

The same goes for the composition of so called compartment shapes. These shapes are useful when a model contains a lot of elements. To keep all the elements together in a neat fashion, one can use compartment shapes, which poses a grouped, textual view of the domain classes. In our case, this is desirable to display the many table–column relations.

[†]Documentation lacks completely currently. For unofficial documentation, see `http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=464793&SiteID=1`—visited April 9th, 2007.

For such a compartment shape, the same path language is used to address the references. Figure 4.7 provides an example.

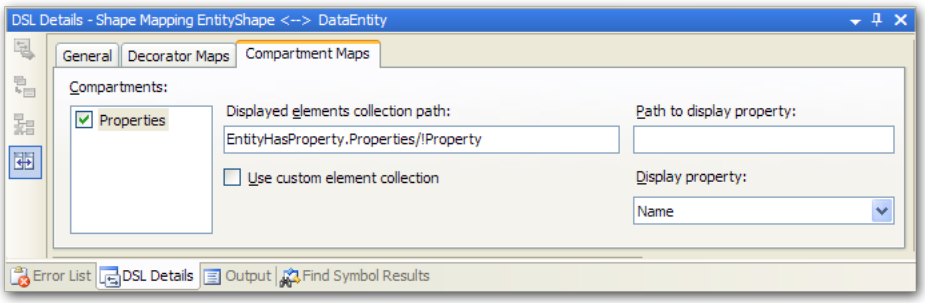


Figure 4.7: The path languages allows for provision of decorator or compartment mappings.

4.2.3 Semantics

DSL Tools uses the given definition of the visual language to generate a framework for the access to the in-memory data representations of the language elements. For each DSL solution, two Visual Studio projects are created, `Dsl` and `DslPackage`, each with its own output. In the project `Dsl`, all C# classes are generated that have to do with the domain framework (domain classes, relations, etc.), while the `DslPackage` project is created for the generation of a plugin for Visual Studio, to provide any programmer with DSL modeling support.

Thus, the most important project by far is `Dsl` and to that also belongs the language definition model file `DslDefinition.dsl`, in which the full DSL (abstract and concrete syntax) is specified. The generation of the DSL framework takes places by the Microsoft T4 text templating engine. That mechanism parses a plain text template file, in which C# code may be provided within template markers (`<# ... #>`). That code is evaluated and the markers will be replaced by the outcome of that evaluation.

Microsoft uses that method to generate C# code. In the directory `GeneratedCode`, numerous text templates (`*.tt`) are to be found that are provided by the creation of a new DSL project by default. These files use the data that is specified in the DSL definition file. This shows from the `requires` attribute of the `Dsl` directive (see Listing 4.1 below).

```
1 <#@ include file="Dsl\DomainModelCodeGenerator.tt" #>
2 <#@ Dsl processor="DslDirectiveProcessor"
3                             requires="fileName='..\DslDefinition.dsl'" #>
```

Listing 4.1: Content of a typical file in the `GeneratedCode` directory.

The `include` directive refers to a system-wide file that is distributed with the Visual Studio SDK and which contains the full-blown template code. A snippet of it is visible in Listing 4.2 on the next page.

From this snippet, it clearly shows that flat C# template code gets transformed by replacing the template markers with the the evaluation results of the template logic defined within.

```
1  [DslModeling::DomainObjectId(" <#= dm.Id.ToString("D") #> ")]
2  <#= CodeGenerationUtilities.GetTypeAccessModifier(dm.AccessModifier) #>
3  partial class <#= dm.Name #>DomainModel : DslModeling::DomainModel
4  {
5      #region Constructor, domain model Id
6
7      /// <summary>
8      /// <#= dm.Name #>DomainModel domain model Id.
9      /// </summary>
10     public static readonly global::System.Guid DomainModelId =
11                         <#= CodeGenerationUtilities.GetGuid(dm.Id) #>;
12     ...
13
14     #endregion
15 }
```

Listing 4.2: Snippet from the template file `DomainModelCodeGenerator.tt` that ships with the Visual Studio SDK.

Note that the template logic is provided in C#, but the output of the template after transformation is also intended to be C#.

To generate a full framework for the use of the DSL, Microsoft defines sixteen such code template files in the directory `GeneratedCode`, of which each implements its own part of the functionality. They can not exists apart from each other, but form the framework together. In the template files `DomainModel.tt`, `DomainClasses.tt` and `DomainRelationships.tt`, C# classes are defined which are actually OO-representations of the defined language elements. For each domain class, a C# class is generated with the same names, properties and relations, of which a large part of default implementation code is generated. These classes form the source for accessing model elements defined with this DSL.

For each incoming or outgoing relation, a property is added to the which abstracts away from the relation proxy and thereby implements relations implicitly. An example of this is provided in Listing 4.3 on the facing page. In lines 3–6, domain properties are visible, while in lines 8–18 the relation proxies can be found. These properties ease traversal of relationships.

The relation classes themselves are not accessible through this way. Each relationship can be addressed using the class determined for that. An example is provided in Listing 4.4 on the next page.

Furthermore, numerous non-public helper classes get generated:

- `Serializers`, for each domain class and domain relationship, which is responsible for the serialisation of living object instances and the instantiating of serialised objects;
- `PropertyHandlers` and `RelationBuilders`, which are defined as helper types for respectively each domain property and domain relationship. These help the assignment of values to properties and the creation of relations, using visual model actions;
- `Shapes` and `Connectors`. As already discussed, these classes provide the concrete syn-

```
public partial class DataEntity : NamedObject
{
    // Domain properties
    public string Schema { get; set; }
    public string Table { get; set; }
    public boolean IsPrimaryEntity { get; set; }

    // Embedded relationships
    public virtual LinkedElementCollection<DataEntityProperty>
                                                    Properties { get; }

    // Referencing relationships
    public virtual DataEntity OneToOneTarget { get; }
    public virtual LinkedElementCollection<DataEntity> OneToOneSources { get; }
    public virtual LinkedElementCollection<DataEntity> Targets { get; }
    public virtual LinkedElementCollection<DataEntity> Sources { get; }
    public virtual LinkedElementCollection<DataEntity> ManyToManyTargets { get; }
    public virtual LinkedElementCollection<DataEntity> ManyToManySources { get; }
}
```

Listing 4.3: The skeleton of the generated `DataEntity` class.

```
public partial class EntityHasProperty : DslModeling::ElementLink
{
    // There are no domain properties defined for EntityHasProperty

    // Role accessors
    public virtual DataEntity DataEntity { get; set; }
    public virtual DataEntityProperty Property { get; set; }
}
```

Listing 4.4: The skeleton of the generated `EntityHasProperty` class.

tax objects for the domain classes and domain relationships. Here is also the place where corresponding `Decorators` and `Serializers` are defined;

- ▷ `Mappings`, classes for mappings between the abstract and concrete syntax elements;
- ▷ Domain enumerations, for self-defined enumeration types usable in model objects.

The `DslPackage` Project

This project is created by Visual Studio when a new DSL solution is created and normally does not require any changes to use it. It facilitates the distribution of the DSL without being obtrusive in the development process of the DSL in the `Dsl` project. The implementation in a separate project is a convenient choice of Microsoft and makes that we are not confronted with uninteresting implementation details regarding this.

4.3 Generating Code

We now have a good indication of how the visual syntax of the language works. Now, we will take a look at what we can do with it. The models that can be constructed with the language just created are in-memory object models that may be transformed. These transformations can be moulded in any desirable fashion—programmatically, the whole model can be traversed.

There are two different approaches to realise the code generation process, each with their own advantages and disadvantages. We will discuss these in detail below.

4.3.1 Text Templating

To generate code from models that we that we are able to create with the DSL, the sole definition of the DSL is all it takes. From the language definition the C# class files are generated and after compilation, the result is a Visual Studio plug-in. The generation of code may then occur in the Visual Studio project that *uses* the DSL.

To understand that, we must temporarily take a step back in the development process. We assume that we have compiled our new DSL and installed the resulting plug-in in Visual Studio, so we are able to use the DSL. When we now create a new project, we can insert a new item kind: a CRUDE model file (`*.crud`). Visual Studio then allows for the creation of a domain-specific model through a graphical model editor. Subsequently, we may use T4 text templating to specify how code (or other textual artefacts) must be generated from the model. This process is in fact similar to how the DSL definition itself is constructed. In other words: the DSL designer itself is a metalanguage, a meta-DSL.

Figure 4.5 shows the content of the text template. In line 3, the extension of the output file is specified. Each file to be generated therefore is limited to a single output file—for each desired output a text template must be specified. In this example we generate a plain text file, but nothing prevents the user of a DSL to generate more complex output (like C#-code or XML).

The code fragment should be self-explanatory. Its output is visible in Figure 4.6. Something that clearly stands out is the differing use of the relation proxy classes. In lines 14–25

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingText-
                                                    Transformation" debug="true"#>
<#@ output extension=".txt" #>
<#@ Crude processor="CrudeDirectiveProcessor"
                                              requires="fileName='Model.crud'" #>

Generated material. Generating code in C#.

This example text template displays all the existing many-to-one
relationships defined in the model file 'Model.crud'.  There is a
simple and a detailed report, the latter requiring explicit proxy
access through the ManyToOneRelation relationship proxy class.

Data entity relations (simple):
<#
    foreach (DataEntity source in this.CrudeModel.DataEntities)
    {
        // Simply display the data entities that are referenced in
        // many-to-one relationships
        foreach (DataEntity target in source.Targets) {
#>
            - <#= source.Name #>  --->  <#= target.Name #>
<#      }
    }
#>

Data entity relations (more detail):
<#
    foreach (DataEntity source in this.CrudeModel.DataEntities)
    {
        // Display relation information, too
        foreach (ManyToOneRelation relation in
                                ManyToOneRelation.GetLinksToTargets(source)) {
            // Find the source entity's primary key
            string primaryColumn = "";
            foreach (DataEntityProperty prop in relation.Source.Properties) {
                if (prop.Primary) {
                    primaryColumn = prop.Name;
                    break;
                }
            }
#>
            - <#= relation.Source.Name #>.<#= primaryColumn #>  --->
              <#= relation.Target.Name #>.<#= relation.TargetColumnKey #>
<#      }
    }
#>
```

Listing 4.5: An example text template generating output from the model by transformation.

```
1
2  Generated material. Generating code in C#.
3
4  This example text template displays all the existing many-to-one
5  relationships defined in the model file 'Model.crud'.  There is a
6  simple and a detailed report, the latter requiring explicit proxy
7  access through the ManyToOneRelation relationship proxy class.
8
9  Data entity relations (simple):
10             - B  --->  A
11             - D  --->  A
12             - D  --->  C
13             - E  --->  A
14             - F  --->  B
15
16 Data entity relations (more detail):
17             - B.A_name  --->  A.name
18             - D.id_A  --->  A.id
19             - D.id_C  --->  C.id
20             - E.id_A  --->  A.id
21             - F.id_B  --->  B.id
```

Listing 4.6: The output of running the text template against the CRUDE model file.

the implicit relation proxy framework is used to access the connected targets through the property `Targets`. That way, the target of the domain relationship can be accessed, but no information on how that target is connected is available, because that information is a property of the domain relationship itself.

If we want to generate code based on those domain relationship properties, we must explicitly address the relationship. This happens in the example in lines 27–47. In the set

```
ManyToOneRelation.GetLinksToTargets(source))
```

we find all outgoing many-to-one relationships of the data entity `source`. It should be clear that sometimes the first, and sometimes the second approach is desirable.

4.3.2 A Custom Code Generator

What happened in the previously discussed approach was that objects from a concrete CRUDE model instance were used to generate code within the very same project[†]. Although this can be a powerful abstraction, in genereal it is not profitable to do so. After all, the knowledge of the DSL syntax must be available to the *user* of the DSL. This turns the DSL itself into merely a vehicle of syntax that carries the relations between model objects, but nothing more than that—there resides no knowledge in the DSL itself. Semantics are given only defined afterwards, when defining the template transformations.

Moreover, it seems illogical an approach within the software factories philosophy: generating product-lines would mean that for each such product both a model (using DSL syntax) and all semantics (using text template transformations) should be provided by the application

[†]Although in the given exmaple, we did not generate code, but plain text files, for simplicity's sake.

programmer. Such a model can merely function as a central source to abstract over application specific commonalities (like a standard routine library), but will in practice involve much more work than simply writing the software manually, without a DSL.

What we want is to put the required semantics in the DSL itself—only then we will achieve added value. We only have to write code once, namely at DSL design time. After all, *what* we want to generate will always be the same (i.e. an application GUI).

When creating the DSL in the metalanguage, the approach followed by Microsoft is the former: the valuable knowledge is "hidden" in the text templates that are added to a new DSL project. Where the DSL team of Microsoft choses to assume that these text templates are added to the project, they could as well have simply provided a code generator in the DSL—the required knowledge would have been zero and no explicit conventions would be necessary. More on this is described in the next sections.

Introduction

In the CRUDE DSL plug-in, an alternative, custom code generator has been built in, which is capable of fully translating any model to an application GUI. In the `DslPackage` project, the text template file `Package.tt` is responsible for the generation of a class `CrudePackage`, which is the repository of services to generate. Through the `ProvideCodeGenerator` attribute, we may hook up our own code generator.

```
1 [VSHost::ProvideCodeGenerator(typeof(CustomTool::CrudeTemplatedCodeGenerator),
2     "CrudeCodeGenerator", "Sogyo Crude code generator for .crud files", true,
3     ProjectSystem = VSHost::ProvideCodeGeneratorAttribute.CSharpProjectGuid)]
4 internal sealed partial class <#= dslName #>Package : <#= dslName #>PackageBase
5 {
6 }
```

The class `CrudeTemplatedCodeGenerator` is also part of the `DslPackage` and is defined in Listing 4.7. It is only a workaround for the currently missing full support for custom code generators in the DSL Tools framework.

Considered a black box, the overridden method `GenerateCode` is responsible for swallowing the filename and the content of a text template and yielding a `byte[]` as its output. However, the workaround consists of tricking this method, replacing the actual content of the template file with a pre-defined file `CrudeReport`, which is attached to the DSL assembly as a resource. This file's content is shown in Listing 4.8.

The workaround is necessary for the manual replacement of the template-variables `%VARS%`, to use the correct filenames, extensions and namespaces from the model definition. At this point in code, there is no notion of those concepts otherwise. Good support for custom template should solve these problems. Microsoft developers have unofficially stated that attention is being paid to this in upcoming versions of DSL Tools[†].

All magic now takes place in line 9, where the `CrudeCodeGenerator` is being invoked and its output fully replaces the whole template. That generator is included in the `Dsl` project. All

[†] `http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=1258205&SiteID=1`—visited April 10th, 2007.

```
[Guid("2E39FF45-B117-4126-9073-7DDCBEC3E113")]
internal class CrudeTemplatedCodeGenerator : TemplatedCodeGenerator
{
    protected override byte[] GenerateCode(string filename, string content)
    {
        ResourceManager manager =
            new ResourceManager("Sogyo.Crude.VSPackage",
                                typeof (CrudeTemplatedCodeGenerator).Assembly);
        FileInfo fi = new FileInfo(filename);
        content = manager.GetObject("CrudeReport").ToString();
        content = content.Replace("%MODELFILE%", fi.Name);
        content = content.Replace("%NAMESPACE%", FileNameSpace);
        content = content.Replace("%EXT%", "cs");

        byte[] data = base.GenerateCode(filename, content);
        return data;
    }
}
```

Listing 4.7: The code generator based on T4's `TemplatedCodeGenerator`.

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.Modeling-
                                                TextTransformation" debug="true" #>
<#@ output extension=".%EXT%" #>
<#@ Crude processor="CrudeDirectiveProcessor" requires="fileName='%MODELFILE%'" #>
<#@ import namespace="Sogyo.Crude.Common" #>
<#
    CrudeCodeGenerator generator = new CrudeCodeGenerator(this.CrudeModel,
        "%NAMESPACE%");
    this.Write(generator.Generate());
#>
```

Listing 4.8: The mock text template that invokes the real custom code generator.

of the above is necessary to enable the implementation of such custom code generator. We shall focus on the code generation itself, using the class `CrudeCodeGenerator`.

Generation

For the generation of code, instead of using text templates, one can chose to fully generate the text by a dedicated class. For CRUDE, we used a helper class that generates actual C# code. We exploit the Microsoft CodeDOM[†], a framework for the construction of source code using a *document object model* (DOM).

The framework assists in writing code constructs descriptively. This implies, among more, that we can generate only syntactically correct statements, following the C# grammar. From the DOM structure to be created, we can subsequently generate a full-blown source code implementation through a CodeDOM `CodeProvider`. Currently, there exist such code providers for C# and Visual Basic.NET, but in essence any code provider may be implemented by third party vendors for any .NET programming language.

Using CodeDOM, code is not written directly, but the structure of the code is built in the DOM model. This implies that a meta-description of the final source code should be constructed. Compare the snippets from Table 4.9. Here, the descriptive nature of using CodeDOM clearly shows.

```
CodeMemberMethod handler = new CodeMemberMethod();
handler.Name = name;
handler.Attributes = MemberAttributes.Private;
handler.Parameters.AddRange(
    new deParameterDeclarationExpression {
        new CodeParameterDeclarationExpression(
            new CodeTypeReference(typeof(Object)),
            "sender"),
        new CodeParameterDeclarationExpression(new
            CodeTypeReference(typeof(EventArgs)),
            "e")
    }
);
handler.Statement.AddRange(
    // CodeStatementCollection goes here
);
```

a. Generation via CodeDOM

```
private void <#=name#> (object sender, System.EventArgs e)
{
    // Statements here
}
```

or

```
Private Sub <#=name#> (ByVal sender As Object,
    ByVal e As System.EventArgs)
    ' Statements here

End Sub
```

b. Text templating

Listing 4.9: A readability comparison.

Nonetheless, we may also see how a relatively small code fragment already takes an extensive amount of code lines to express the structure of the code. Grasping the fragment therefore costs more effort than the template-based variants, where the code structure is of no importance, and therefore no limitation.

Despite it, this might be a little distorted view of reality. The text templating approach ignores the importance of generating semantic or context-sensitive expressions. Invalid values can be entered and incorrect code can be the result. The form and valid point of occurrence of language constructs must be respected. A variable name, for example, should be rejected if it is a language keyword, contains illegal characters, is too long, or occurs at an invalid place in code.

[†] http://msdn2.microsoft.com/en-us/library/650ax5cx(vs.80).aspx—visited April 11th, 2007.

Even worse, code entered can influence the meaning of its environment. For example, fill in the value `"dummy() //"` for the variable `name` from Listing 4.9b and the result is not only a different method name, but also a different signature. This code is never found erroneous, since it will even compile as if it were correct. On the contrary, using the CodeDOM framework, all of these issues are avoided. Any constructs violating the language's grammatical rules are directly reported through exceptions during the generation process.

The large own responsibility that comes with generating code using text templates is that those templates are always a piece of code for a specific language. This means that, for each potential target language, a different template file must be written—and those files should mutually be kept in sync. After all, the templates are different dialects of the same code body. Modifications should be made in all such templates, which is rather fragile and has very negative consequences for maintainability.

Although distributing the DSL might not be relevant for Sogyo—the DSL will mainly be used for internal purposes—generation using the CodeDOM framework will lead to a scalability increase (by being language independent) an a must if (syntactical) code correctness is specifically desirable.

The fundamental disadvantage of the custom code generator is the verbosity involved with the description of the code structure, and in particular the decreasing readability that results from it. In practice, this can be quite well dealt with by annotating the generation process with exemplary code intentions in comments. Of course, the risk of outdated comments will remain an issue then.

In Table 4.1 below, a brief comparison between the two methods of generation is provided.

Aspect	Text Templating	Custom Code Generator
Soundness	✗ No control over correctness. Output is essentially meaningless, but may be valid compiler input if used as intended. Incorrect output can not be detected.	✓ Output syntactically sound due to typed CodeDOM classes that force correct structure of code.
Completeness	✓ No limitations due to free text form. Any language construction is expressible.	✗ CodeDOM might pose limits on code structure. For example, the unary boolean inverse operator is currently not provided.
Multiple output files	✗ Only by making multiple templates, one for each output. Code duplication may lay at risk.	✗ Only by work around.
Readability	✓ Pretty straightforwardly readable. May be misleading, though.	✗ Very verbose. Hardly readable without guiding comments.
Logic resides	✗ At application development time (DSL usage time).	✓ At DSL construction time.
Extensibility	✓ Templates can be extended for specific purposes within specific application boundaries (*ad hoc*).	✓ Yes, structurally after recompilation and distribution.

Table 4.1: Comparison of the two generation methods.

4.3.3 Towards Generation of a GUI

In the previous discussion, we have seen how code can be generated. Now, we will focus on the generation of the GUI from a specific application model. It is convenient and appropriate to exclude any functionality from the generation process that essentially does not belong to a concrete application model to start with. This functionality will be statically implemented in a class library.

To do that, CRUDE generates GUI classes for the main window and the various edit windows. Figure 4.8 shows the foremost classes that are generated in their hierarchical structure. Technically, one can look at this as a three layer model. Layer 1 contains all statically implemented functionality which is the same for all CRUD application and should thus always be available. Examples hereof are the interface elements—an empty main window and empty dialog windows. In layer 2, these elements are extended with generated code from the given model. Hence, this layer facilitates the generation of multiple applications, varying within the variabilities that are enabled by the model. In layer 3, finally, manual extensions can be made in the generated program code.

Note that this figure shows classes from the re-implementation in layers 2 and 3, as will be discussed in Chapter 5.

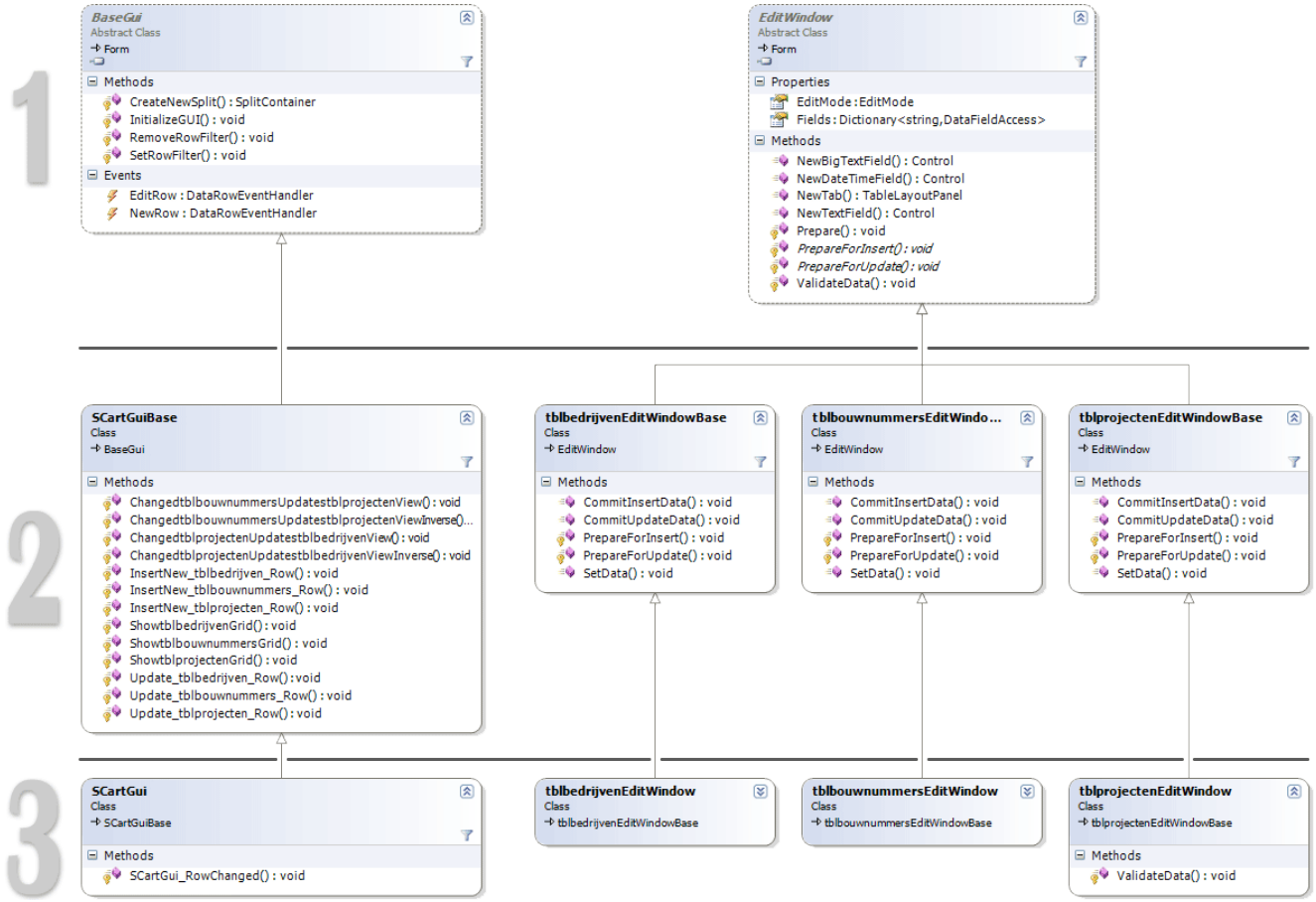


Figure 4.8: An overview of the most important classes generated by the code generator.

The generated functionality is partly enclosed in the top-level classes `BaseGui` and `EditWindow` (implemented statically in the library `Sogyo.Crude.Gui.dll`, which is dis-

tributed along with the DSL) and partly in the classes `SCartGuiBase` (for the main window) and `*EditWindowBase` (for each edit window). The classes on the lowest level of the hierarchy are empty inheritors of classes that provide the full functionality. This is required for facilitating possible extension of the applications (see Section 4.4.1 for a discussion). We shall discuss the exact functionality of these classes in the following sections.

The `BaseGui` Class

The static implementation of `BaseGui`, from `Sogyo.Crude.Gui.dll` contains all basic functionality that should be available, but will not be generated by the code generator explicitly. Instead, it provides a framework that the generated code may *use*.

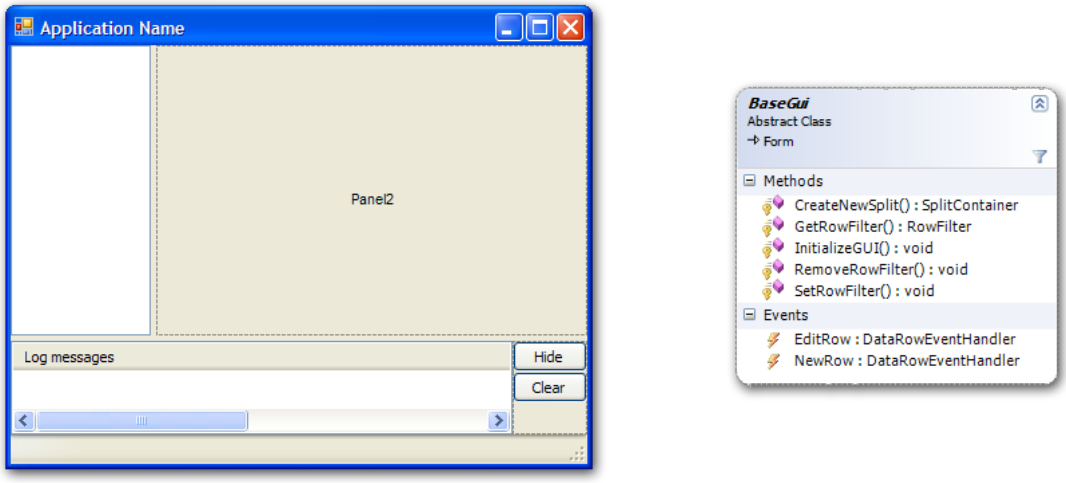


Figure 4.9: The base GUI in the Visual Studio form editor on the left. On the right, the class diagram shows how the base GUI exposes several functions to manipulate the interface.

An accompanying advantage is that the base of the application may be graphically designed using Visual Studio's form designer, see Figure 4.9. In this statical base implementation, a number of methods have been implemented:

- `CreateNewSplit()`:
  Used to split the screen in two in a specific GUI container, horizontally or vertically, create two new containers. This is used to create relationship chains on screen in the interface;
- `InitializeGUI()`:
  Used to support initialisation in case double-derived support is not desirable, see also Section 4.4.3;
- `SetRowFilter()`, `RemoveRowFilter()`:
  Used to define "row filters", essentially forming the internal lookup system for dynamically setting the conditionals for each data entity relationship to update related "child tables" upon row selection changes. This static functionality is used in the `Show`*`Entity`*`Grid` methods;
- `EditRow`, `NewRow` (events):
  Used as the place where events regarding the insertion and/or updating of table data can be hooked up. See also Section 4.4.3.

These abstract base class is inherited by `SCartBaseGui` (a specific form generated from the SCart model as will be discussed in Chapter 5). That way, statically implemented functionality and dynamically generated functionality can be combined, but at the same time be physically separated, which improves maintainability. The generated methods in `SCartBaseGui` are the following (method names with italic *Entity*'s are available for each data entity).

- `Show`*`Entity`*`Grid()`:
  These create the grids with the desired columns, execute the query that fills the grid, register the event handlers that are invoked when the grid selection changes (to update relational data recursively) and when a row gets double-clicked (for editing);

- `Changed`*`Entity`*`Updates`*`Entity`*`View()` and
  `Changed`*`Entity`*`Updates`*`Entity`*`ViewInverse()`:
  These are the event handlers that contain logic to respectively reflect the *-to-one and the *-to-many relations between the tables correctly in the interface. The difference between both lies mainly in the look-up style (recall Figure 3.7 on page 40);

- `InsertNew_`*`Entity`*`_Row()` and
  `Update_`*`Entity`*`_Row()`:
  These methods are both responsible for the creation of the according `EditWindow` and the passing-through of the correct data. They differ in implementation mainly in how they provide the data source to use and how the action results get committed to the database. Both cases require a different approach.

Figure 4.8 only provides these methods since they are most important. In practice, the re-implementation contains a lot more methods. For more information, we refer to the technical appendix.

The `EditWindow` Class

The implementation of the `EditWindow` hierarchy resembles the hierarchy of the `BaseGui` (see Figure 4.10 on the next page).

The static implementation of the `EditWindow`, also from `Sogyo.Crude.Gui.dll`, contains the following properties and methods:

- `Fields` (property)—a read-only dictionary for abstract access to the field data in the form, through the wrapper class `DataFieldAccess` (see Figure 4.11 on the following page).

  This class contains the properties `IsNull`, `Value` and `ValueOrNull`, which provides abstract access to the filled in field data from the edit form from outside the `EditWindowBase` classes themselves (i.e. from `SCartGui`) without technical implementation details of form controls;

- `NewTab()`:
  For the creation of new tab pages;

- `NewTextField()`, `NewBigTextField()`, `NewDateTimeField()`:
  Methods for the creation of new field types. Currently, only text fields and date/time fields are supported (see the discussion on page 79).

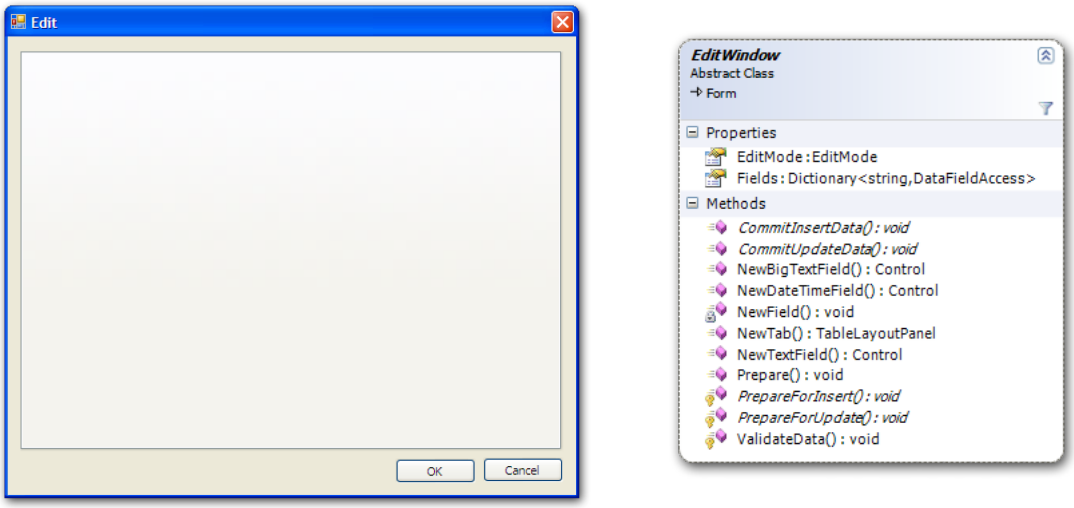


Figure 4.10: The edit window GUI in the Visual Studio form editor on the left. On the right, the class diagram shows how the edit window GUI exposes several functions to manipulate the interface.

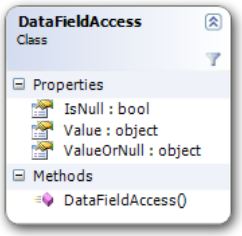


Figure 4.11: The `DataAccessField` wrapper.

This abstract class is inherited by the generated classes `*EditWindowBase`. The generated methods in the `EditWindowBase` class are:

- `CommitInsertData()`, `CommitUpdateData()`:
  Methods that commit the edited data, after validation, back to the typed `DataRow` that is given to the form, so that it can be written back to the database. The differing implementation of both methods lies in which fields they process (these may differ for insert or edit actions, as we know);

- `Prepare()`:
  Method that, depending on the edit mode (`Insert` or `Update`) call the accompanying method;

- `PrepareForInsert()`, `PrepareForUpdate()`:
  Abstract methods for the preparation of the dialog window by creating edit controls to edit the data entity properties for the given data entity. The difference in implementation lies in the specific fields that they handle;

- `ValidateData()`:
  Virtual method implementation for the validation of entered field data. This method can be extended with extra data checks by overriding the implementation in inheriting classes. By default, the generated implementation will check all entered values on

their validity within the column type. This is useful because a text box can be used to communicate an integer value with. In that case, only decimal digits are regarded valid inputs of the text box. Other values are rejected.

Apart from that functionality, for each data entity such an `EditWindow` class gets generated. In it, the following methods are contained:

- `CommitInsertData()`, `CommitUpdateData()`:
  Concrete implementations of the abstract methods from `EditWindow`;
- `PrepareForInsert()`, `PrepareForUpdate()`:
  Concrete implementations of the abstract methods from `EditWindow`;
- `SetData()`:
  Used to pass a reference to the typed `DataRow` to the `EditWindow`. Ideally, the `SetData` method should be declared in the base class as abstract method, analogous to the `Commit*Data` and `PrepareFor*` methods. However, this can not be implemented as such by overriding. This is because the signature of the method differs per class. Each `row` parameter is in fact a typed `DataRow`. Hence, there exists no signature that can be declared in an abstract base method.

```
1 // From 'tblbedrijvenEditWindowBase'
2 public virtual void SetData(tblbedrijvenRow row) { ... }
3
4 // From 'tblprojectenEditWindowBase'
5 public virtual void SetData(tblprojectenRow row) { ... }
```

  At the utmost, we can generate a `SetData` method in each class `EditWindowBase` and assume (safely) that this method exists. Despite the necessity of the method to assure the correct function of `EditWindow`, we can not already enforce this at layer 1.

In Figure 4.8 only the generated classes for the three most important data entities are included. In practice, the generated code for the re-implementation contains much more `EditWindow` classes.

4.4 Extension

An explicit wish regarding the use of DSL is the possibility to build in custom variations in the generated product. A generated implementation is never the intended result and changes should be possible to make. In this section, we will illustrate what possibilities for extension the code base provides, what is necessary to make such extensions and how much effort that takes.

We already saw in the previous discussion that we are capable of indeed generating a GUI from the model and that this takes minor effort. The implementation of a DSL is, because of its domain-specific nature, per definition soaked with design decisions—it provides one big refinement for the high-level model. When alternative implementations are desirable, differences in concrete implementations can be modelled as variabilities.

However, sometimes that does not suffice. It might be desirable to merely use the generated application as a starting point for further development. In that case, we speak of extension. Where at the implementation of the code generator we are free to make design decisions and chose implementations, here we are facing limitations that are strongly dependent on the extent to which extensions are accounted for during DSL development.

A major influence on extendibility is result of the design decision to either generate a framework that can be used by the DSL user, or that the framework itself has control over execution and is by itself a functional program. CRUDE is an example of the latter.

The available extension mechanisms provide by the current CRUDE implementation will be clarified below.

4.4.1 Double Derived Classes

Since the release of .NET 2.0, it is possible to use so called partial classes. Partial classes are a physical division of a class' definition, for example into multiple files. Together, the partial class definitions form a logical class, as if they were implemented in a single class definition.

This functionality is extensively used in the DSL Tools framework. Generated classes are typically implemented by partial classes, so that a user can define his own partial class extensions, adding functionality.

These opportunities for extension manifest at class member level. New methods and properties may be provided in partial classes, but logic of existing methods can not be modified. This is a problem in particular with the class' constructor. When the constructor can not be modified, classes can not be correctly initialised. Here, we must fall back on ye olde inheritance model, in which re-implementation of inheriting classes is not an issue.

However, the real power lies in the combination. Microsoft calls this the "double derived" construct. The pattern is a (generated) base class `B`, of which the inheriting class `A` is a partial class with an empty implementation. In code, this is shown in Listing 4.10 on the next page.

Consider this code as generated by the DSL. Given that code, as user of the DSL, we may define our own partial class implementation `A` in a separate file, in which we can add methods (because of the class being a partial definition), a constructor (because of a lacking constructor for class `A`) and override existing methods (by way of inheriting virtual methods)[†].

Please note that this construct implies that a constructor can be provided in only one partial class definition, even if we make a whole variety of extensions in multiple files of class `A`. If a strict separation of concerns (including separate initialisation logic) is regarded necessary, one could consider stacking multiple double derived constructs, one for each extension requiring initialisation logic.

Another implication of double derived classes is that the use for private variables will be completely wiped out. Using this technique, protected members are the most stringent level of accessibility.

[†]Note that we must realise that definitions made in the base must have accessibility levels of protected or public in order to address them in deriving classes. Moreover, they must be marked virtual if overriding is desirable. In other words: the base class should be aware of the fact that it is used in a double derived construct.

```
public class B
{
    protected int m_myprop;

    public B()
    {
        // Implementation goes here...
        this.m_myprop = 0;
    }

    protected virtual void Method(int x)
    {
        // Do some settings...
        this.m_myprop = x;
        SomeOtherMethod(x);
    }
}

public partial class A : B
{
}
```

Listing 4.10: The double derived construct in code.

In CRUDE, the class `CrudeModel` provides a property `GeneratesDoubleDerived`. With it, the user of the DSL may specify whether the code generator must generate a double derived construct or not.

4.4.2 Model Flags

A different way of extension, is by the incorporation of hooks in code where standard functionality does not need to be generated, because the end user offers an alternative implementation. Such can be modelled as a variability based on which functionality either is provided or left out. This approach is technically speaking identical to the way the property `GeneratesDoubleDerived` is implemented (see previous section).

This kind of extensions are also found in CRUDE, for example in the `GenerateTypedEvents` in the `DataEntity` class. A more in-depth discussion of that property is provided in Section 5.3.2.

4.4.3 Via Events

Sometimes, specific extensions may clearly be anticipated upon. An example is validation, which is different in each product, but the framework for providing and extending validation may pose a valuable one. When this is the case, one can decide to facilitate this extension through events. If in certain places in the generated code events get fired, users may define event handler to respond to such general events.

In CRUDE, two of such general events are implemented—`NewRow` and `EditRow`. These get fires just before the moment the entered data in the data set gets committed. The event makes it possible to change some of the event data last-minute, or even cancel the action.

These event handlers should somehow be hooked to the events, something that not occur by itself. Aftel all, the generated code owns the control loop, so there must *a priori* be agreed and decided upon a specific place in code to attach these events. This is possible in two ways:

1. The preferable approach is to reside to the use of double derived classes. In such case, the coupling of event handlers is easy, namely in the constructor:

```
public partial class SCartGUI
{
    public SCartGUI() : base()
    {
        // Hook up some custom application events
        NewRow += new DataRowEventHandler(NewRow_EventHandler);
        EditRow += new DataRowEventHandler(EditedRow_EventHandler);
    }

    ...
}
```

2. In cases where double derived classes are not used, the user of the DSL is not capable of gaining control over the framework's control loop—and that control is necessary to attach the event handlers to the event hooks. In those cases, the DSL might provide a dedicated empty virtual method implementation in `BaseGUI`, called directly after the class' construction. This is workaround for artificially providing an initialisation method without requiring the need for a constructor.

```
public abstract partial class BaseGui : Form
{
    public BaseGui()
    {
        // Initialization of BaseGui
        ...

        // Allow overridden classes to perform logic
        InitializeGUI();
    }

    /// <summary>
    /// This method can be overriden to hook up user events
    /// to this class in a partial class implementation.
    /// </summary>
    protected virtual void InitializeGUI()
    {
    }
}
```

   The final result is that in a partial class the `InitializeGUI` method may simply be overriden:

```
public partial class SCartGUI
{
    protected override void InitializeGUI()
    {
        base.InitializeGUI();
```

```
6
7           // Hook up some custom application events
8           NewRow += new DataRowEventHandler(NewRow_EventHandler);
9           EditRow += new DataRowEventHandler(EditedRow_EventHandler);
10      }
11
12      ...
13  }
```

4.4.4 Functionality Removal

Finally, the "removal" of logic is tricky. Modifications to generated code are not allowed, because during the next model change, all modifications are directly undone because all code is generated again. In contrast to the extension of code, removal of code is technically impossible. Partial classes may only result in "growing" classes.

Undesired generated logic can merely be "removed" by implementing a piece of logic through a model flag (see Section 4.4.2). Essentially, this comes down on an extension of the DSL itself, not a custom user extension. Design decisions (refinements) have been made in the DSL implementation, so the only way of "removing" this functionality is to change the refinements, i.e. implement the changes in the DSL itself.

The property `HasCustomEntryPoints` is an excellent example of that, see Section 5.3.4.

Summary

In this chapter, we have documented the design and the implementation of CRUDE, our software factory for the specification of CRUD applications.

Subsequently, we have defined factory inputs and outputs, and defined factory operations, all based on the product-line analysis from the previous chapter. In particular, we have paid attention to the core aspect of the software factory, the DSL. We have specifically described its abstract and concrete syntax. We provided alternative solutions to implement the semantics of the language and made a case for the use of a custom code generator.

Furthermore, the code generator's output architecture is described using a comprehensible three layer model. Finally, specific attention is paid to extensibility as an important practical measure and we have shown which extension mechanisms are provided by the factory.

In the next chapter, we will make a re-implementation of the original SCart application, to facilitate a comparison of the traditional and the software factory approach to the application's development. The factory's use is illustrated with that and extensions are provided to clarify how practical extensions can be made to such generated application.

5 Re-implementing SCart

Now that we have seen from the previous chapters how a software factory and domain-specific languages are created with the tooling that Microsoft offers, we will in this chapter make an actual implementation of an application from such a software factory. The application that will be created using this DSL will be a re-implementation of the SCart application (see Chapter 3).

Re-implementing SCart serves a triple purpose. First of all, a re-implementation of SCart shows that all important aspects from the original application may be generated by the CRUDE software factory. Secondly, it shows what effort it takes for such implementations to adapt various aspects of the generated software. Thirdly, it provides a solid base for quality comparison, which will be relevant for Chapter 7.

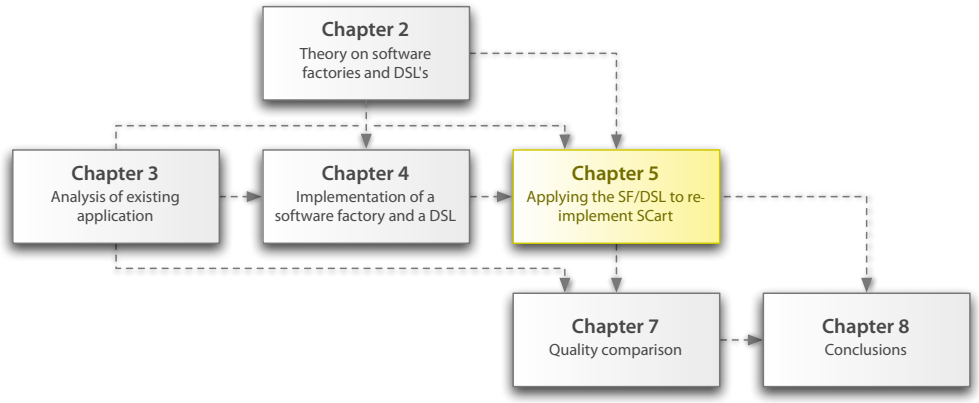


Figure 5.1: Chapter 5 in its context.

Specifically, in this chapter, we will define a large enough subset of SCart and actually implement that subset to show that DSL's might indeed be useful.

5.1 Subset Selection

The goal of the re-implementation of SCart is not to make a fully functional clone of the existing application, but to show that the application *can* be also generated from a DSL.

Functionality	SCart	Re-implementation
Database		
Entities (bedrijven, projecten, bouwnummers)	✓	✓
Entities (other)	✓	✗
Double primary keys	✓	✗
GUI elements		
Primary views (master grids)	✓	✓
Detail views (detail grids)	✓	✓
Edit data	✓	✓
Edit relationships	✓	✗
Primitive data type editing	✓	✓
Complex data type editing	✗	✗
Validation	✓	✗
CRUD-related		
Read from tables	✓	✓
Filtering	✓	✗
Visualisation of relationships	✓	✓
Update rows	✓	✓
Remove rows	✓	✗
Performance optimisation	✓	✗
Specific		
Actions	✓	✗
System preferences	✓	✗
Remote database access	✓	✗
Login / multi-user access	✓	✗
Primitive versioning	✓	✗

Table 5.1: Implemented functionality comparison.

To show this, we will iterate over SCart's functionality and for each part indicate how we have implemented it, or in case we did not implement it, argue why not and how such would be possible alternatively. Table 5.1 provides a quick overview.

5.1.1 Database

First and foremost, the size of the original SCart database is decreased. This is done to simplify the overview of the database. With less tables, we are still able to show our point. The smaller, less relevant, tables are simply left out and only 3 of the 6 main tables are used. We did take care that every relation type is at least used once in the design. The three main tables used (`bedrijven`, `projecten` and `bouwnummers`) form the central spindle of the application.

Some concession has been made with respect to the content of the remaining tables. SCart makes used of a custom made version control system, by storing a version number and a `isDeleted` bit for each table row (in each table). The current implementation is, although unnecessary, a composite primary key over the columns `ID` and `version`[†]. We have therefore decided to use only a single primary key column, `ID`.

The current implementation of CRUDE supports only primary keys over maximally one column. Each table now has exactly one column serving as a primary key column. This has the consequence that:

- ▷ versioning does not work—these columns are therefore left out in the database design;
- ▷ relationship roles over multiple columns are not supported currently.

5.1.2 Primary and Detail Views

The primary view implementation is vert straightforward. Column names can be chosen identically to the columns as they exist in SCart. The column order can be determined by the sorting of the data entity properties within the data entity in the model[††]. The data is collected through the `Fill` method of the according table adapter object. The typed `DataSet` provides access to the table adapter objects.

Filling the details grid based on the current master grid's selection also happens by invocation of the `Fill` method of the according table adapter, after which the correct rows are filtered. This essentially is an efficiency problem, because all table data is first collected, but a large part is never displayed. Instead of a query "at the gate", filtering takes place only late in the process.

This implementation is the result of the loose coupling between the `DataSet` and the CRUDE model. Because the `DataSet` is generated apart from CRUDE and CRUDE merely assumes its existence, a few assumptions have to be made. Apart from the use of CRUDE, as little

[†]Merely the `ID` column is used for identification purposes. The `version` column is used to facilitate optimistic locking [21]. The choice to use a composite primary key is not harmful, but also essentially unnecessary and therefore, quite unfortunate.

[††]DSL Tools currently provides support for sorting elements on the model surface, but not yet within compartment shapes. This is poor support of a product still immature. Internally, in the XML file, this impediment may be fixed manually, though.

information as possible should be required to use the system. An explicit requirement for CRUDE, thus, is the usage of Microsoft's data set generator, but only if that step is trivial.

So we assume the generator's output to be the most simple—obtained by walking through the wizard with default settings. In that case, the table adapters are provided with the methods `Fill`, `Insert`, `Update` and `Delete`. These functions are wrappers for the actual SQL queries (`Fill` is a wrapper for a standard full table `SELECT`). Microsoft offers users the possibility to provide custom queries with the data set generator. Typical use of that is the definition of parameterised queries to collect conditional data. Hence, there are three options, listed in ascending order of desirability:

- ▷ We may simply ignore custom query definitions. In that case, we can not use optimised queries. Instead, we collect all data and afterwards filter the rows to show based on a given predicate function;
- ▷ We provide the possibility to define custom queries manually and have a variability built-in in the model, specifying the name (and parameters) of the query to use instead of the default `Fill` function;
- ▷ We specifically generate all the custom queries necessary to provide efficient alternatives to the default `Fill` method. The generated code thereby extends the `DataSet`.

The first of these alternatives is the current implementation. In practice, the implementation is not a scalability issue yet—performance is acceptably well. The other alternatives must be considered in case performance needs to be increased. To quote Hoare: *"Premature optimisation is the root of all evil."*

5.1.3 Relations

The three types of relations that may exist between data entities are all supported through master grid–detail grid connections. Each relation is shown identically in the various detail grids. The relation type is of no importance to that, although another look-up is performed for *-to-one and *-to-many relations (recall Figure 3.7 on page 40).

Note that we differ from the way displays *-to-one relations. For SCart, some of these relations are represented as conceptual properties of the data entity. This is done by horizontally joining the related data with the master table and shown as one or more extra columns in the master grid. This gives a more complete view of the data, since related data of more than one row is visible at once.

Nonetheless, for simplicity's sake, CRUDE does not support that. After all, the goal of the re-implementation is not to make a fully functional clone of SCart, but to show that all data entities and their relations can be visualised.

5.1.4 Editing and Insertion

SCart supports its edit and insert actions through specific user-interface actions. To insert a row in a table, a dialog window is shown where initial row values may be entered. In SCart, all rows have a single primary key, so in this step a new unique key is automatically generated. These primary keys are always hidden in the interface, but serve as row addressers

in the background. With edit actions, nothing happens to the primary key—it can not be modified.

Both with edit and insert actions, different fields are categories and sorted based on their `Category` property (a freeform string representing the tab page name) and their defined order in the model. For edit and insert actions, the visible fields may differ, although mostly all fields will be visible in both actions.

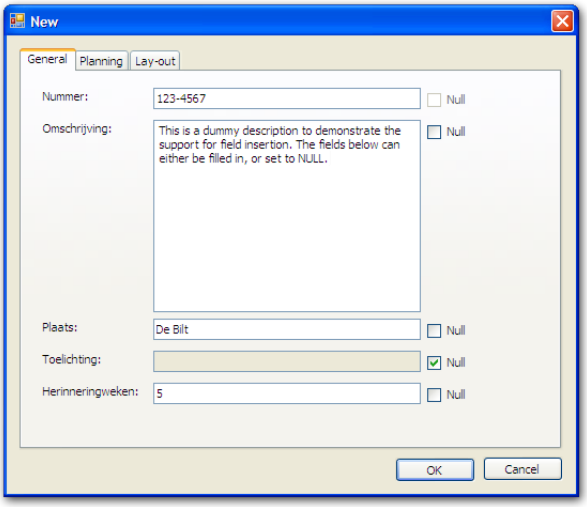


Figure 5.2: Editing project data through the `EditWindow` class.

For each data entity property, a field is created (see Figure 5.2 above). Each such field contains a control capable of editing the given data type. Behind the column a checkbox shows for each optional field to nullify a field's value. For each type column (`ColumnType`), a specific control is used to visualise and edit that type of data. Currently, the data types `String` and `StringClob` (large string object) are implemented using a `TextBox` control (in case of a `StringClob` field, a larger text box is used). For the data types `DateTime` and `Timestamp`, a `DateTimePicker` control is used. For all other data types edit controls may be easily defined (see Listings 5.1 and 5.2). Until that is done, the facilities of the `Convert` class from the .NET Framework is used to convert any kind of data to its string representation. A `TextBox` is then used to edit that data.

5.2 Application Generation

Building an application using CRUDE happens in four steps, that hardly require time, because all the effort and knowledge is put under the DSL's hood. The steps that subsequently must take place, are the following:

- ▷ The derivation of a model with data entities and data entity properties from an existing database structure, matching them with the tables and columns. This is facilitated in CRUDE using the drag & drop support from the Server Explorer[†]. This support improves the production process significantly, because a whole table structure can be converted

[†]This support is heritage of the open source tool ActiveWriter, of which CRUDE borrowed parts of its source code.

```
/// <summary>
/// Get a code reference to the Convert.* conversion method.
/// </summary>
private CodeMethodReferenceExpression GetConversionMethod(DataEntityProperty prop)
{
    string convertTo;
    switch (prop.ColumnType) {
        case NHibernateType.DateTime:
            convertTo = "ToDateTime"; break;

        case NHibernateType.Int32:
            convertTo = "ToInt32"; break;

        case NHibernateType.Boolean:
        case NHibernateType.YesNo:
            convertTo = "ToBoolean"; break;

        // TODO: Add cases to support other types

        default:
            convertTo = "ToString"; break;
    }
    CodeMethodReferenceExpression conversionMethod = new CodeMethodReferenceExpression(
        new CodeTypeReferenceExpression(typeof(Convert)),
        convertTo
    );
    return conversionMethod;
}
```

Listing 5.1: Simple implementation showing how you can add data type conversion routines. This code can be found in `Crude\Dsl\CodeGeneration\Helpers.cs`.

```
switch (prop.ColumnType)
{
    case NHibernateType.StringClob:
        fieldCreator = "NewBigTextField"; break;

    case NHibernateType.DateTime:
        fieldCreator = "NewDateTimeField";
        if (String.IsNullOrEmpty(prop.DefaultValue))
            defaultExpression = new CodeFieldReferenceExpression(
                            new CodeTypeReferenceExpression(typeof(DateTime)), "Today");
        else
            defaultExpression = new CodeMethodInvokeExpression(
                                        GetConversionMethod(prop), prop.DefaultValue);
        break;

    // TODO: Add cases to support more field creators and/or default values

    default:
        fieldCreator = "NewTextField"; break;
}
```

Listing 5.2: Simple implementation showing how you can add extend field creators (and default values for those) for specific column types. This code snippet is an excerpt from `Crude\Dsl\CodeGeneration\EditWindowGeneration.cs`.

into a domain model for CRUD applications. Especially since, as we have said before, CRUDE is "loosely coupled" to the database. Automation of this step replaces the tremendous amount of trivial manual work that would otherwise have been necessary to "copy" the domain model from the database structure.

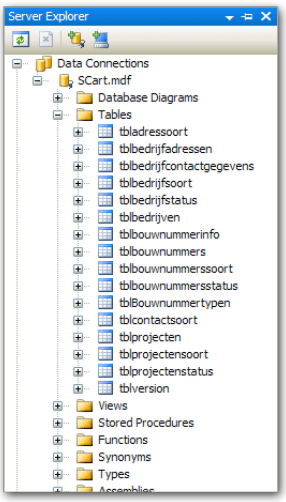


Figure 5.3: The tables as they exist in the given database.

When a series of tables is selected in Server Explorer (see Figure 5.3 above), the table information is read and for each table object a data entity is created and coupled to the original table. The same goes for the columns and the relationships between those.

- This from the database structure derived domain model is yet incomplete and must be extended with some additional information in order to generate the application. Titles of data entities and or specific properties may be changes in order to provide a more user-friendly experience for the end-user of the application. The usage of the DSL is nothing more than the editing of a large, visual configuration file. The result of the edition is shown in Figure 5.4 on the next page.

  Next to the modification of properties, structure-technical properties of the model can also be modified. In essence, this functionality is not very common, but CRUDE supports them explicitly nonetheless, since the need may arise that a model structure must be changed (for example when a change in database structure must be reflected in the model).

- Apart from creating the model, a typed dataset must be created, because CRUDE uses that functionality. As said before, this step is trivial with Microsoft's dataset generator. When the dataset is generated, the model should be informed by pointing the model variables `TypedDataSet` and `TypedDataSetNameSpace` to it.

- Finally, the only remaining issue is telling the Visual Studio project how the model file should be processed. This is done by pointing the property `CustomTool` of the model file to the CRUDE code generator, see Figure 5.5 on the following page.

Done all that, classes will be generated as visible in Figure 4.8 on page 65. The concrete implementation will be generated up to layer 2. In layer 3, we may make our own extensions, as we will discuss in the following section.

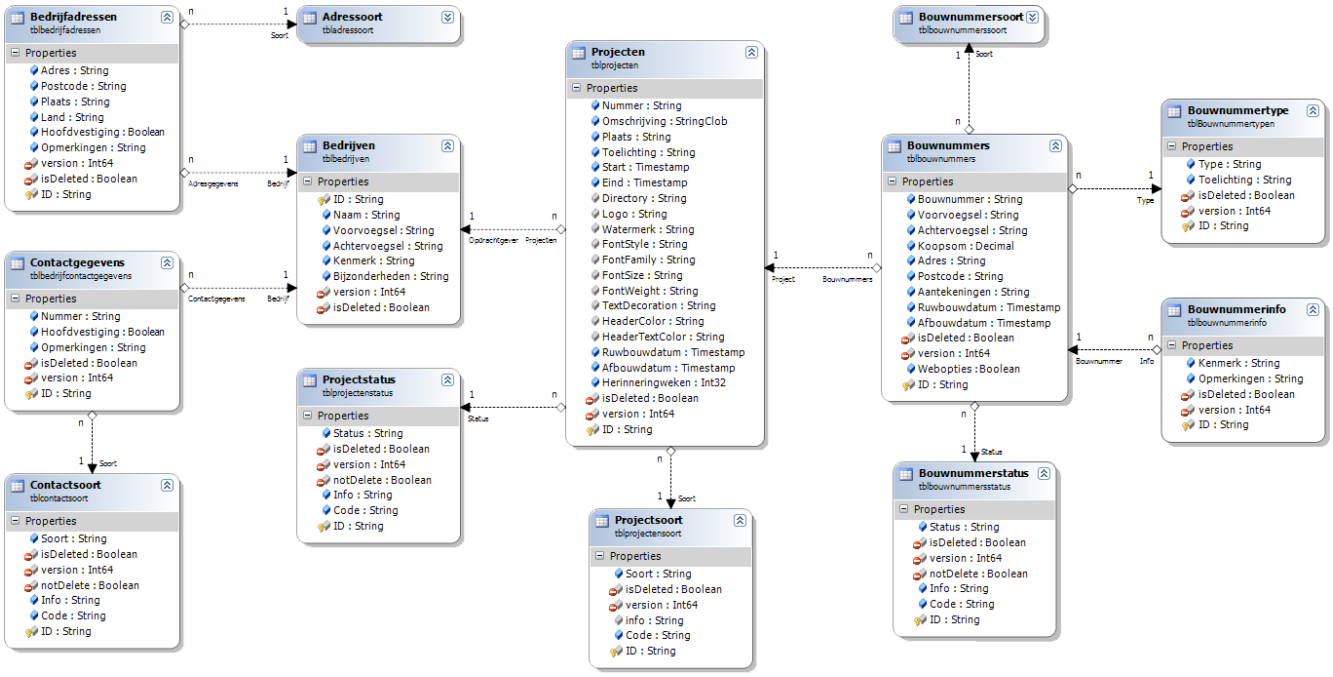


Figure 5.4: The model for SCart visualised using CRUDE.

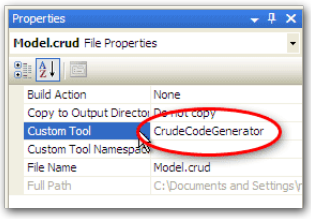


Figure 5.5: Changing the custom tool that generates the code.

5.3 Custom Code Extensions

As announced in Chapter 4, there are several ways built-in to support extension of generated CRUD applications. This section will focus on each of them. We will discuss extension through method overriding, event handling, addition of and removing functionality. Extendibility of generated applications is extremely important for its practical value. Generating code is only useful to save manual work, but it should not stand in the way of modifying the application for customisations.

5.3.1 Extension Through Overriding

The first way of extending the application is by overriding specific methods. The only method currently supported in CRUDE is the `ValidateData` method, for validating entered values in the edit windows. In the re-implementation, a few example extensions are made to show the ease of use in these situations.

In Chapter 4, we already discussed that data is already checked on validity on data type level (i.e. a text box used to communicate an integer field must accept only text values that may be converted to an integer), but sometimes checks at a more conceptual or application-specific

level are necessary.

For example, in the data entity `Bedrijfadressen`, there is a postal code field, which must be checked for validity, if not set to `NULL`. In order to facilitate that, we just have to extend the generated application with an overriden method `ValidateData`, which we will give the following implementation:

```
partial class tblbedrijfadressenEditWindow
{
    protected override void ValidateData(List<string> errors)
    {
        // Invoke any generated checks, first
        base.ValidateData(errors);

        // Check postal code
        if (!Fields["Postcode"].IsNull)
        {
            Regex checkPostal = new Regex("^[0-9]{4} [a-z]{2}$",
                                                RegexOptions.IgnoreCase);
            if (!checkPostal.IsMatch(Fields["Postcode"].Value.ToString()))
                errors.Add("Ongeldige postcode.  Formaat: \'9999 XY\'");
        }
    }
}

```

Another example spanning multiple fields is found in the extension of the `Projecten` data entity. In that class, during the insert action in the category “Planning”, there are fields for the start and end date of a project, which must be checked for chronology. This is done quite straightforwardly by extending the partial class `tblprojectenEditWindow`:

```
partial class tblprojectenEditWindow
{
    protected override void ValidateData(List<string> errors)
    {
        // Invoke any generated checks, first
        base.ValidateData(errors);

        // Projecten must be have a finish date after today
        if (!Fields["einddatum"].IsNull && !Fields["startdatum"].IsNull)
        {
            if (((DateTime)Fields["einddatum"].Value) <
                                        ((DateTime)Fields["startdatum"].Value))
            {
                errors.Add("Field 'Einddatum' must be later then 'Startdatum'.");
            }
        }
    }
}
```

5.3.2 Extension Through Event Handling

Another way of extending the application, is to act on occurring application events, fired by generated code. The current re-implementation fires three such typical CRUD events, namely `NewRow`, `EditRow` and `DeleteRow`.

Note that these are not data checks—the events are fired just before the associated action occurs and will modify the table. Hence, this is the last chance to change its value. The actions may be cancelled through the `Cancel` property of the `DataRowEventArgs` event argument, although rejecting actions should preferably be cancelled by validation checks.

A scenario in which using the `Cancel` property would in fact be useful is in the case of deletion. Something that has not been mentioned yet, in the SCart database, there are a few table rows that require to be available to the system—i.e. rows that may never be deleted[†]. These rows are marked with a `notDelete` bit column set to `True`.

In the re-implementation, extensions are made through events in two different ways. First, we may use untyped event handlers to perform custom logic and secondly, we may want to use the typed variants of the same events.

The hooking up of event handlers will typically be constructed either in the class constructor (when double-derived classes are generated) or by overriding the `InitializeGUI` method. The former approach will be used in the examples, since using double-derived classes is the preferred approach.

Untyped Event Handlers

Firstly, when CRUDE generates the GUI class for the application, it generates three event hooks, namely `NewRow`, `EditRow` and `DeleteRow`, of type `DataRowEventHandler` (see Listing 5.3).

```
1 public class DataRowEventArgs : CancelEventArgs
2 {
3     public DataRowEventArgs(DataRow row);
4     public string Table { get; }
5     public DataRow Row { get; }
6 }
7
8 public delegate void DataRowEventHandler(object sender, DataRowEventArgs e);
```

Listing 5.3: Definition of delegates and row event argument class.

These types are statically defined in the base GUI library (`Sogyo.Crude.Gui.dll`). Just before data rows (of any type) are about to be modified by any of these CRUD actions, the appropriate event gets fired. As can be seen, the information that gets passed along in the event argument is the table name on which the action is about to perform, and the row data is captured in the `Row` property, of type `DataRow`.

As `DataRow` can hold any kind of data, we can not tell at compile-time what row object we are dealing with. When we want to trim any entered company name, we are responsible ourselves for casting event arguments in order to work with them in a type-safe manner (or to just do it type-unsafe, which is even worse). Typically, this will imply tedious constructs like in Listing 5.4 on the facing page. In this case, we typically want to extend the application with type-safe event handlers, as we will discuss in the following section.

[†]An example of this is the `tbladressoort`, marking the address kinds. This table contains four rows: “private”, “visitors”, “work” and “headquarters”, which together form an enumeration defined in the database. The enumeration is extendible by adding rows to this table, but may never be shrunken.

```
protected void RowChange(object sender, DataRowEventArgs e)
{
    if (e.Row is SCartDataSet.tblbedrijvenRow)
    {
        // Trim the name
        SCartDataSet.tblbedrijvenRow row = (SCartDataSet.tblbedrijvenRow)e.Row;
        row.naam = row.naam.Trim();
    }
}
```

Listing 5.4: Type-unsafe handling of row modification events.

However, there can in fact be use for such a type-unsafe event handler. For example, consider the scenario discussed above, where some rows in the database may never be deleted. To provide accidental removal, we may handle the `DeleteRow` event as in Listing 5.5 below.

```
protected void SCartGui_DeleteRow(object sender, DataRowEventArgs e)
{
    try
    {
        object value = e.Row["notDelete"];
        e.Cancel = value != DBNull.Value && (int)value == 1;
    }
    catch (ArgumentException)
    {
        // Do nothing and ignore the non-existence of "notDelete"
    }
}
```

Listing 5.5: Preventing deletion of rows that may not be deleted.

Typed Event Handlers

Another way of extending the application is through type-safe event handling. In the re-implementation, this can be done by asking the code generator to also generate type-safe events for each desired data entity. The reason that, by default, these type-safe event handlers are not generated, is to avoid event explosion in the generated class, since for each data entity three events will be generated. Besides, since handling these events is probably more an exception than a rule, for each specific data entity, these type-safe events can be generated simply by setting the domain property `GenerateTypedEvents` for the appropriate `DataEntity` to `True`.

As we saw in Listing 5.4, trimming the name column's value of inserted or edited row data is tedious. Compare this to the implementation when type-safe events are enabled for the `Bedrijven` data entity (Listing 5.6).

A big advantage using this type-safe way of handling events is that we have much more object information available at compile-time already.

```
protected void RowChange(object sender, tblbedrijvenDataRowEventArgs e)
{
    e.Row.naam = e.Row.naam.Trim();
}
```

Listing 5.6: A cleaner, shorter and type-safe way of handling the row modification events.

5.3.3 Adding Other Functionality

Furthermore, non-framework related functionality might also be necessary to incorporate in the generated software. To facilitate that, knowledge of how the framework is technically constructed is required and the changes may not be too obtrusive. Typical changes that fall in this category are configuration of GUI looks. We will give short example of how such might be facilitated. In the re-implementation, the look and feel of the GUI is once set at start-up, in the constructor, using the following statements:

```
public partial class SCartGui
{
    public SCartGui() : base()
    {
        // Basic modifications of inherited Form properties can be used to change
        // the default appearance of the application
        this.Font = new Font("Verdana", 8f, FontStyle.Regular);
        this.ForeColor = Color.DarkGray;
        this.BackColor = Color.LightYellow;
    }
}
```

The same goes for the edit windows, of course.

Mailings

Functionality based on the state of the application is more difficult to build in. In the re-implementation, such an extension has been made, showing how extensions such as sending a mailing to all companies can be facilitated. This is shown in Listing 5.7 on the facing page.

Nevertheless, it is clearly visible that, although the code fragement necessary to extract data from the application is quite small, the code still requires a lot of low-level access to objects (i.e. `BindingSource` and `DataRowView`) and manual casts (see lines 34–37). Ideally, functionality to access data objects is the foremost feature request, in order to support application extension in a more general fashion. More on this can be read in the recommendations (see Section 8.3).

5.3.4 Removing Functionality

The last way of extending generated applications is facilitated by the removal of functionality. In Section 4.4.4, we have already seen that removal can merely be facilitated by the provision of built-in optional properties influencing whether or not the code generator provides the functionality or just leaves it out, thereby “removing” it.

```
public partial class SCartGui
{
    private System.Windows.Forms.MenuStrip menuStrip;
    private System.Windows.Forms.ToolStripMenuItem sendMailingToolStripMenuItem;

    public SCartGui() : base()
    {
        Initialisation of the menu strip goes here
    }

    private void actionsToolStripMenuItem_DropDownOpening(object sender, EventArgs e)
    {
        // Only enable sending mailings if "companies" is selected
        sendMailingToolStripMenuItem.Enabled = categoryList.SelectedItems.Count > 0
            && categoryList.SelectedItems[0].Tag is string
            && ((string)(categoryList.SelectedItems[0].Tag)) == "tblbedrijven";
    }

    private void sendMailingToolStripMenuItem_Click(object sender, EventArgs e)
    {
        BindingSource bsrc = m_crudetblbedrijvenBindingSource;
        SCart.SCartDataSet.tblbedrijvenRow row;
        DataRowView curr;
        StringBuilder addressees = new StringBuilder();

        // Keep the currently selected item's index, for we are going
        // to change the selection temporarily
        int pos = bsrc.Position;
        Cursor = Cursors.WaitCursor;
        try
        {
            for (bsrc.MoveFirst(); bsrc.Position < bsrc.Count - 1; bsrc.MoveNext())
            {
                if ((curr = bsrc.Current as DataRowView) == null)
                    continue;
                if ((row = (SCart.SCartDataSet.tblbedrijvenRow)(curr.Row)) == null)
                    continue;

                // From 'row', get the 'email' column
                addressees.AppendFormat("{0}, ", row.email, Environment.NewLine);
            }

            // Send out the mailing
            SendMailing(addressees.ToString());
        }
        finally
        {
            // Restore the original selected row's position
            bsrc.Position = pos;
            Cursor = Cursors.Default;
        }
    }

    /// <summary>
    /// Mock method to simulate sending out a mail to all given addressees.
    /// </summary>
    private void SendMailing(string addressees)
    {
        MessageBox.Show("Sending mail to:" + Environment.NewLine + addressees);
    }
```

Listing 5.7: An extension of non-CRUD functionality.

In the re-implementation, such an example is already seen when looking at the generation of type-safe events (see Section 5.3.2). Another such bit of functionality which is by default enabled, but can be left out, is the generation of the "entry point" selector, shown on the left of the default GUI. This functionality is generated dependent on the `HasCustomEntryPoint` property. For example, if the main form should be empty by default and the grids should merely by exposed by a menu selector, see Figure 5.6 below.

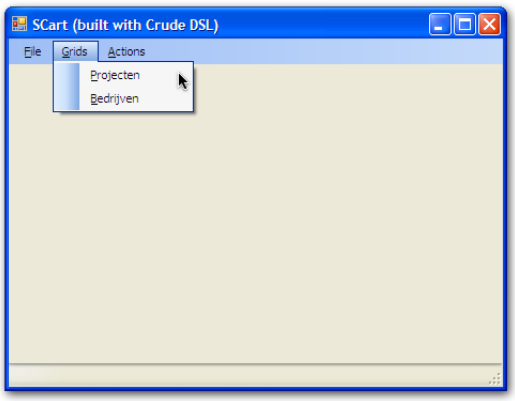


Figure 5.6: Removing default entry point functionality and replacing with custom logic.

This is facilitated by the straightforward implementation from Listing 5.8 below.

```
1  protected void bedrijven_Click(object sender, EventArgs e)
2  {
3      ShowtblbedrijvenGrid(logSplitter.Panel1, 0);
4  }
5
6  protected void projecten_Click(object sender, EventArgs e)
7  {
8      ShowtblprojectenGrid(logSplitter.Panel1, 0);
9  }
```

Listing 5.8: Implementation of custom entry point logic.

5.4 Practical Issues

An simplification of the existing SCart database is made for the case study (see Section 5.1.1), in order to reduce non-essetial complexity and to avoid obscuring the case implementation at hand. Although we may do so, we must address any issues raising from that simplifications if it has any practical consequences. In particular attention will be paid to incorporating the primitive versioning system into the generated software again.

5.4.1 Primitive Versioning

Primitive versioning is an essential detail and a must for Sogyo's clients (due to company and legal policies). Therefore, versioning is of very practical value and if software factories are ever used to re-implement a fully functional version of SCart, this is not something that can be neglected. This section is not intended to give a formal proof of how such extension can

be made, but rather provided to give Sogyo some practical clues on how to re-incorporate the functionality.

Modifying the Query Model

The first thing to observe is that this primitive versioning mechanism touches upon the most primitive form of data access—it is an implementation on the query level. After all, each query is extended with a condition causing only the correct row version selection. The database structure gets extended with two columns, `version` and `deleted`, indicating the versioning status. With "primitive versioning", we mean that of each conceptual row, only one version may be stored at a time. Thus, there can not exist two versions of a conceptual row. New rows get version 1, updated rows increment their version column and deleted rows can not be resurrected. Together, this lets us keep the `ID`-column the only primary column. Versioning is thus not used to keep a history, but merely to facilitate optimistic locking [21].

So, instead of the default read implementation

$$\texttt{SELECT } t(c_1), \ldots, t(c_N) \texttt{ FROM } t$$

we now get:

$$\begin{aligned} \texttt{SELECT} \quad & t(c_1), \ldots, t(c_N), t(c_{\texttt{version}}), t(c_{\texttt{deleted}}) \texttt{ FROM } t \\ \texttt{WHERE} \quad & t(c_{\texttt{deleted}}) = 0 \end{aligned}$$

In this new implementation, we must simply must make sure that the data is not deleted. Also, the query for row updates must be reviewed to increment the version number. This also incorporates the optimistic lock:

$$\begin{aligned} \texttt{UPDATE} \quad & t \\ \texttt{SET} \quad & t(c_1) := values(c_1), \ldots, t(c_N) := values(c_N), \\ & t(c_{\texttt{version}}) := version_{curr} + 1 \\ \texttt{WHERE} \quad & t(c_{\texttt{id}}) = id_{curr} \\ \texttt{AND} \quad & t(c_{\texttt{version}}) = version_{curr} \\ \texttt{AND} \quad & t(c_{\texttt{deleted}}) = 0 \end{aligned}$$

Here, $values$ is the array containing all new column values, as edited by the user. The variables id_{curr} and $version_{curr}$ hold the current ID (the primary column's value) and the version number that were last read from the database. (Also, we must assert that the row has not been deleted.) They identify the conceptual row that the user is editing. When we write back the changes to the database, we increment the version column.

The key idea is that, if someone else wrote some changes in the mean time, the version column is already incremented, so the row in the database does not match the $version_{curr}$ value and the update query will fail.

Finally, any `DELETE`-query of the form

$$\begin{aligned} \texttt{DELETE FROM} \quad & t \\ \texttt{WHERE} \quad & t(c_{\texttt{id}}) = id_{curr} \end{aligned}$$

should be replaced by the following:

$$\begin{aligned}
\texttt{UPDATE} &\quad t \\
\texttt{SET} &\quad t(c_{\texttt{deleted}}) := 1 \\
\texttt{WHERE} &\quad t(c_{\texttt{id}}) = id_{curr} \\
\texttt{AND} &\quad t(c_{\texttt{version}}) = version_{curr} \\
\texttt{AND} &\quad t(c_{\texttt{deleted}}) = 0
\end{aligned}$$

Since rows may never be truly deleted from the database, no `DELETE`-queries are allowed. Instead, the deletion is now implemented as an `UPDATE` query setting its status to deleted. Of course, also here, we must assess that the row is not changed since the last time it was read from the database.

Extending the Typed DataSet

Now that we know what we must change at the query layer, we must actually implement this in the generated DataSet since that is our wrapper for the actual queries. One thing to do is to manually specify the specific queries, but this would be very tedious, of course.

Preferably, we will extend the generated data set's querying model, by overriding the generated implementation of the `InitAdapters` method. The essential snippet is shown in Listing 5.9.

We propose the implementation as given in Listing 5.10 on the next page. Note that this proposal was not tested in practice yet. Undoubtedly, issues will occur when engaging in this business, but these should be able to overcome. A minor note on the re-implemention itself: we can not override the implement of the method, since it is marked private by the dataset generator. When we override, we create a new method with the same name, and explicitly declare it "new", then calling the base method instead, so we at least have the illusion of overriding.

This approach should give Sogyo some guidelines for implementation.

5.4.2 Composite Primary Keys

Finally, the same could be asked for the second implication of the database simplification. The issue of multiple primary keys can be simply overcome, though. For each unique combination of primary key column's values, one may record a unique identifier and use that identifier as the singular primary key instead. Any referencing composite foreign keys may then instead reference the new primary key column. Using that recipe, any database structure may thus be converted to a database structure that CRUDE can handle.

Summary

This chapter has illustrated the ease of which a CRUD application is generated using the CRUDE software factory approach. Technical implementation details are clarified and justified and particular attention is given to the extension of the generated output to provide custom functionality such as it is present in the original SCart application.

```
namespace TestWithData.SCartDataSetTableAdapters {
    ...

    public partial class tblbedrijvenTableAdapter : System.ComponentModel.Component {
        ...

        private void InitAdapter() {
            this._adapter = new System.Data.SqlClient.SqlDataAdapter();
            ...
            this._adapter.DeleteCommand = new System.Data.SqlClient.SqlCommand();
            this._adapter.DeleteCommand.CommandText =
                @"DELETE FROM dbo.tblbedrijven " +
                @"WHERE ((ID = @Original_ID) AND ... " +
                @"  AND (deleted = @Original_deleted))";
            ...
            this._adapter.InsertCommand.CommandText =
                @"INSERT INTO dbo.tblbedrijven " +
                @"(ID, naam, achtervoegsel, ..., version, deleted)" +
                @" VALUES (@ID, @naam, @achtervoegsel, ..., @version, @deleted)";
            ...
            this._adapter.UpdateCommand.CommandText =
                @"UPDATE dbo.tblbedrijven " +
                @"SET ID = @ID, naam = @naam, ... " +
                @"WHERE ((ID = @Original_ID) AND ... " +
                @"  AND (version = @Original_version))";
            ...
        }
    }
}
```

Listing 5.9: Generated queries should be replaced.

```
public partial class tblbedrijvenVersionedTableAdapter : tblbedrijvenTableAdapter
{
    private new InitAdapter()
    {
        // Get all the default implementation details
        base.InitAdapter();

        // Then, only adapt the queries
        this._adapter.DeleteCommand.CommandText =
            @"UPDATE dbo.tblbedrijven SET deleted = 1 WHERE ...";
        this._adapter.InsertCommand.CommandText =
            @"INSERT INTO dbo.tblbedrijven (ID, naam, ..., version) " +
            @"VALUES (@ID, @naam, ..., 1)";
        this._adapter.UpdateCommand.CommandText =
            @"UPDATE dbo.tblbedrijven SET ID = @ID, naam = @naam, " +
            @"WHERE ...";
        this._adapter.SelectCommand.CommandText =
            @"SELECT * FROM dbo.tblbedrijven WHERE ...";
    }
}
```

Listing 5.10: Proposed re-implementation of the `InitAdapter` method.

Most important, the re-implementation made in this chapter provides a base component comparable to the original SCart application, to facilitate the a quality assessment, which will be the focus of the upcoming two chapters.

6 An Evaluation Framework

From here, this research will focus on the case at hand, as discussed in the last chapters, in order to make a funded comparison between the case scenarios. Before doing so, we should have a notion as to compare on what grounds.

This chapter's intent is to prepare a framework for evaluating software quality systematically. Important factors for judging software quality will be obtained and the nature of their relation is made visible. Next, which factors are most relevant are selected after discussion and finally, an approach to measure the quality using indicators is proposed.

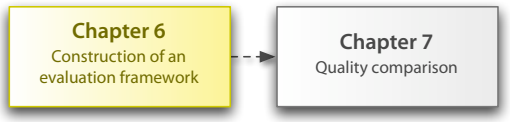


Figure 6.1: Chapter 6 in its context.

6.1 Software Quality

Since the early days of software engineering, a lot of literature has been devoted to the measurement of software quality attributes [2, 7, 39, 49, 57, 63, 73]. The problem with "quality" is that its definition is in the eye of the beholder. The perspectives of people in the software development process will differ.

Definition 5. *Software quality* is "the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs" [39].

6.1.1 Factors & Criteria

McCall, Richards and Walters [49] have prepared a list of software quality *factors* that is commonly embraced in software engineering literature. They identify three categories—operational, revisionary, and transitional—into which each of the quality factors can be divided.

- ▷ **Product *operation*.** These factors concern product quality when working with it (i.e. operating it).
- ▷ **Product *revision*.** Concerns the factors that determine the quality of the product when developing it.
- ▷ **Product *transition*.** The factors that have to do with the ability of the product to adapt to changing environments in which it is put to use.

Table 6.1 shows these quality factors categorised by their nature. Next, they propose the quality *criteria* that strongly relate to the quality factors in a more low-level fashion. These criteria are shown in Table 6.2.

Factor	Description
Operational	
Correctness	The extend to which the product satisfies the technical specifications and the user requirements.
Reliability	The probability that the product will not fail during a specified time under given conditions.
Efficiency	The amount of resources required by the product to perform its task. This can either be memory, disk space, (CPU) time, bandwidth, etc.
Integrity	The extent to which the product can be accessed or controlled by unauthorised persons or computers.
Usability	The amount of time it takes for a person to learn to operate, prepare input, and interpret output of the product, and perhaps even to understand it.
Revisionary	
Maintainability	The effort required to maintain the product in its broadest sense, both preemptive and corrective. This not only concerns fixing errors, but also extending the product, or understanding the source code.
Testability	The amount of effort required to ensure that the product does what it is intended to do.
Flexibility	The amount of effort it takes to modify an operational program.
Transitional	
Portability	The amount of effort required to transfer the product between two environments.
Reusability	The extent to which the product, or at least a part of it, can be reused in other products.
Interoperability	The effort required to couple other applications to the product and the other way around.

Table 6.1: Quality factors by category.

Table 6.3 shows the relationships between the factors and the quality criteria. The separation of quality *factors* from *criteria* is somewhat subjective and literature does not fully agree on this issue. Pressman [57] and Sommerville [63] for example do not even attempt to make this separation and mention a flat list of characteristics that they call quality *metrics*. Furthermore, both Arthur [1] and Kitchenham [43] argue that complexity is an important factor that misses in this enumeration. However, complexity *is* accounted for in McCall's original model, although not directly visible—it is reversely formulated in the quality criteria *simplicity*.

Criterion	Description
Access audit	The ease with which the product can be checked for standard or requirement compliance.
Access control	The amount of control that is exposed and protection that is available in the product.
Accuracy	The precision of the product's computations, the required precision of the input and the precision of the product's output.
Communication commonality	The extent to which standard protocols and interfaces are used.
Completeness	The extent to which the required functionality is available/implemented in the product.
Communicativeness	The amount of effort with which inputs and outputs can be assimilated.
Conciseness	The compactness of the product's source code, in terms of lines of code, for example.
Consistency	The extent to which uniform design and implementation techniques and notations are used throughout the development of the product.
Data commonality	The extent to which standard data representations are used.
Error tolerance	The extent to which continuity of operation is ensured under adverse conditions.
Execution efficiency	The run-time efficiency of the product, in terms of speed.
Expandability	The extent to which storage requirements or functionality of the product can be expanded.
Generality	The wideness of the potential target domains that the product (or a component thereof) can be used for.
Hardware independence	The extent to which the product is dependent on the underlying hardware it runs on.
Instrumentation	The extent to which the product provides for measurements of its use or identification of errors.
Modularity	The extent to which modules are independent of each other.
Operability	The amount of effort required to operate the product.
Self-documentation	The extent to which the product provides in-line documentation that explains the implementation details of functionality or components.
Simplicity	The extent to which complexity is avoided, i.e. the ease with which the product can be understood.
Software system independence	The extent to which the product is independent of its non-hardware environment. Consider non-standard language constructs, required operating system, libraries, database management systems, etc.
Storage efficiency	The compactness of required run-time storage resources of the product.
Traceability	The ability to directly and easily link software component implementations to requirements.
Training	The ease (i.e. amount of time it takes) for an inexperienced new person to learn to operate the product.

Table 6.2: Quality criteria.

	Operation					*Revision*			*Transition*		
	Correctness	Reliability	Efficiency	Integrity	Usability	**Maintainability**	**Testability**	**Flexibility**	Portability	**Reusability**	Interoperability
Access audit				✓							
Access control				✓							
Accuracy		✓									
Communication commonality											✓
Completeness	✓										
Communicativeness					✓						
Conciseness						✓					
Consistency	✓	✓				✓					
Data commonality											✓
Error tolerance		✓									
Execution efficiency			✓								
Expandability								✓			
Generality								✓		✓	
Hardware independence									✓	✓	
Instrumentation							✓				
Modularity						✓	✓	✓	✓	✓	✓
Operability					✓						
Self-documentation						✓	✓	✓	✓	✓	
Simplicity		✓				✓	✓				
Software system independence									✓	✓	
Storage efficiency			✓								
Traceability	✓										
Training					✓						

Table 6.3: Relation between quality factors and criteria, adapted from [49].

It is important to note that the factors as mentioned here concern the *output* of the software development process, i.e. the final product that is delivered to Sogyo's customers.

6.1.2 Consequences

Table 6.4 shows the trade-offs between the factors mutually. Some factors affect others in a positive or negative fashion. For example, efficiency has negative influence on almost any other factor. Looking only at the relevant factors for this research, we may conclude that each of them only positively contributes to the other factors. If flexibility increases, for example, then maintainability, testability and reusability will each be positively affected, too—which is not hard to imagine. This being the case, we do not have to worry about troublesome trade-offs, negatively contributing to some factors when targeting another.

Nevertheless, although any contributions made to these factors will always have a positive

	Correctness	Reliability	Efficiency	Integrity	Usability	**Maintainability**	**Testability**	**Flexibility**	Portability	**Reusability**	Interoperability
Correctness		+			+	+	+	+			
Reliability	+				+	+	+	+		–	
Efficiency				–	–	–	–	–	–	–	–
Integrity			–		+			–		–	–
Usability	+	+	–	+		+	+	+			
Maintainability	+	+	–		+		**+**	**+**	+	**+**	
Testability	+	+	–		+	**+**		**+**	+	**+**	
Flexibility	+	+	–	–	+	**+**	**+**			**+**	
Portability			–			+	+			+	+
Reusability		–	–	–		**+**	**+**	**+**	+		
Interoperability			–	–					+		

Table 6.4: Trade-offs between software quality factors, adapted from [73].

impact on the others, there are still factors that might be negatively affects, but fall out of the scope of this research—efficiency being the most obvious example. We will keep this in mind and make note of it, although not going into the details.

6.2 Scope

Basically, the operational properties are not interesting to consider, because the comparison between traditional development and development using software factories does not affect operational characteristics of the output product. Instead, it affects revisionary and transitional characteristics the more.

Therefore, we will narrow down the categories, leaving some quality factors that might be interesting will be selected. The three most important factors concerning this research are *maintainability*, *flexibility*, and *reusability*. To a lesser degree, *testability* might be considered, too.

From this, we may derive the list of quality criteria that will be relevant for this research. The low-level quality criteria that relate most to the selected factors are also shown in bold face—in order of importance: *modularity*, *generality*, *expandability*, *self-documentation*, *conciseness*, *simplicity*, *consistency*, and *instrumentation*. *Hardware* and *Software system independence* turn out to be not really interesting. Details are given in the next section.

The criteria that are relevant for answering the research questions will depend on the quality factors that play important roles in the problem causes, described in Section 1.3. Table 6.5 provides an overview of the relationships and relevance between the quality criteria and the causes.

	Programming paradigm limitations	Software engineering methodology limitations	Development environment limitations
Conciseness	✓		✓
Consistency		✓	✓
Expandability	✓	✓	✓
Generality		✓	
Hardware independence			
Instrumentation			✓
Modularity	✓	✓	✓
Self-documentation			✓
Simplicity	✓	✓	✓
Software system independence			

Table 6.5: Relation between quality factors and causes.

6.2.1 Conciseness

Conciseness, being a measure for the compactness of code, is often expressed in low-level terms of lines of code. Influential factors on that measure are the paradigm and the technical development environment. Methodology changes—within the same paradigm—are not of any substantial influence.

- ▷ **Paradigm limitations.** Paradigm may influence the effort necessary to express a particular problem. The classic example being that it requires far less constructs (and thus, lines of code) in a functional language to sort a list then in OO-languages. The more the mental model of the problem domain matches the language model, the less code will be necessary;

- ▷ **IDE limitations.** The second influential limitation are the ones imposed by the IDE. Modern IDE's are so extensively automating the process that they are eroding the idea that decreasing the lines of code improves quality. A better measure maybe would be the number of *hand written* lines of code. Dragging and dropping user interfaces (UI) together in modern IDE's might generate thousands lines of code, but would not require a lot of mental effort nor decrease understandability, for example[†]. Generally

[†]Understandability of the generated source code would possibly decrease. However, specific tools, such as the UI designer itself, would represent that information in a way that it is less complex to understand.

speaking, IDE's reduce complexity by hiding the underlying lines of code, which makes those lines virtually irrelevant and should not falsely be used as an indicator of quality[†].

6.2.2 Consistency

Consistency is concerned with the uniformity of the vision, models, design and implementation details that are used throughout development. Paradigm limitations do not pose an issue on the consistency issue. Methodology, though, has a much bigger impact on the consistency. Also, the IDE limitations might pose a threat to it.

- ▷ **Methodology limitations.** First, the methodology used will have a significant influence in determining what is considered important or not. For example, compare the extreme programming (XP) methodology to the more established methodologies like RUP, or DSDM. The former values *adaptability* over *predictability*, which has major consequences for consistency, in exchange for other software qualities (most related to *flexibility*) [36];

- ▷ **IDE limitations.** The second way in which consistency might be threatened is by a limit in the supported methods in an IDE. A software's design may assume a certain level of supported language constructs. When a design uses these constructs, and the IDE does not, or not fully, support these constructs, the developers have a problem and should work around it—which compromises consistency.

6.2.3 Expandability

Expandability concerns the extent to which efforts are necessary to add functionality to the product. When there already is a certain partial product implementation, the product must be extended (add implementation details) in order to satisfy requirements (be they existing or new ones). Expandability is influenced by all three factors.

- ▷ **Paradigm limitations.** The foremost influential factor of the expandability criterion is the paradigm that is used. Programs consist of constructs representing both functions and data. Traditionally, functional languages have the property that they easily support the extension of functions to the program, but at the same time data types are hard to extend. This is because all data members are kept together in one place, where it is defined. Object oriented languages have the reverse problem—they are good in extending data (think of inheritance), and functions are harder to add;

- ▷ **Methodology limitations.** Next, the methodology plays another important role. As with the previous discussion in the consistency criterion, the methodology significantly determines which system properties are preferred over other (adaptability over predictability). Adaptability, as preferred by the extreme programming methodology, improves expandability because the development process is shaped to specifically aim for that property, for example, by preferring software that is more flexibly adaptable to different situations over software that strictly follows the specifications;

[†]The UI designer—or any other IDE abstraction for that matter—requires an indicator for the required mental effort of a specific task.

- ▷ **IDE limitations.** Finally, also the IDE influences expandability. It is of great importance that the IDE provides support for expandability, i.e. guaranteeing that any choice made in the past would not lock-in development. While design decisions always lead to a certain level of lock-in, they should be changeable with relatively low effort. At a low level you can think of refactoring support, but at a high level one would require the IDE to be able to change the software model and to have the implementation-so-far reflect the new model, and *vice versa*—i.e. one would require full synchronisation between the model and the code.

6.2.4 Generality

Generality is concerned with the wideness of potential (relatively similar) domains that the product might be able to target. In case that direct targeting of domains is not possible, generality might as well reflect the ease of which the product might be adapted to enabled the targeting of such a (new) domain. This is an important quality factor.

Paradigm limitations are of no influence to this criterion, because the paradigm choice obviously does not affect the broadness of the product's potential target domains—for example, equally general or specific programs can be written using either functional or imperative languages.

Whether IDE's or frameworks can be of influence to the generality of the end product might be questioned. One might argue that tooling can be able to assist the programmer to write more general code. For example, refactoring functionality is included in modern IDE's, which makes the programmer's menial life indeed easier. However, before applying such tools, the initiative and the creative or intellectual efforts necessary for a programmer to perform are not in any way improved by such tooling. Therefore, we argue that generality is not affected by IDE's or frameworks.

- ▷ **Methodology limitations.** Methodologies, on the other hand, do play an important role and affect the program's generality significantly. Methodologies that value flexibility (in terms of adaptability) higher than accuracy or completeness are more likely to produce general applications, because their very nature prescribes to produce the most common set of functionality and implement the variabilities in such a general fashion that the software is highly configurable of adaptable to changing requirements. This approach is also reflected in the FAST-methodology [75];

6.2.5 Hardware independence

Independence of hardware is an important software property. However, this is not applicable in relation to the given causes, for the following reasons. Firstly, programming paradigms does not specify anything related to hardware nor about being independent from it—they describe a perspective from which computational models are derived.

Secondly, the relation to methodology is as irrelevant as that to the paradigm. Methodologies do not prescribe approaches to hardware independence—they describe a perspective of *how* programming problems might be best modeled. Hardware independence can be a specific requirement for software projects, but it is decoupled from the methodology.

Finally, IDE's might of course influence the hardware independence, because IDE's are sometimes focused to a specific platform. However, the hardware independence question would be irrelevant for Sogyo, because the project's scope is specifically oriented to the Microsoft .NET platform, rendering this quality factor a useless one.

6.2.6 Instrumentation

Instrumentation concerns the extent to which the product provides measurements for use and identification on errors. Being a very technical quality criterion, and a specific product-related one, this is only affected by the IDE. Paradigm and methodology have nothing to do with instrumentation because it is a technical quality criterion.

- **IDE limitations.** Programming languages will be the factors that are responsible for run-time error identification. Part of the quality of a good programming environment is the way in which low-level errors can be translated or interpreted to match high-level constructs, as written in the language. The same goes for IDE interfaces that build on the underlying programming languages. The extent to which underlying program errors are, or can be, translated to concepts from these interfaces determine the quality of those interfaces.

6.2.7 Modularity

This criterion is a very important software engineering concept, which literature clearly agrees on. The three factors influence this property visibly. The modularity criterion as such strongly enables the expandability criterion, which makes them related.

- **Paradigm limitations.** Firstly, the chosen paradigm affects the modularity level of software. The paradigms as such do not specify how modules should be formed, but different paradigms will yield different mind sets, causing a different grouping of software concepts into modules. For example, in functional languages, data definitions are often kept together tightly in modules. In object oriented languages on the other hand, functions are typically grouped inside modules. This shows that the paradigm influences the modularity of software;
- **Methodology limitations.** While all software engineering methodologies preach modularity, some of them will prescribe a more stringent approach than others. For example, modularity has a very prominent role in domain-driven design;
- **IDE limitations.** Finally, the development environment might influence the level of modularity that is technically achievable. Negative influence is clear when an IDE is not designed for modularity. For example, once a piece of software is designed, that part would not be (easily) extensible with new functionality, or responsibility is not orthogonally divided among objects or modules.

6.2.8 Self-documentation

Self-documenting software is the walhalla for software developers. The main reason that documentation is necessary, is that software becomes unreadable when you stop working with it for a while.

Of course, the paradigm used is of no importantance for this quality criterion. Arguably, the methodology might pose some influence to self-documentation, but that would rather be a valuation of it, not an influential factor to the quality itself.

- ▷ **IDE limitations.** The main road to a manageable documentation procedure is to write it along with the code. Automation might play an important role in this, and modern IDE's provide automatic documentation templates at the press of a key. Although these systems lower the threshold to actively document, they often lack capabilities to keep the documentation up-to-date—the documentation is not synchronised with the source code.

6.2.9 Simplicity

Simplicity encompasses the relative ease with which the product can be understood, modified, or learned to use. It concerns the amount of base knowledge required to mentally grasp the product's intent, or its inner workings. How much knowledge is required will, amongst more, be determined by the internal complexity and elegance of the (source code of) the product. The more intuitive the product is, the less knowledge is required.

Which methodology is used is of no influence to the simplicity. The organisation of the development cycle is unrelated to the complexity of the software created in it. The other two factors are relevant, though.

- ▷ **Paradigm limitations.** The first factor that may evidently affect the complexity of the product is the paradigm. Using procedural languages, for example, one might suffer the risk of creating so called "spaghetti code". Functional and object oriented languages provide abstraction mechanisms that allow for a higher level of reasoning, improving simplicity;

- ▷ **IDE limitations.** Secondly, the IDE might greatly influence the complexity of the software created with it. This can either be positive as well as negative. IDE's may guide software development by generating a lot of source code for the programmer, or modify existing code—for example, think of refactoring or pattern templating.

  However, there also might be a downside to it: when an IDE generates code, this code may be pretty long and not straight-forward to understand (i.e. complex). Without guidance to map the code to a mental picture, complexity grows instead of shrinks.

6.2.10 Software system independence

This concerns independence on non-hardware environment, for example non-standard language constructs, or run-time requirements such as available libraries or the operating system.

Similar to, and for the same reasons as hardware independence, software system independence is only relevant in cases where it is an explicit requirement for the product—not in general as a quality measure.

6.3 Measuring Quality

Measurements are a way to prove the presence of characteristics of things. Measurement is limited by the capacity and specificity of instruments available. Especially measuring “soft” (qualitative) attributes is complex, if possible at all. Instrumentation can merely measure “hard” (quantitative) indicators that have a correlation with the soft attributes. This will guarantee by no means that, measuring hard attributes will pose a measure for the soft ones. On the other hand, it also does not mean that we can not draw any conclusion at all from the measurement.

Quality is a typical soft attribute of software. In order to be able to at least measure some properties of the software, we suggest a model for valuing the selected quality criteria for this framework, based on relatively quantitative attributes of the software, which we will call *indicators*. Each of the selected criteria will be discussed and the indicators for those attributes will be proposed and discussed.

Now that we have set up the approach for measuring the characteristics of software quality, we might go on and actually compare the development methods using these indicators. This will be the topic for the rest of the research.

Criteria	Indicators	Objectiveness	Best-case	Worst-case
Conciseness	▷ Percentage of code lines written manually.	★★★★★	Few	Much
Consistency	▷ Extent to which the tooling forces the programmer to produce consistent output.	★★★	Much	Not
	▷ The extent to which the programmer might still abandon from that in special cases.	★★	Much	Not
Expandability	▷ Impact of changes for an architectual extension.	★★	Little	Severe
	▷ Impact of changes for a procedural extension.	★★	Little	Severe
	▷ Impact of changes for a data extension.	★★★	Little	Severe
Generality	▷ Diversity of domains the product can be used for.	★	Many	One
Instrumentation	▷ Semantic level of *compile-time* error messages.	★★★	Domain	Language
	▷ Semantic level of *run-time* error messages.	★	Domain	Language
Modularity	▷ The ease of which logical units of functionality (intent) can be located in the implementing source code.	★★★	Easily	Hardly
Self-documentation	▷ The ease of which logical units of functionality (intent) can be extracted from implementing source code.	★★★	Easily	Hardly
Simplicity	▷ The amount of knowledge required to *understand* the application's source code.	★★★	Very low	Very high
	▷ The amount of knowledge required to *modify* the application's source code.	★★	Very low	Very high

Table 6.6: Indicators for each quality attribute. The hardness column indicates the degree to which the indicators are intuitive, precise, clear and directly measurable. Low-starred indicators might still be subjective to a degree.

The indicators are not objective as such—when two peers are asked to measure the quality indicators for a given software product, the answers will probably be different. However, the indicators are acceptably objective when used in comparison. This is comparable to the question of whether a book is thick as an indicator for measuring the quality of the book. Each peer asked would have a different opinion on the question, thereby making it a useless one due to its subjectivity.

However, in some cases, we might for example set a threshold value for the amount of pages in order to rate a book's thickness. In those cases, we might ask "Is this book thicker than 500 pages?", which turns the question into an objective one. The difficulty in this approach is to find a threshold that makes the question useful without over-simplifying the subject under measurement. In practice, this is a baroque field of tension and exactly the reason why soft attributes are so hard to measure.

The approach that we will take here instead, therefore, is to not compare to an arbitrary chosen threshold (the 500 pages), but to rather compare to subjects and tell the difference on each aspect. That turns the question into "Which book is thicker?", posing an objective comparison. Although some knowledge is not collected by this approach (e.g. "How much thicker?"), but we are not interested in that initially.

Summary

In this chapter, a framework for the comparison of software products based on eight specific criteria is developed. The criteria are selected based on their relevance to Sogyo's business challenges, but essentially, the framework is developed in a way that can easily serve as a blueprint for similar future research, for other business challenges. Finally, indicators are proposed to measure relative quality as objectively as possible.

In the next chapter, we will use the framework and use it to assess the relative quality of the traditionally developed SCart application from Chapter 3 with the re-implementation as constructed in Chapter 5.

7 Comparison

In this chapter, we will place the traditional methods of software development and the method of using software factories side by side and compare them. We will base our comparison on the indicators, as selected in Section 6.3. Based on this comparison, we will finish with some conclusions with respect to the quality and usability of the software factories methodology in Chapter 8.

Also, we will share insights that came to light during the re-implemention and try to extrapolate these to express some generic presumptions on the use of software factories.

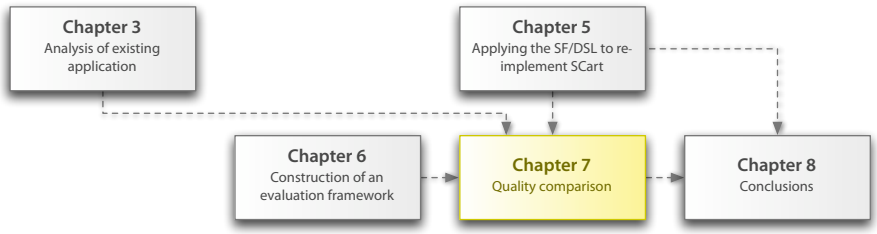


Figure 7.1: Chapter 7 in its context.

The comparison is performed by looking at the differences between on the one hand the case study (the existing SCart application) and on the other hand the re-implementation from Chapter 5. We will conclude with an interpretation of the comparison results.

7.1 Comparison

In this section, we will perform the actual comparison of the original with the re-implemented versions of SCart. Each of the eight subsections below will discuss the products on a specific criterion.

7.1.1 Conciseness

To determine the conciseness of both projects, we will purely focus on the amount of code lines that exists in the projects.

These indicators render a perception of the total size of both the projects as the size of the manual labour involved in writing them. From here, we may simply calculate the percentage of manual labour for each project.

To make a fair comparison between both projects, the projects should be of equal size and functionality. This is currently not the case. The original SCart project contains quite a lot more logic not directly related to CRUD applications (reporting, e-mail functionality, remote access, etc.). It would not be a funded comparison is we would include this functionality in the comparison. Therefore, we have tried to exclude this functionality from the comparison by removing all code specifically related to these aspects—all files from the directory `Wizards` and all filters (`Filter*.cs`).

Furthermore, we have also excluded all code as generated by Microsoft's dataset designer, since that code body is, although very large, fully generated. By doing this, we have only kept the code body that is directly involved in database or CRUD actions or directly related to required GUI elements.

The Visual Studio solution SCart has been stripped down for this purpose to a version at least comparable to the re-implementation. In this stripped-down version, we look only to the `Forms` project, containing the actual application. All the other projects are setup projects, or projects to facilitate remote database connections. Therfore, these projects are removed from the solution: `ClientDataLayer`, `ClientSetup`, `ClientSetupCustomActions`, `Documentatie`, `Raportage`, `Scartweb`, `ScartwebSetup`, `Security`, `ServerDataLayer`, `ServerSetup` and `SqlServerProject`. Because the project `Forms` contains no direct references to any of the removed directories, the solution is still usable and the application can still be compiled.

Comparing the amount of code lines, we take the file types into account. We only want to count the amount of code lines from the actual source code files. Extensions of files that are excluded from line count are:

```
*.jpg *.png *.gif *.bmp *.ico *.cur *.resx
*.dot
*.mdf *.ldf
*.xsc *.xsd *.xss
*.crud *.diagram
*.sln *.csproj *.suo *.user *.settings
*.vspscc *.vssscc
*.vstemplate
*.snk *.rdl *.dll
```

We found:

Indicator	**SCart**	**CRUDE**	**Re-impl.**
Total lines of code.	68,350	30,312	6,767
Number of lines of code written manually.	65,694	6,303	120
Percentage of code written manually.	**96%**	**21%**	**1.7%**

The remaining difference in the number of source code lines can now merely be caused by the fact that, in the re-implementation a only a limited subset of the database tables was used. At the original SCart-database, the implementation of course uses all tables.

An important remark that should be made at this point is that the absolute amount of code lines that manually has to be written, will significantly grow steeper for the original SCart application with respect to the re-implementation if the number of copies increases, see Figure 7.2.

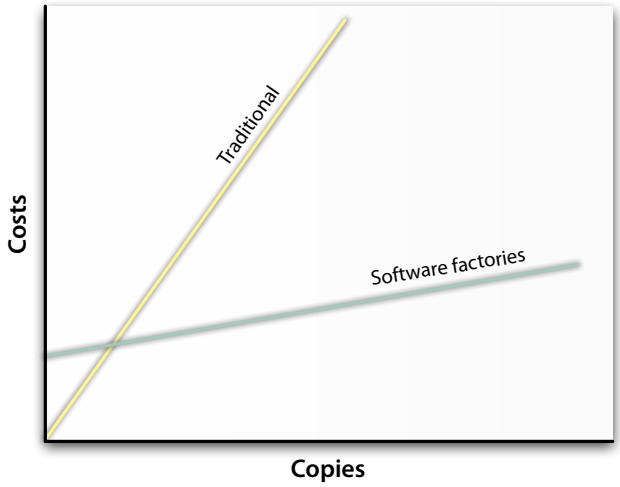


Figure 7.2: Amount of manual work in terms of lines of code to be written manually.

7.1.2 Consistency

By making a framework along which classes can be structured in SCart (for example through inheritance of GUI components, recall Section 3.5.1), the number of legal implementation constructs is reduced for the sake of consistency. However, these are purely limitations that are imposed by the SCart designers themselves. Judging from the mere tooling there is a relatively high amount of freedom as it imposes no limitations whatsoever. At language level, there is implementation freedom. Although the tooling does enforce correct language constructs (grammatically speaking), it does not contribute to consistency as such.

This lack of steering offers freedom, but also lower consistency, since the extent of consistency is thereby determined by the discipline of the individual programmer [17]. In Chapter 3.5.2, we already saw that this is a real problem to SCart.

With the re-implementation, one is bound to the domain model. This means that the implementation is described at a high level of abstraction and control over generation lies not in the hands of the programmer[†]. Modifying code in the code generator will lead to a consistent change of that code throughout the generated program. This implies that generated code is much more consistent.

For example, when the representation of a boolean field should change from a check box to a selection box containing options “Yes” and “No”, such change can simply be made in one place in the code generator, but will require many line changes in the original application. SCart’s implementation therefore is much more susceptible to code erosion.

Of course, the downside of that is that the code is also more rigid. Exceptions to the rule are more easily built in in SCart, since there are no implementation limitations. Considering

[†]Note that it does lie in the hands of the DSL programmer, but not in that of the DSL user.

a model from which code is generated, extensions can hardly be facilitated unless they are anticipated on beforehand. It is the DSL maker's responsibility and challenge to think of that *a priori*. The re-implementation shows that diverging from the generated framework is possible by providing possibilities for extension at smart places in the code, but those possibilities are also limited in practice to exactly those.

7.1.3 Expandability

Extending the application can take place on a few levels. Pressman [57] indicates three to which expandability might be applicable: architectural, procedural and on data level. Architectural changes regard how implementation parts of the software work together. Procedural changes concern the operational aspects of the application, for example the algorithmic. Finally, data extensions are the extensions that regard the data on which those algorithms work.

Architectural

An example of an extension with architectural impact, is the decentralisation of the database. It is rather imaginable that a few SCart users decide they want shared access to the database and that there are multiple databases that are kept in sync from time to time.

For SCart, such a change would probably mean starting from scratch, since its architecture is not built to support such constructs. SCart's source is a single database structure. If this fundament is changed, the application must drastically be rebuilt. For the re-implementation the exact same holds, because it is not a re-implementation on essential architectural levels.

Both applications are thus not just like that shielded to this kind of changes, which is rather characteristic for changes with architectural impact—this is not a meaningful comparison. The implementation of architectural changes although will be significantly harder in the re-implementation than it is in SCart, since the code generator should fully follow the new structure of the application. At the same time, the code generator is also more verbose, which thwarts the process even more.

Procedural

An example of an extension that requires procedural changes, is the addition of revision management. We have already touched upon what should be changed in order to incorporate such change in Section 5.4.1. The structure of the application remains identical, although the queries should be re-implemented.

In SCart, this would, among more, mean that all `buttonSave_Click` methods from all classes inheriting from `EditBase` would be manually re-implemented. In the current SCart version, this simple change already affects 58 classes alone.

Making changes like this in CRUDE based applications means re-implementing the functionality on one place, namely in the method `GenerateInsertOrUpdateMethod` in the class `CrudeCodeGenerator`. The CRUD model allows us to have access to the necessary table information during compile-time and thus generate implementations for all table types.

The downside to this is that the generated functionality is uniform, so differing from that

code is harder in CRUDE-based applications. For example, when such procedural modification would be required for only an instance or two, there must be fallen back on the model in which this conditional must be implemented as a new variability.

Data

An extension on the data level might regard the extension of the database with extra tables, columns or relations. These extensions can be easily dealt with at DSL usage time, by reflecting them in the application model. Application code is automatically updated to reflect the new state.

In SCart, again, these changes should be reflected manually in code. Although the changes are easy to make, the work involved can be significant. The SCart code is soaked with statements like:

```
bedrijvenrow.IDsoort =
        this.smitshoekTableDataSet.tblbedrijven.Rows[0]["IDsoort"].ToString();
bedrijvenrow.isDeleted = false;
...
```

Almost the full source code of SCart consists of such statements. In CRUDE, we can profit fully from the compile-time knowledge resulting from using typed field names. This makes the extension of the model already sufficient to deal with the changing underlying data model.

7.1.4 Generality

Generality is hardly measurable a criteria. We will try to provide a degree to which (with little effort) the application source code can be adapted such that the application is also usable within another (business) domain.

Generally, the flexibility of an application's architecture is measure that influences the generality of the application's applicability. Both SCart as well as the re-implementation are specific implementations, but each in their own way.

The SCart source code is soaked with specific logic for dealing with domain-specific jargon, such as BOUWNUMMERS, etc. That negatively influences the generality of the application's domain. On the other hand, that code is flexible to such a degree that making adaptations to those specific constructs is relatively easy. Whether it was intended to, redundancy of code meant, purely technically speaking, also flexibility.

In the re-implementation, the opposite is the effect. The (generated) application code contains no specific data-related constructs, so modifying the application on data level is relatively easy. This is a direct reflection of the variabilities that are modelled. But it is exactly this blessing that also is a curse: when a change can not be expressed within variabilities of CRUDE, the code generator must immediately be adapted with architectural impact.

The diversity of the domains for both implementations differs enormously. Looking purely at other CRUD applications, for example a personnel administration for a dentist practice, then SCart requires much more work (again, based on the new database design a new application

must be developed from scratch), while with CRUDE, this requires only minor tweaking—reflecting the database structure in the model and regenerating the application is all there is to it.

However, when we look outside the CRUD domain, then CRUDE's value will decay quickly. This is a direct result of the domain-specificity of CRUDE, of course. To illustrate how quickly the value will drop, even an apparently small difference in domain may already cause this. A financial accounting application shows some resemblance with a CRUD application, but even this would be a problem to CRUDE. After all, specific logic is required for an accounting application, because the CRUD data in such an application is heavily guarded by all kinds of business rules and the application will typically behave differently depending on that data.

7.1.5 Instrumentation

In the re-implementation, besides language-level error messages, also model-level errors can be given, because the abstractions from the model allow for that. Error messages thrown can speak of model concepts instead of (low-level) code constructs. The model has notion of invalid constructs (for example, cyclical connections) and the model may be checked during compile-time already. This trick is actually facilitated by the two-steps generation process, in which first the model source code gets generated (error messages on model-level possible, as depicted in Figure 7.3), and next, the resulting code gets compiled (error messages on the language level).

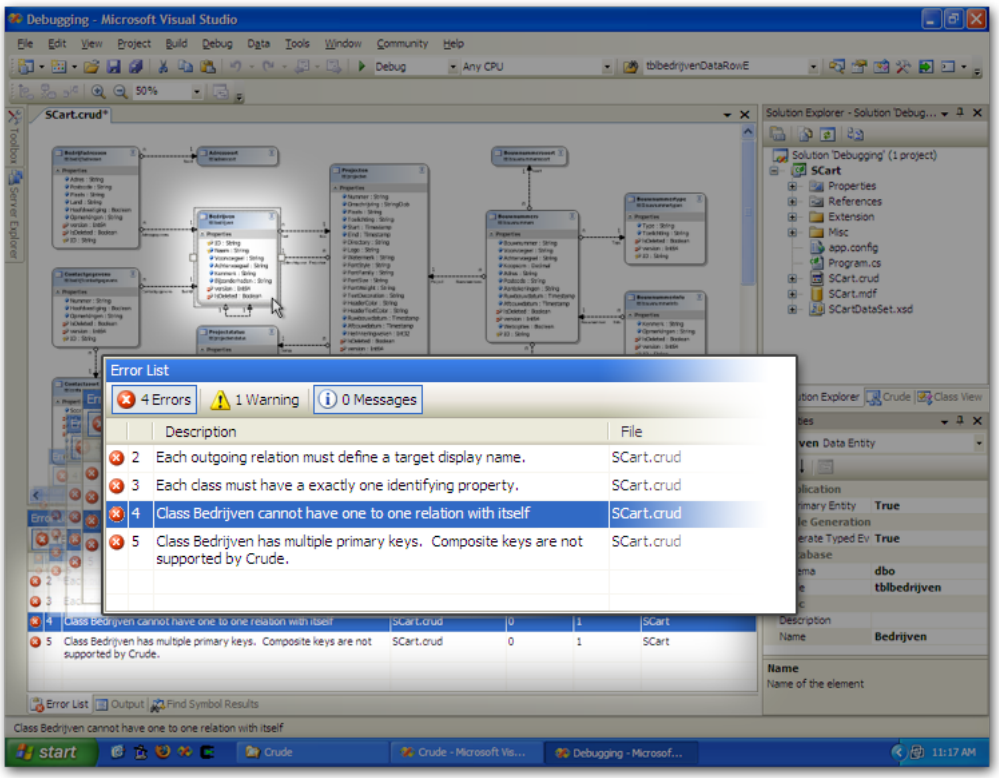


Figure 7.3: Model checks allow for conceptually meaningful errors during compile-time.

When using C# solely, the only thing there is is that low-level C#-code implementation, so there can not exist a notion of model concepts. The implementation of such domain concepts

can probably be found encoded in C#, but to check semantic validity of object structures, these must first get instantiated, which is of course done only at run-time.

A big advantage lies in the intermediate step in which model-level errors may be detected (high-level of abstraction). This advantage is a crucial advantage of model-driven design.

7.1.6 Modularity

The modularity of both applications is rather medium. Neither of both applications is designed with high modularity in mind.

The original application is more of a flat implementation of the requirements, not showing directly which objects are responsible for which application logic. For example, initially, it is not clear which class is responsible for the logic of committing a row just edited back to the database. When we try to locate exactly this logic, it turns out to be implemented inside the `Click` event handler for the "OK" button on the specific edit dialog window. This reduces application modularity, because responsibility of modifying data is now mixed up with GUI logic—if we would like to edit data without invoking a dialog window, we would have an extendibility problem.

In the re-implementation, we took care of this and followed an approach with higher modularity. We made the base window, the caller of the edit window, responsible for the actual commit to the database. (The dialog still remains responsible for the *validation* of the data.) In this scenario, logic is positioned at the conceptual object responsible for the given task.

It should be noted that it is important to include CRUDE's modularity in the comparison, too. Since CRUDE's semantics are fully implemented in the code generator, all of the generated application's functionality is contained in that code generator's logic. Since the code generator exists of only one class, and although invoking some helper classes, all application logic to be generated is contained somewhere in this huge generation process. Although the *output* of the generation process is quite modular, the code generator itself is very far from that. Its target domain is a C# file containing all logic, so every generation step involves C# code output, instead of logical functionality.

Although CRUDE's code generator is set up to follow the structure as much as possible by dividing the generation process into methods, each generating a specific part of the application, in essence the generator's output remains a code structure, which differs from the application structure. All functionality therefore is "encoded" in the generation of C# code, but there is no structural way of telling which code generator method implements a specific part of the application logic.

We will illustrate this with an example. To tell which part is responsible for the committing of the data to the database, we should first "translate" this to the structure of the code that would hold this functionality. Since we want to commit the data upon an edit or insert action, we can make an educated guess that the `GenerateInsertOrUpdateMethod` will probably hold the logic to generate this functionality. In fact, it does, among other functionality, which makes it hard to locate the logic we are looking for. This affects modularity pretty negatively.

We made extensive use of code regions, as supported by Visual Studio's code editor, to guide

the reading process of the verbose code (see Listing 7.1), which improves readability somewhat. Reading the guiding region comment in line 29, we may know that the commit code is contained in that region. Yet, this is as close as we can get to locating functionality.

```
1  private void GenerateInsertOrUpdateMethod(DataEntity de, bool insert,
2                                            CodeStatementCollection s)
3  {
4      First, get the current row (variable: 'row')
5
6      Next, create the dialog window
7
8      #region Show the dialog window and handle the data
9
10     // Ex: dialog.SetData(row);
11     s.Add(new CodeMethodInvokeExpression(
12         new CodeMethodReferenceExpression(
13             new CodeVariableReferenceExpression("dialog"), "SetData"),
14         new CodeVariableReferenceExpression("row")
15     ));
16
17     // Ex: if (dialog.ShowDialog() == DialogResult.OK) { ... }
18     CodeConditionStatement editSuccessful = new CodeConditionStatement();
19     editSuccessful.Condition = new CodeBinaryOperatorExpression(
20         new CodeMethodInvokeExpression(
21             new CodeVariableReferenceExpression("dialog"), "ShowDialog"),
22         CodeBinaryOperatorType.IdentityEquality,
23         new CodeFieldReferenceExpression(
24             new CodeTypeReferenceExpression("DialogResult"), "OK")
25     );
26
27     Attach any possibly linked parental tables
28
29     Invoke registered event handlers and commit/reject the data
30
31     #endregion
32 }
```

Listing 7.1: Code regions guide the reading process of verbose code generation logic.

7.1.7 Self-documentation

Self-documentation is concerned with the ease of which a mental model of intent can be distilled, given a snippet of source code. Actually, this is the reverse of the modularity discussion as discussed above—there the measurement involved the finding of the concrete implementing code snippet for a given piece of intent.

SCart is a low-level implementation in C#, existing of merely some source code. This code is only available on a low-level and thereby is far from self-explanatory on the application-level. A piece of code can be looked at in isolation by best effort, while the bigger picture is necessary to be self-documented. In Chapter 2 we saw already that [30] stated that source code is an awfully bad vehicle for intent. In practice, annotated comments are used to guide the human code reader, clarifying the functionality of subroutines without the necessity of reading through all bits and bytes. But, as we already saw, this approach is also subject to

its tendency to get outdated quickly [57, 63, 73].

In the re-implementation, on the contrary, a high-level overview is available. The guiding model is self-documented to a high degree, especially because its a visual one. When given a detailed code implementation for a specific part of the application, the corresponding part of the high-level model can be quickly derived from that—the code method names are directly derived from the model object names, so the link can be easily made. The application can be "read" both on a high as well as a low-level—the intent can be distilled from source relatively easy.

A note must be made regarding the self-documentation of the CRUDE code generator. As also stood out from the modularity discussion, the structure of the code generator is not very well suited to describe or carry intent. We extensively use guiding annotated, but informal, comments and code regions to structure the code to the user, which in practice proves very useful, but makes the human readers of the verbose code very dependent on it. This very negatively influences self-documentation.

7.1.8 Simplicity

In SCart, you always have to deal with the source code of the application. "Seeing through" the source code is per definition harder than with the re-implementation. In the re-implementation, the model is guiding—it provides an overview of the high-level structure of the data entities and generates all basic functionality of the application, which makes that this code body does not have to be fully understood or inspected to grasp the application. This basic functionality in the re-implementation is something that may simply be assumed to exist and correctly is implemented. The grasping of extensions to this basic functionality is very easy since extensions are readable apart from the generated logic, which essentially makes the code can be inspected in a very isolated setting. Extensions made in SCart are woven into the code, on the contrary.

In the re-implementation there does not exist a one-to-one relation between the model and the generated code. The extension of a data entity in the model has the consequence that the generated code gets extended in several places. There are extra methods generated, specifically for showing this data entity, visualisation logic for all its relations (incoming and outgoing), standard data checks are generated, a visual dialog window is generated, and so forth. Because naming conventions of all of this functionality always refer to the originally modelled data entities, the link back and forth between the high and low abstraction levels can easily be constructed. It is very clear which pieces of functionality exist for what reasons. Looking at the generated code, this is also simpler than that of the original SCart source code. In SCart, this high-level overview does not exist, which makes the SCart code more complex to understand.

Naturally, this simplicity comes at the cost of increased complexity on the development side of the DSL—making extensions to the DSL is more complex than extending the SCart source code with exactly the same functionality. Firstly because of the imperative way of specifying the code structure and secondly because of its descriptive character, which reduces simplicity significantly (mainly due to decreased readability).

7.2 Interpretation

Table 7.1 on the facing page shows a short overview of the comparison as made in the previous sections.

Relating all of this back to the original business challenges, through Table 1.1 and Table 6.5, we may specifically assess which business challanges are affected and what the assessment means for the specific business challenge. This is shown in Table 7.2. Some of the criteria are not applicable (due to Table 6.5) since they are of no influence to the criterion—these are marked “n/a”.

Looking at Table 7.2, we may observe a striking difference in applicability of software factories. The first business challenge, dealing with changing requirements, is pretty negatively affected by the use of software factories, while the second, enabling structural reuse, benefits much from it. The contrast is very big, but rather explicable.

Most plausible, the dealing with requirement shifts involves the ability to modify application development once most design decisions have already been made. Since design decisions in the DSL code generator are very verbose to write and target the full product-line, changes to exactly this are very hard to incorporate afterwards.

On the contrary, there is the challenge to enable structural reuse when it comes to the core of a whole product-line of applications. As we have illustrated, using a DSL to generate those applications costs almost no effort—the hard work on the creation of the DSL pays off here. Reusability is therefore greatly affected positively.

Criteria	Indicators	SCart	Re-implementation	
			CRUDE	Re-impl.
Conciseness	▷ Percentage of code lines written manually.	96%	21%	1.7%
Consistency	▷ Extent to which the tooling forces the programmer to produce consistent output.	Not	n/a	Much
	▷ The extent to which the programmer might still abandon from that in special cases.	Much	n/a	Not
Expandability	▷ Impact of changes for an architectual extension.	Medium/Severe	n/a	Severe
	▷ Impact of changes for a procedural extension.	Medium	n/a	Medium
	▷ Impact of changes for a data extension.	Medium	n/a	Little
Generality	▷ Diversity of domains the product can be used for.	Narrow	n/a	Only CRUD
Instrumentation	▷ Semantic level of *compile-time* error messages.	C#	n/a	Domain + C#
	▷ Semantic level of *run-time* error messages.	Domain	n/a	Domain
Modularity	▷ The ease of which logical units of functionality (intent) can be located in the implementing source code.	Medium	Hard	Easy
Self-documentation	▷ The ease of which logical units of functionality (intent) can be extracted from implementing source code.	Hard	Hard	Easy
Simplicity	▷ The amount of knowledge required to *understand* the application's source code.	Medium	High	Very low
	▷ The amount of knowledge required to *modify* the application's source code.	Medium/High	High	Low

Table 7.1: The result of the comparison. Where CRUDE is of any influence on the indicators, its assessment is mentioned, too. Red values are close to the worst-case value, green values are close to the best-case values, orange values are somewhere in between or of no significant meaning (see Table 6.6).

Criterion	Dealing w/ Changing Requirements	Enabling Structural Reuse
Conciseness	A more concise code base will imply that less code has to be changed to reflect a changing requirement.	The more concise code base will promote structural reuse either through library routines or generated duplication.
Consistency	More consistent code may imply more rigid code—it will possibly decrease the ability to deal with changing requirements.	Higher consistency will mean less "exceptions to the rule", which makes code suitable for structural reuse.
Expandability	Extending the application is much more costly when it comes to changes that require changes to the DSL code generator. Contingencies can not be easily dealt with.	Extending the reusable code base is bound to the places that are considered beforehand. Contingencies can not be handled easily.
Generality	The DSL is targeted at a specific domain (by its very nature). The lack of generality makes dealing with changing requirements very hard. Changes, then, are very costly.	n/a
Instrumentation	n/a	Increased instrumentation makes the IDE a more powerful tool during development, due to its increased level of semantic validation. This power comes freely to any project reusing the DSL.
Modularity	New requirements are hard to incorporate in the DSL due to the rather low modularity of the code generator.	Using DSLs does not influence the modularity of their output. This is determined by the DSL programmer's skills. High modularity is possible, but not a result of using DSLs as such. Reuse, hence, is not affected.
Self-documentation	n/a	Increased self-documentation of reusable components leads to better understandability—each piece of code can be directly related to the high-level application model.
Simplicity	Increased complexity of the DSL code generator increases the effort of dealing with requirements.	Increased simplicity on using the DSL to generate a firm amount of the application code stimulates reuseability.

Table 7.2: Relating back the results to the business challenges, by criterion.

Summary

The comparison made in this chapter is essentially based purely on the proposed indicators for the various quality criteria. Each indicator is used to provide a relative judgement of the impact or applicability of the criterion to both the implementations. Finally, we have interpreted these findings, resulting in a list of implications for using software factories, judging the relative positive or negative attributes.

Finally, from the comparison in this chapter, we may now draw some conclusions and pinpoint specific issues for which further research may be required or may be interesting, which is addressed in the next chapter.

8 Conclusions

In this Chapter, we will discuss the contributions of the research and recapitulate what we have established by way of answering the research questions from Section 1.6 explicitly. Next, we will place these results in their context and finally offer some suggestions for further research.

8.1 Results

We will start off answering the posed research questions. Each of the sections below represent one of the main research questions. The sections are all built up answering all of the sub questions first, followed by the integral answer to the matching main research question.

8.1.1 Development Implications

We have identified the circumstances under which software factories might be optimally used. The first benefit is the size and range of the target domain. A well-known, familiar domain makes that much more knowledge is available to the level of implementation detail necessary for a domain solution. This makes the refinement steps to take smaller, less complicated and straightforward. Also, a technical domain is preferable, because this means that more implementation details are available. From the re-implementation, the "high-level" model still has a relatively technical character—it concerns database implementation details. Generation is useful for low-level details. Lastly, the more specific the DSL output domain, the more applicable a software factory becomes. Too much variation makes the model unwieldy because of variability growth—the model becomes increasingly complex.

Secondly, software factories are optimally used in a relatively static environment. The smaller the change that changes occur in the environment requirement wise, the more applicable software generation becomes. Environments that frequently change program requirements—like legislation or relatively young environments in which rules of the game are not clear yet—are particularly unsuitable to be developed with software factories.

Finally, when a lot of repetition of code or development tasks occurs either within the same application (e.g. as with SCart), or among several programs of the same form (e.g. multiple

CRUD applications), and routine libraries do not suffice, code generation using software factories becomes increasingly applicable and worth the consideration.

In Chapters 2 and 4, we have identified the steps that should be taken in the process of *creating* a software factory. From literature, we have discussed the backgrounds of the software factory philosophy and its application area and we have outlined which steps can be distinguished in the creation of a software factory. A commonality and variability analysis can be used for the design and implementation of a DSL, which in turn is put to use to generate applications. The analysis is illustrated in Chapter 3, and applied to the creation of the factory in Chapter 4, using real examples. When extending the software factory with functionality, the commonality and variability analysis must be applied implicitly each time again, asking the question whether this bit of functionality is either an explicit fixed property of the product-line (a commonality), a variable property within the product-line (a variability), or belonging to a specific instance of a product out of the product-line.

In Chapters 2 and 5, we have identified the steps that should be takes in the process of *using* a software factory. Taking the DSL from Chapter 4, we can define a model from which the core of the application is generated which we can extend during the development of a specific product instance. We have shown, with a real example, that this extension is possible to extend at places that are defined beforehand in the DSL. When an extension can not be expressed or implemented in a product-specific extension, one must fall back on an extension of the DSL itself.

During the development of software using factories, one must pay specific attention to the process of generating ranges of applications, instead of the direct implementation of a specific instances. This makes that the mindset of the programmer of a DSL is more heavy than that of traditional programmer. The intellectual effort shifts in this process almost fully towards before the deliverance of the DSL. After the DSL is delivered, a strong decrease in intellectual effort is necessary to use the DSL to generate applications.

8.1.2 Effect on Dealing with Requirement Shifts

In Chapter 1, the possible causes behind the business challenge of dealing with (last-minute) requirement shifts that Sogyo faces have been analysed and proposed. We have looked at the limitations that influenced the challenge most and we have established that the programming paradigm and the software engineering methodology were the factors most influential. Then, in Chapter 6, we have determined, based on literature, which quality factors are of influence to the causes as identified in Chapter 1. The foremost quality criteria that relate to these factors, as we have distinguished them in Chapter 6, are conciceness, consistency, expandability, generality, modularity and simplicity.

In Chapter 7, then, a comparative study of quality has been carried out, based on our experience with the practical tooling. For each of the mentioned quality criteria, we have assessed how that criteria affects the ability to deal with changing requirements, either positively or negatively. From that assessment, it clearly shows that the foremost contribution of these criteria to the challenge is extremely negative. Only the “conciseness” criterion has a positive contribution, the rest is negative, in some cases even very native, and even decrease the opportunity of dealing with requirement changes.

When we look at solely this business challenge, we may well state that software factories are not a solution to the challenge and even pose a risk to Sogyo's conduct of business.

8.1.3 Effect on Enabling Structural Reuse

In Chapter 1, the possible causes behind the business challenge of enabling structural reuse that Sogyo faces have been analysed and proposed. We have looked at the limitations that influenced the challenge most and we have established that the programming paradigm and the IDE were the factors most influential. Then, in Chapter 6, we have determined, based on literature, which quality factors are of influence to the causes as identified in Chapter 1. The foremost quality criteria that relate to these factors, as we have distinguished them in Chapter 6, are conciceness, consistency, expandability, instrumentation, self-documentation and simplicity.

In Chapter 7, then, a comparative study of quality has been carried out, based on our experience with the practical tooling. For each of the mentioned quality criteria, we have assessed how that criteria affects the possibilities to enable structural reuse, either positively or negatively.

That assessment clearly shows that the foremost contributions of these criteria to the business challenges are very positive. Besides "expandability", each of the criteria has a positive contribution to the business challenge and greatly improve chances of enabling structural reuse by using software factories.

Solely looking at the business challenge of enabling structural reuse, we may state that software factories are a serious candidate for it.

8.2 Reflection

The results form the previous section should be put into perspective in order to interpret them correctly. This reflection will explain how the research results should be interpreted and where they apply. The scientific contributions of this research to the theories of software factories are mostly of practical, examining nature. The form of it is a case study—a specific assessment of an exemplary scenario.

This form yields the following. First, it holds a practical example of how software factories can be used, both for Sogyo as well as for a scientific audience. For Sogyo in particular, the example is based on a well-known application, developed by Sogyo itself. Secondly, the research may be used to get a notion of the possibilities and limitations of the tooling used. Finally, practical findings of this research are directly related to and held against existing theories. The re-implementation, by the way, is never intended to be fully functional and should not be regarded that way. Its goals is mainly illustrative. It is used to support, confirm and justify theoretical foundations from software factory literature in the research.

This form of research has the consequence that the judgement that is formed of software factories is based on only one case, and therefore can not be regarded as the absolute truth. The judgement is the judgement of a specific tool (DSL Tools), for the implementation of a specific application kind (CRUD applications), used in a specific setting (Sogyo Development), focused merely to its contributions to specific business challenges.

Of course, some general findings can be deduced from this specific scenario. But if doing so, there always has to be noted that under all conditions further research is required before scientific value may be assigned to it. Until that moment, they remain hypotheses. Any hypotheses based on the findings in this research, can serve as a starting point for further research initiatives. The current research must not be interpreted as a fundamental theoretical contribution to software factories foundations, nor must it be regarded a practical judgement of software factories as such.

Regarding the comparison framework for software quality, something different holds. This framework is a specific extension, built on existing general software quality literature, for the assessment of relevant quality criteria of software given some business challenges. Therefore, it can be used as a (theoretical) guideline for the development of other such frameworks, typically for assessments within other business settings. The framework's criteria are quite complete, considering the quality aspects are taken from existing software engineering and software quality literature that is established over a vast amount of years. However, future work may focus on the development of better measurement instruments or indicators.

8.3 Managerial Recommendations

Some practical implications follow from this research. The recommendations depend on the audience. We will discuss the implications for Sogyo, any other potential adopters of the software factories and for the scientific audience.

8.3.1 Sogyo

The main message to distil from this research for Sogyo is to be sceptical towards the current level of maturity of DSL Tools with respect to the generation of software. We have made clear that generation using software factories merely contributes to one of the two business challenges that Sogyo faces. When the domain is more stable (i.e. when requirements are not subject to heavy change rates or change impact is low), the use for software factories will increase and Sogyo might consider software factories.

Another consideration that must be made, is whether software factories can be of use when only small pieces of the eventual software product can be generated—pieces which then can be put together by humar labour. Despite the conclusion that DSLs are not suited to generate full application source code, we should not throw out the baby with the bath water. DSLs will probably come in handy for the generation of smaller pieces of functionality.

In this research, we have focused on the generation of a full application implementation, including a fully functional GUI. The case may be that the output of the factory should not be a fully functional application, but rather a supportive source code asset, such as a routine library. In those scenarios, the user of the DSL will require more explicit knowledge of how to wield its output, but will remain more control over it at the same time. This is an interesting field of tension and there must be found a way to cope with it in a pragmatic fashion.

DLSs might pose to be a particularly interesting solution if used to automate small pieces of a larger piece of software (small libraries, wrapper classes, configuration files, etc.). Would we have approached the SCart application that way, then we would imaginably have set up

the code generator to generate controls, database logic, etc. without tying them together with application logic. Although that would be a small step forward, it would not have the impact of the giant leap that we take now, automation-wise. Instead, given the automatically generated edit controls, data extraction routines, etc., the putting-it-all-together would still require lots of tedious work. Remember that there are 58 edit dialogs in the original application only, not even considering difference in insertion and modification dialog windows.

In the SCart case, it is exactly *this* manual work that is automated, not the generation of edit controls or data access logic. Although undoubtedly there is more work that might be a candidate for generation, looking back, the generation of a fully functional application may have turned out to be too opportunistic and too rigourous an approach. Finding the optimal balance in this will require a more in-depth research.

Also, the software factories technology of Microsoft is only a very young technology. Since the start of this research in September 2006, one major and one minor release of the Visual Studio SDK have been published[†], each of which contained a lot of changes to the DSL Tools software, which broke compatibility of old models. Since the technology itself is under so heavy development, conclusions drawn in this research may not be preservable as the technology matures.

Even more so, there is extensive literature available on domain-specific languages, but there was only little scientific literature available on the topic of software factories and visual domain-specific languages as tool support for generating software. Even more so, documentation on DSL Tools by Microsoft was extremely modest and mostly informal of character. Especially on the topic of extending the DSL Tools modeling framework, there is no well-documented knowledge available—here is a lot of work to be done by Microsoft. Much of the work for this research regarding the extension of DSL Tools has been done so through trial and error.

There are significant business risks involved in using tools that are that minimally documented and are still subject to change.

Finally, we have shown that coming from the illustrative re-implementation made for this research to a fully implemented one, usable in practice by Sogyo customers, is not trivial. In particular, recall the discussion on primitive versioning in Section 5.4.1 (page 88). There is still an extensive amount of work to be done to support such a crucial element of the software. Involving in software factories therefore is not recommended to Sogyo at this moment.

8.3.2 Potential Adopters

Software factories accommodate a powerful concept, but it should not be overrated. A lot of the power stands or falls with the maturity of the tooling. One thing this research has made clear very explicitly is that software factories are extremely useful for facilitating structural reuse. Potential adopters should embrace software factories if their environment resembles a relatively stable environment in which structural reuse is an explicit desirable situation.

[†]First, the September 2006 CTP ("v3") release included DSL Tools for the first time. In the November 2006 release, significant improvements to DSL Tools were made. These broke existing DSLs. Next, in the February 2007 ("v4") release, DSL Tools was updated with some bug fixes.

If software factories are a serious consideration for adopters, there must in particular be paid attention to other environmental requirements that software factories may have their influence on, but which are not covered by this research. More on this is discussed in Section 8.4.

Practical “imperfections” should be taken into account. More on this can be read in the next section. The recommendations to the scientific audience may therefore be well suited to any potential adopters, too.

8.4 Further Research

This study is among the first scientific studies carried out regarding software factories. And although there exists literature on domain-specific languages, that literature mostly covers textual languages, not visual ones. This research is carried out largely by a case study and a practical comparison of two software implementations. This implies that the study can not possibly have discussed all facets of software factories in detail. Although effort has been made to provide an as comprehensive overview as possible with the current knowledge on the topic, both by a study of available literature and through practical experience, further research is advised to be carried out to complement the findings of this research.

In particular, Chapter 1 makes some assumptions with respect to the underlying causes of the business challenges. There is no scientific evidence provided in these reasoning steps, but they are rather collected from Sogyo employees and supervisors. Since these “assumptions” are rather plausible, demonstrating their validity fell beyond the scope of this research. Further research may confirm these assumptions. A similar situation is found in the discussion in Chapter 6, were quality factor strengths are assigned to the causes.

Another implication is that this study has left some topics unattended. Although software factories have turned out not to be the solution for Sogyo, there clearly is a case for them. This research provided the insight that software factories are an explicit candidate for promoting structural reuse within a software company. Undoubtedly, there are more such challenges where software factories may contribute to. Further research can be carried out to bring more specific strong and weak points of software factories to the light. Might such initiative be started, then this thesis may provide a framework for the set up of such assessment.

From a scientific point of view, software factories are no new technology. They pose a practical aid for the generation of (pieces of) software, but the process is mainly unguided. Semantics are not strictly coupled to the syntax. Creating the abstract syntax creates C# classes, which the code generator may or may not use. There is no coupling in the sense that changing the abstract syntax requires a change in the code generator. This has important consequences.

First of all, one can not show, let alone prove, properties of the generated software (e.g. correctness, completeness, etc.). This is of course the result of the absence of coupling of the syntax with the semantics.

Among others, a two-way synchronisation between the model and the generated code is not possible during DSL usage (e.g. application development) time. Exactly that gap should have been bridged by the code generator, but Microsoft’s implementation is a one-way synchronisation. Using DSLs, after the code generation (when the model gets saved), the user is left

by himself.

Research initiatives concerning the possibilities to make a tight coupling between syntax and semantics for visual DSLs in the tooling is greatly encouraged. Essentially, DSL Tools should profit from such a tight coupling. Just like coupling abstract and concrete syntax, semantics should also be promoted a vital part of the language design and implementation. Ultimately, in such scenario, the end user may switch back and forth between model and source code and keep them in sync. This might be compared to the way the GUI editor in Visual Studio already is capable of such synching. Adding such support will make DSL Tools relevant scientifically speaking.

Software factories were a presumed potential for contributing to Sogyo's business challenges. This research's focus was not solving those business challenges, but to assess to what extent software factories would. Alternative solutions should be researched further.

One thing will continue to hold, however. Technologies may lower the load of the programmer's mind, but no technology as such will ever be the key to creativeness. That remains a human matter.

List of Acronyms

CASE	Computer aided software engineering
CPU	Central processing unit
CRUD	Create, read, update, destroy
DDD	Domain-driven design *(or domain-driven development)*
DOM	Document object model
DSDM	Dynamic systems development method
DSL	Domain-specific language
GPL	General purpose language
GUI	Graphical user interface
IDE	Integrated development environment
MDD	Model-driven design *(or model-driven development)*
OO	Object oriented
PK/FK(-relations)	Primary key/foreign key (relations)
RAD	Rapid application development
RUP	Rational unified process
R&D	Research & development
SCV(-analysis)	Scope, commonality and variability (analysis)
SDK	Software development kit
SF	Software factories
SQL	Structured query language
TDD	Table-driven design *(or table-driven development)*
UI	User interface
UML	Unified modeling language
VM	Virtual machine
XP	Extreme programming

Bibliography

[1] L.J. Arthur. *Measuring Programmer Productivity and Software Quality*. John Wiley & Sons, Inc. New York, NY, USA, 1985. 6.1.1

[2] M.R. Barbacci et al. *Quality Attributes*. Carnegie Mellon University, Software Engineering Institute, 1995. 6.1

[3] K. Beck and M. Fowler. *Planning EXtreme Programming*. Addison-Wesley, 2001. 1.3.5

[4] J. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8): 711–721, 1986. 2.6

[5] T.J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5: 169–226, 1998.

[6] James Kenneth Blundell, Mary Lou Hines, and Jerrold Stach. The Measurement of Software Design Quality. *Ann. Softw. Eng.*, 4:235–255, 1997. ISSN 1022-7091.

[7] B.W. Boehm et al. Characteristics of Software Quality. 1978. 6.1

[8] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1993.

[9] F.P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[10] C.M. Christensen. The Innovators Dilemma: When Technologies Cause Great Firms to Fail. *Harvard Business School Press, Boston, MA*, 1997. 2.1

[11] G. Coleman and R. Verbruggen. A Quality Software Process for Rapid Application Development. *Software Quality Journal*, 7(2):107–122, 1998. 1.3.5

[12] C. Consel and R. Marlet. Architecturing Software Using a Methodology for Language Development. *Proceedings of the 10 thInternational Symposium on Programming Language Implementation and Logic Programming, number*, 1490:170–194, 1998.

[13] The Cutter Consortium. Some New Data in the Project Failure Statistics Wars. *The Software Practitioner*, Nov. 2005. 2.1

[14] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 1998. 2.4.1

[15] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[16] E.W. Dijkstra. *Notes on Structured Programming*. Technological University, Dept. of Mathematics, 1970. 2.4.1

[17] E.W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF. 7.1.2

[18] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying Software Product-Line Architecture. *Computer*, 30(8):49–55, 1997.

[19] Dan Douglas. *Assessing Languages for Specific Purposes*. Cambridge University Press, 1999.

[20] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. 1.5, 2.5.2

[21] M. Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Pearson Education Inc, 2003. †, 5.4.1

[22] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? 2005. http://www.martinfowler.com/articles/languageWorkbench.html. 1.5, 2.6, †

[23] Martin Fowler. Introduction to Domain Specific Languages, 2006. http://www.infoq.com/presentations/domain-specific-languages—visited December 11th, 2006. 2.6, 2.6

[24] C. Fry. Programming on an Already Full Brain. *Communications of the ACM*, 40(4): 55–64, 1997. 1.3.6

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Inc., 1995. 2.3.2

[26] Nasib S. Gill. Factors Affecting Effective Software Quality Management Revisited. *SIGSOFT Softw. Eng. Notes*, 30(2):1–4, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1050849.1050862.

[27] S.E.C. Goldrei. The Design, Implementation and Use of Domain Specific Languages. 2004.

[28] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[29] Jeff Gray and Gabor Karsai. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. *hicss*, 09:325a, 2003. doi: http://doi.ieeecomputersociety.org/10.1109/HICSS.2003.1174892.

[30] J. Greenfield, K. Short, et al. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. 1.5, 2, 2.1, 2.2, 2.3, 2.3.1, 2.4, 2.5, 2.5.1, 2.6, 2.6, 2.6.2, 7.1.7

[31] The Standish Group. The CHAOS Report, 2004. 2.1

[32] The Standish Group. The CHAOS Report, 1994. http://www.standishgroup.com/sample_research/chaos_1994_1.php. 2.1

[33] J. Heering, M. Mernik, and AM Sloane. Domain-Specific Languages. *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 323–323, 2003.

[34] Jan Heering. Application Software, Domain-Specific Languages, and Language Design Assistants. *Centrum voor Wiskunde en Informatica*, 2000. Report SEN-R0010.

[35] RM Herndon Jr and VA Berzins. The Realizable Benefits of a Language Prototyping Language. *Software Engineering, IEEE Transactions on*, 14(6):803–809, 1988.

[36] J.A. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley, 2002. 6.2.2

[37] P. Hudak. Modular Domain Specific Languages and Tools. *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, 1998.

[38] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

[39] IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990. 6.1, 5

[40] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995. 2.3.1

[41] S.N. Kamin. Research on Domain-Specific Embedded Languages and Program Generators. *Electronic Notes in Theoretical Computer Science*, 14, 1998.

[42] JC Kelly and YS Sherif. Comparison of four design methods for real-time software development. *Information and Software Technology*, 34(2):74–82, 1992.

[43] B. Kitchenham. Towards a Constructive Quality Model: Part I: Software Quality Modelling, Measurement and Prediction. *Software Engineering Journal*, 2(4):105–113, 1987. 6.1.1

[44] A.G. Kleppe, J.B. Warmer, W. Bast, and A. Watson. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. †

[45] T.S. Kuhn. The Structure of Scientific Revolutions. *IL: University of Chicago*, 1970. 2.1

[46] P.W. Kutter, D. Schweizer, and L. Thiele. Integrating Domain Specific Language Design in the Software Life Cycle. *Applied Formal MethodsFM-Trends*, 98:196–212, 1998.

[47] C. Lengauer, D. Batory, C. Consel, and M. Odersky. Domain-Specific Program Generation. Number 3016 in Lecture Notes in Computer Science, 2004.

[48] G. Lenz and C. Wienands. *Practical Software Factories in .NET*. Apress Berkely, CA, USA, 2006. 2.1, 2.2, 3.6.1, 3.6.2

[49] J.A. McCall, P.K. Richards, and G.F. Walters. *Factors in Software Quality*. NTIS, 1977. 6.1, 6.1.1, 6.3

[50] M. Mernik, J. Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[51] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

[52] J. Miller, J. Mukerji, et al. Model Driven Architecture (MDA). *Object Management Group, Draft Specification ormsc/2001-07-01, July*, 9, 2001. †

[53] B. A. Myers. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *SIGCHI Bull.*, 17(4):59–66, 1986. ISSN 0736-6906. doi: http://doi.acm.org/10.1145/22339.22349. 1.3.6, 2.6.2

[54] Hanne Riis Nielson and Flemming Nielson. Semantics With Applications—A Formal Introduction. 1999.

[55] DL Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 2.4.1

[56] B.A. Philp and B.J. Garner. Knowledge Mediation in Software Quality Engineering. *Proceedings 2001 Australian Sofware Engineering Conference*, pages 153–159, 2001. 1.3.6

[57] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill International Ltd., fifth edition, 2000. 1.1, 1.3.5, 6.1, 6.1.1, 7.1.3, 7.1.7

[58] Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[59] R.W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2005. 2.6, 2.6.1

[60] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[61] Y. Smaragdakis and D. Batory. Application Generators. *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*, 2000.

[62] JM Smith and D. Stotts. Elemental Design Patterns: A Link Between Architecture and Object Semantics. *Proceedings of OOPSLA*, 2002.

[63] Ian Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2001. 1.1, 6.1, 6.1.1, 7.1.7

[64] D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *The Journal of Systems & Software*, 56(1):91–99, 2001.

[65] T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung*. dpunkt-Verl., 2005. 2.5.2

[66] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley Longman, Inc., second edition, 1997.

[67] M. Tatsubori, S. Chiba, M.O. Killijian, and K. Itano. OpenJava: A Class-Based Macro System for Java. *Reflection and Software Engineering*, 1826, 2000.

[68] R.N. Taylor, W. Tracz, and L. Coglianese. Software Development Using Domain-Specific Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–38, 1995.

[69] F. van der Linden. Development and Evolution of Software Architectures for Product Families. *Lecture Notes in Computer Science*, 1429, 1998.

[70] Arie van Deursen and Paul Klint. *Domain-specific Language Design Requires Feature Descriptions*. Centrum voor Wiskunde en Informatica, 2001.

[71] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *CWI (Centre for Mathematics and Computer Science)*, 1997. 2.6

[72] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000. 2.6

[73] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Inc. New York, NY, USA, 2000. 1.1, 6.1, 6.4, 7.1.7

[74] Piet Verschuren and Hans Doorewaard. *Designing a Research Project*. Lemma, 1999.

[75] D. Weiss. Creating Domain-Specific Languages: The FAST Process. *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL*, 97, 1997. 2.4.1, 3.6.1, 3.6.2, 6.2.4

[76] D. Wile. Supporting the DSL Spectrum. *Journal of Computing and Information Technology*, 2002. 2.6

[77] U. Zdun. Concepts for Model-Driven Design and Evolution of Domain-Specific Languages. *Proc. of the Intl Workshop on Software Factories at OOPSLA 2005*, 2005. 2.5.2