Chapter 15
Software Project Management for Combined Software and Data Engineering

Seyyed M. Shah, James Welch, Jim Davies, and Jeremy Gibbons

15.1 Introduction

Software engineering is an established discipline for the systematic creation of large and complex software systems. Relatively more recent are attempts to systematise the creation and management of large data sets, in data engineering. A key feature of these disciplines is managing engineering processes using several stages that form a development *life cycle*. This leads to a methodical process for development and separation of work into modular elements. A challenge when using this approach is integrating software engineering life cycles with the wider context of the software, for example, business processes, user requirements or indeed other codependent development life cycles. This chapter is on combining software engineering with data engineering to enable better project management, foster reuse and harness best practice from two seeming disparate domains.

The presented methodology is created from experiences on the ALIGNED project, a large, interdisciplinary research project that applies state-of-the-art software and data engineering techniques for the development of real-world systems. Several use cases have been identified that are challenging due to a lack of coherence between the software and data engineering, from collection and management of anthropological research to cataloguing and exploration of jurisprudence data and machine interpretation of large encyclopaedic data sets.

In the project, several state-of-the-art tools are under development from both domains such as Booster [1], the Model Catalogue [2], RDFUnit [3], Repair Framework and Notification (RFN), Ontology Repair and Enrichment (ORE), Dacura [4], the PoolParty Confluence/JIRA Data Extractor (CJDE) [5] and External

S.M. Shah (✉) • J. Welch • J. Davies • J. Gibbons
Software Engineering Group, Department of Computer Science,
University of Oxford, Oxford, UK
e-mail: seyyed.shah@cs.ox.ac.uk

Link Validation (ELV) and the Unified Governance Plugins (UGP) [6]. The methodology described in this paper is used as an initial framework to support the combination, development and application of these systems, to address the project goals. These tools would not ordinarily or easily be use or developed in unison, due to the differing approaches of software and data engineering.

The importance of the presented methodology is clear: by approaching software and data engineering life cycles coherently, efficient reuse of artefacts and applying established best practice from both domains become possible. This chapter discusses the issues of integrating the two fundamentally differing approaches to building systems. The data engineering approach broadly involves building tools that act on a wide range of data, while the approach taken in software engineering is to restrict and enforce the acceptable data, making these two approaches difficult to combine in practice.

There are several further reasons that make a combined methodology impractical. Each of the life cycles can have multiple stages, which can vary across life cycles. Creating a single methodology that encompasses all of these stages is problematic because there are many possible ordering and interleaving of process steps and there are a range of possible software and data engineering life cycles that a project may use. Furthermore, software engineering life cycles tend to take a prescriptive approach to software development, whereas data engineering life cycles tend to be descriptive. The two approaches also diverge on handling unexpected data, as data engineering tools tend to filter from a large range of inputs, which leads to failure-tolerant systems, while the fail-fast or fail-safe approaches of software engineering mean systems are expected to halt on unexpected data or deal with specific classes of error. These differences mean that dependencies between the two approaches can lead to duplication of effort, complexity in the project planning and incompatible systems due to the differing development philosophies.

Instead of a single monolithic methodology, this chapter presents a combined *meta-methodology*, which consists of a lightweight, descriptive method for combining and pairing of software and data engineering life cycles. The methodology consists of a matrix of synchronisation points that are defined on a project-by-project basis and informed by the opportunities and requirements for reuse between two loosely coupled life cycles.

This chapter is structured as follows: Sect. 15.2 presents an introduction to established software engineering and data engineering project life cycles. Section 15.3 contains a discussion on the major issues with a combined methodology. In Sect. 15.4, the methodology is presented in essential form, which consists of a flexible generic mechanism to track synchronisation between separate project life cycles. The tools and use cases of the ALIGNED project are analysed post hoc using the methodology, to form the initial evaluation of the methodology in Sect. 15.5. In Sect. 15.6, a discussion of related work is presented. The chapter concludes with plans to further apply, develop and evaluate the methods described.

15.2 Software and Data Engineering Project Life Cycles

There are several software engineering methodologies that could be considered for the alignment between software and data engineering. The subset presented here is not intended to be complete; instead an overview of some prominent and notable methodologies is presented.

In the waterfall model [7], the development process is seen as a series of downwards flowing, strictly linear steps towards an end software product. The ordering and kinds of steps can vary, but typically include “analysis”, “design”, “implementation”, “validation” and “maintenance”, as depicted in Fig. 15.1. The process mimics the traditional process for large-scale physical engineering projects. This methodology was first presented as an example of how not to develop software; it is notable for observing the effects of an ordered set of distinct steps for software development. Later adaptations addressed the rigidity by allowing earlier steps to be revisited. Several derivative methods have since been developed including the incremental waterfall, evolutionary model, spiral model and structured systems analysis and design method [8].

Rapid application development (RAD) [9] and prototyping promote the role of early prototyping to inform and shape further development iterations. Rapid application development was proposed in response to the cumbersome, inflexible waterfall-like techniques, but is no longer used as a stand-alone methodology. Agile methodologies include techniques such as test-driven development, extreme programming, feature-driven development and scrum. These so-called lightweight methods focus on delivering useful software frequently, closely working with users, simplicity and adaptability to change. The principles are further set out in the Agile

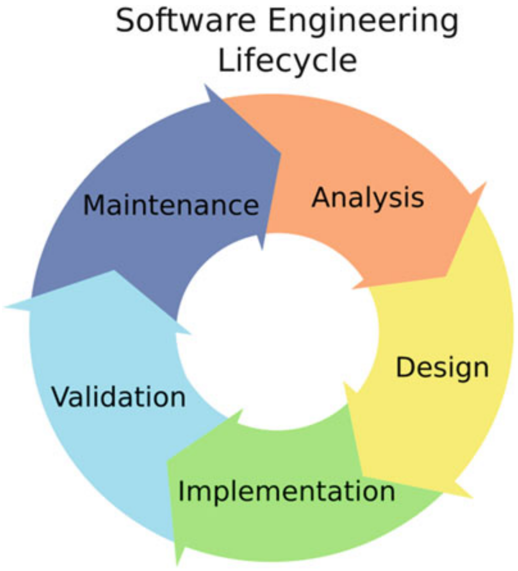

Fig. 15.1 Typical software engineering project life cycle stages

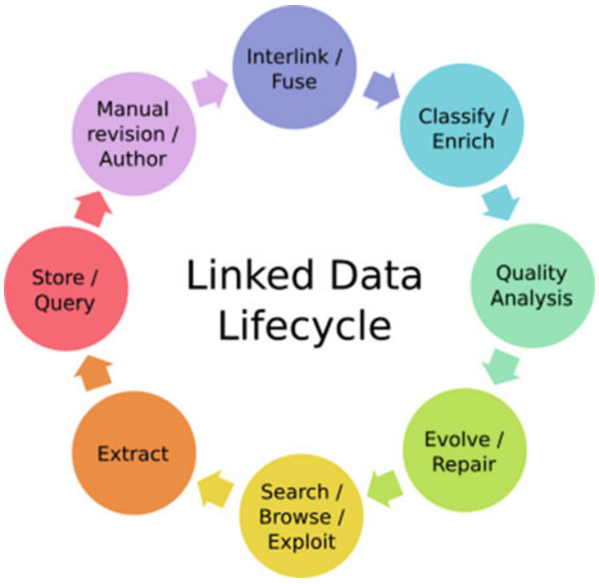

Fig. 15.2 Linked open data life cycle adapted (Adapted from Ref. [12])

Manifesto [10]. Essentially, the agile methods are designed to be reactive to the needs of the users, rather than relying on rigid one-off steps or specific languages or processes. Agile tends to afford greater trust to developers and teams to define their own processes, architectures and the selection of appropriate languages and tools.

Model-driven software development [11] revolves around the creation of high-quality, detailed models at each stage of the development process. The method relies upon the availability of sophisticated tools that use models to derive software artefacts, by a process of transformation. For example, the transformations can be used to translate designs into software implementations and models into test cases. Model-driven development aims to bring a greater level of formality to each stage of development, which can mean new tools and languages must be developed. The methodology is closely related to the techniques found in formal methods, generative programming and computer-aided software engineering. Like agile development, it does not enforce a specific set of steps, tools or languages, but instead provide a general framework within which software can be developed.

The linked open data life cycle consists of seven stages for data engineering and is described in detail by Auer et al. [12]. Essentially, the eight stages shown in Fig. 15.2 relate to the capture and management of data to ensure the data is consistent with the linked data principles. The process can begin at any stage and stages may be omitted, so the *Extract* phase can be used to take information represented in unstructured form or conforming to other structured or semi-structured formalisms and map it to the RDF data model. Once there is enough data for the intended purpose of the data, tools are required for efficient *Store and Query* of the data in triple form. The next phase is *Author* to manually create, modify and extend the structured data. As there can be many publications and sources of information about the subjects in the store, *Link*s between data sets must be appropriately established. Since linked data relies on triples as the basic unit of data, information can be captured without the need for a high-level schema, *Enrich and Classify* is necessary to capture high-level structures, in order that data can be analysed and reusable queries can be performed. Importantly, the *Quality* of the data must be analysed and used for *Evolve and Repair*. Finally, once data has passed these phases, it should be made available for end users to *Search, Browse and Explore*, and the process can continue to refine and improve the data set.

15.3 Challenges for Projects Combining Software and Data Engineering

This chapter focuses on the alignment between software and data engineering life cycles for managing projects. This section sets out the key differences in the approaches as challenges for a combined methodology that is useful in practice.

Methodologies for software and data engineering are created for the separation of work into distinct stages during development and are sometimes referred to as life cycles, processes or methods. Each self-contained stage organises work into separate subtasks to be carried out in order to produce an end result. The stages of a methodology are usually followed linearly, and each stage results in one or more artefacts that inform subsequent stages; both are typically performed iteratively, such that the results of each iteration may be used and provide requirements for the next. Both engineering life cycles have similar goals: the management, manipulation and interpretation of data systems for eventual consumption by the end user; however, this task is approached from two divergent doctrines. Unsurprisingly, data engineering is data oriented—the data is the principal concern; in software engineering, it is the software programs that manipulate data that are the fundamental product. The software engineering life cycle results in a multi-author software artefact; in data engineering the result is a multi-origin data artefact.

While the software and data engineering life cycles produce different kinds of artefacts, there are several potential advantages of having an integrated methodology. Firstly, there is a scope for overlap in processes in each methodology. Software engineering techniques are employed to create software for data engineering (or applications that consume or manipulate the data artefacts), and data engineering processes often affect the software engineering life cycle. However, the two processes often run separately and in parallel to each other with few synchronisation points. Furthermore, artefacts produced in data engineering processes should ideally be reused in software engineering and vice versa, with reuse being a fundamental principle in both software and data engineering. This can in turn lead to software that more closely reflects the domain, and software more tailored to the data, at lower effort.

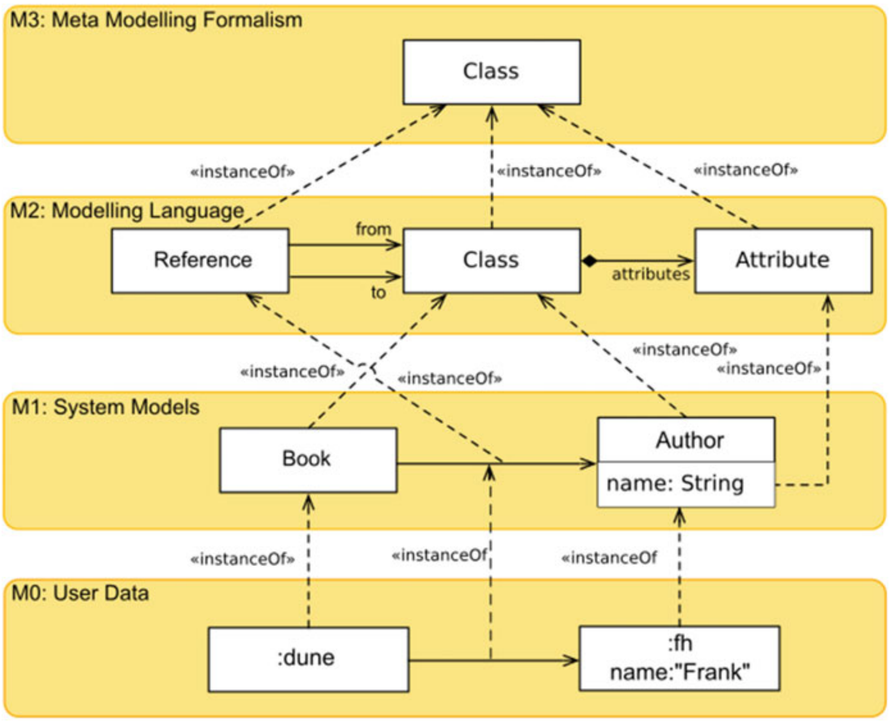

Fig. 15.3 Metalevel hierarchy in model-driven engineering

In metamodelling, data is stratified across so-called metalevels with each level used to define the primitives of the lower levels. So, instance or user data exists at level M0, which is defined in terms of the system models at level M1, which records the software and schemas to which instance data must conform. There are further metalevels, notably M2 for defining metamodels, where tools for software engineering processes are defined. Considering the metalevels, software and data engineering processes tend to be concerned with levels M1 and M0, but with differing approaches. Metalevels are a conceptual construct that simplify the definition and comprehension of complex software, by allowing separation of concerns. Existing software and data engineering data, programs and development tools can be expressed in terms of these metalevels, as shown in Fig. 15.3. Tools developed at the M2 level can be used and reused by developers at the M1 level where each new problem has common aspects that are dealt with elegantly by the M2 tools.

The approach of software engineering is to create software at the M1 level based on models that are fixed at development time and rarely change. The data that is accepted by the software is limited, and the software usually has well-defined functionality, which can involve quite complex data-dependent manipulation. This amounts to what is known as the "closed world" assumption [13]; although

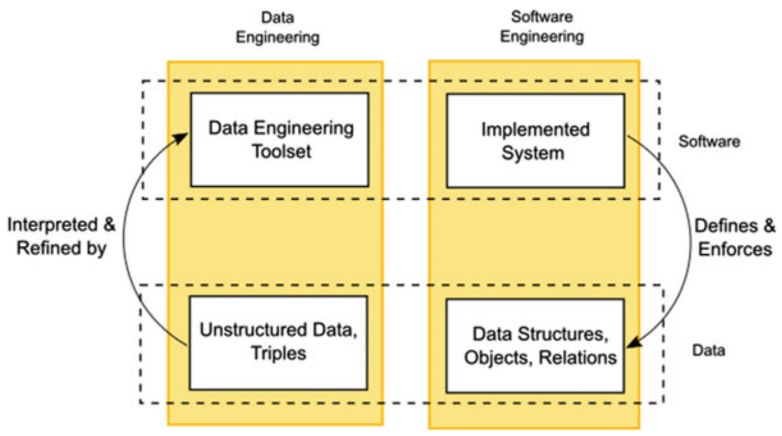

Fig. 15.4 Software and data engineering: approaches to software and data set development

it is possible to develop software without this constraint, this assumption is typical for commercial software. Tools engineered this way rely on high-quality data, and the premise is that the system should only act on fixed models and metamodels. This means that software will fail once data is encountered that does not conform to the expected model. Software engineering life cycles tend to be prescriptive with a set of procedures that must be followed in order, so a system developed using a methodology cannot be implemented until a design is in place.

Data engineering processes tend to use data-independent software tools that consume M1 level schemas, ontologies and vocabularies, to manipulate instance data in a generic way, at the M0 level. Tools to define and manipulate the schemas, ontologies and vocabularies are essential to data engineering. These tools tend to be less restrictive on the data accepted as input. Higher complexity in data processing and the ability to handle unexpected or low-quality data may require higher computational and engineering overhead; however, this usually results in tools that can be reused for unrelated data sets. The tools tend to be tolerant of low-quality data, in the sense that any unexpected data is ignored without interrupting processing. The methodology for data engineering tends to be descriptive—used more as a set of guidelines than a rigorous process, so, for example, classification and enrichment can be carried out without interlinking and fusing. The contrasting approaches of software and data engineering are shown in Fig. 15.4.

This dissimilarity between the two processes means that combining the two into a single monolithic methodology is problematic. A choice would need to be made, a priori, about prescriptive versus descriptive, fail-fast versus failure tolerant, the selection of a specific data engineering and software engineering “flavour”, the ordering and interleaving of process steps and so on. This would ultimately result in a methodology that favours the perspective of either software or data engineering and may even result in a worst-of-both-worlds scenario. The methodology presented here will not set out universal commandments, with multiple combined points of failure or a prescribed, redefined set of steps derived from either method. Instead, the aim is to try to integrate the best of both.

The methodology is based on the processes used in the ALIGNED project, where data engineering and software engineering are combined in practice. This means the methodology reflects on the observed practices and experiences of the ALIGNED consortium. This also means that the methodology may, initially at least, be specific to the tools and use cases of the ALIGNED project. Many techniques are used for data engineering and software engineering; rather than selecting and combining a single one of each, a more generic approach is suggested.

We propose the concept of an agile *meta-methodology*. This gives the software and data engineering practitioners a lightweight, basic method for combining their own methods and processes. This should, by design, be adaptable and reusable depending on the software engineering and data engineering context. The methodology provides practitioners with a system that is largely controlled by the methodology users and works practically on a project-by-project basis.

15.4 Methodology for Combined Software and Data Engineering

This section outlines the proposed *meta-methodology* for combined software and data engineering in ALIGNED. It is important to note that the present methodology is in under development and is expected to evolve and develop as the project goes on, based on the needs of the consortium and the project findings. Several areas of the method may be extended, based on observations of practice.

As a lightweight methodology, the technique requires some initial setup and maintenance by the software and data engineering processes. The main setup task for combined software and data engineering is to determine the synchronisation points between the software engineering and data engineering. A synchronisation point is at any pair of points in the two life cycles where specific artefacts and processes should be shared between software and data engineering processes. Once the synchronisation points are determined, they can be used during the project to resynchronise as development to the artefacts occurs.

In order to visualise and understand the synchronisation points between data engineering and software engineering, a table can be created that combines the two life cycles. The table allows the developers in both life cycles to determine where in each of the two life cycles synchronisation should happen. As shown in Table 15.2, the table columns are the software engineering life cycle stages, and the rows are the data engineering life cycle stages. The software engineering or data engineering life cycle stages can be used for rows or columns, as desired. Importantly, the exact stages of the software engineering or data engineering used may change depending on the data engineering or software engineering methodology used in the particular project. The table cells represent intersections between the software and data engineering life cycles, where artefacts can be reused between life cycles, and are represented in Table 15.2 as tools that facilitate this reuse.

Before the software engineering and data engineering methodologies can be combined, terminology and equivalence between the terms must be agreed upon within a project. A standard equivalence between the generic concepts in data, model-driven software and program language engineering is shown in Table 15.1. Depending upon the scope of the project, the equivalence may not be as direct as those shown. So, for example, a particular upper ontology may be used as a model in software engineering, which may be represented at program runtime in practice, for a particular project. The abstraction level at which each artefact is expected to be used when shared between data engineering and software engineering processes should be documented as part of the process.

Table 15.2 shows an example of the incomplete synchronisation table for the ALIGNED project. The acronyms used here are expanded in more detail in Table 15.3. It uses the data engineering life cycle stages defined in the LOD2 project [12] and the software engineering life cycle stages used in the ALIGNED project. In this table, the synchronisation points indicate where data engineering tools are expected to interoperate with the software engineering processes and vice versa. Each entry in the table indicates a point that requires collaborative effort between software and data engineering processes.

Table 15.1 Comparison of terminologies in software and data engineering

Data engineering	Software engineering	Programming	Metalevel
Schema, ontology language	Meta metamodel	Grammar notation	M3
Upper ontology	Metamodel	Language grammar	M2
Domain ontology, schema	Model	Program definition	M1
Triple, data set	Instance, object	Program runtime	M0

Table 15.2 An in-progress table of the synchronisation points on the ALIGNED project

	Software engineering				
Data engineering	Analysis	Design	Implementation	Validation	Maintenance
Manual revision/ author	UGP	UGP			Dacura
Interlink/fuse					
Classify/enrich	ORE	Model Cat.			
Quality analysis	Dacura	ORE			Dacura
Evolve/repair	ORE	Dacura		RFN	
Search/browse/ explore	Model cat.				Booster
Extract	Dacura	CJDE			
Store/query			Booster		

Table 15.3 A tool-oriented synchronisation table for the ALIGNED project

	Software engineering				
Data engineering	(1) Analysis	(2) Design	(3) Implementation	(4) Validation	(5) Maintenance
(A) Manual revision/author	UGP	UGP			Dacura
(B) Interlink/fuse					
(C) Classify/enrich	ORE (enrich)	Model catalogue			
		ORE (enrich)			
(D) Quality analysis	Dacura	Dacura			
	RDFUnit	ORE			Dacura
	ORE (validation)				RDFUnit
					RFN
					ELV
(E) Evolve/repair	ORE (repair)	Dacura	RFN		RFN
		ORE			ELV
(F) Search/browse/explore	Model catalogue				Booster
(G) Extract	Dacura				
	CJDE				
(H) Store/query			Booster		

For example, the Model Catalogue tool as presented in [1] appears as a synchronisation between the design phase of the software engineering life cycle and the classify/enrich phase of the data engineering life cycle. In the context of the ALIGNED project, this means the Model Catalogue can produce an artefact (a model) that can be consumed by the data engineering "classify" phase, as a schema. In this case, the Model Catalogue can also consume schemas from the data engineering life cycle and use them as models in the software engineering life cycle. The identification of synchronisation points is very much dependent on the project and tools involved and must be done with agreement between software and data engineering processes.

The filled-in table can then be used during the project to trigger discussion and collaboration between software and data engineers. In the Model Catalogue example, this means if a new model has been created as a result using of the Model Catalogue, the model may need to be passed to the data engineering process. The exact formats and mechanism of the interchange must be decided on a case-by-case basis and requires close collaboration between the Model Catalogue developers and the schema producers to agree the format and processes for the exchange. The exchanged artefacts can include specific items such as schemas or less formal documentary artefacts such as a list of requirements.

The initial creation of the table involves at least the software and data engineering practitioners, to decide where the synchronisation points exist. There may be multiple tools at each synchronisation point, and each may have more than one mechanism to align the software and data engineering processes. However, the practitioners must at least decide the metalevel of the exchange and the model(s) at the $(m + 1)$ metalevel (from Fig. 15.3) that will be used to parse and manipulate the data. The frequency of synchronisation must also be determined; for example, a fixed schema may be exchanged once during development, so that the conforming data can be passed between running tools automatically and continuously via APIs at runtime. The data and systems' users can also be involved in the creation of the table, to identify where the requirements for the project will be met. Also data managers, curators and software users may be able to identify from the table where software and data engineering tools interact. An overview of the process is shown in Fig. 15.5.

Both life cycles are expected to run in parallel, and the aim of a combined methodology is twofold: to reduce effort and to produce artefacts that reflect best practice in both software and data engineering. However, the artefacts exchanged at any given synchronisation point may be critical to further development, which may lead to overlap in work done, bottlenecks and problematic dependencies. As engineering life cycle stages produce artefacts, there is potential for waste of effort, as artefacts may be replaced with input from an external life cycle. In part, this is mitigated by knowing which artefacts are expected and when in the life cycle, as per the table. Iteration with a high frequency can also help: synchronisation need not be final. For example, the schema need not be finalised in the data engineering process before matching software is developed—an intermediate version can be produced

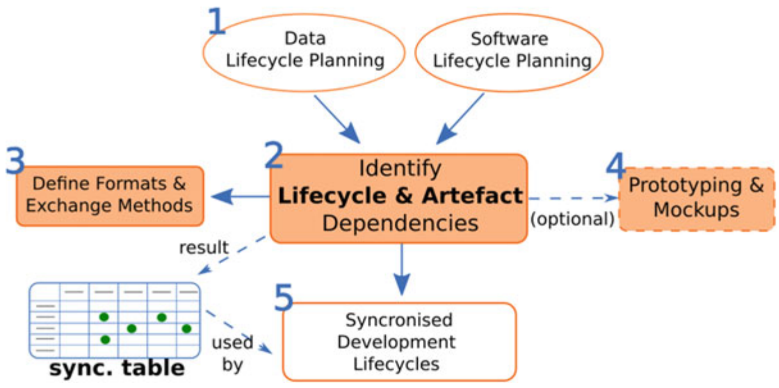
1
Data Lifecycle Planning
Software Lifecycle Planning
2
Identify Lifecycle & Artefact Dependencies
3
Define Formats & Exchange Methods
4
Prototyping & Mockups
(optional)
result
sync. table
used by
5
Syncronised Development Lifecycles

Fig. 15.5 Life cycle stages for integrated software and data engineering

so that both sides can continue working and an updated schema can be produced during the next iteration of the project life cycle.

15.5 ALIGNED Project: A Case Study for Combined Software and Data Engineering

In this section, we demonstrate how the ALIGNED project has been managed using the proposed methodology. The ALIGNED project can be seen as a concrete instantiation of the proposed *meta-methodology*. In this project, there are 28 tools with synchronisation points identified. Without the table, these points may have been overlooked or identified late. Each one of the points now has a strategy for carrying out the synchronisation. The tools and software components in the development of ALIGNED are described in the synchronisation table, below.

Table 15.3 outlines the tool-oriented view of the synchronisation between software and data engineering life cycles for ALIGNED. Each entry of the table represents a synchronisation point within the project. Note that there may be several synchronisation points per table, where multiple tools exploit reuse and best practice between domains. Also not all cells contain any synchronisation points, which reflect the scope and focus of the project. The following summary describes the high-level features of the synchronisation taking place between life cycles at each point and the development bridges between software and data engineering:

15.5.1 *Manual Revision/Author (A)*

- Unified Governance Plugins (UGP) (A1, A2): This tool makes use of the RDF data extracted from a software engineering support tool for bug tracking. It is

used to assist requirements gathering in the software engineering life cycle by managing documents and issues (e.g. duplicate detection or suggestions) and affects the design decisions of the system.
- Dacura (A5): In the Semantic Web maintenance phase, tools can help to identify software bugs that are caused or triggered by data-quality issues and produce notifications for authors to correct these data errors.

15.5.2 Classify/Enrich (C)

- ORE (C1): Tools for ontology validation, enrichment and repair that can provide ontology violations, suggest new ontology axioms or suggest semi-automatic fixes for the benefit of software engineers.
- Model Catalogue (C2): In the design phase of a software engineering life cycle, tools are used to create models, by applying domain expertise and reusing existing models. Models may be used as the basis for software. These schemas can be transferred from the software engineering domain to the data engineering domain as ontologies. The ontologies from the data engineering life cycle can also be transferred to the software engineering life cycle for use as the basis for creating software.
- ORE (C3): Based on the ontology analysis by ORE, changes are made to the ontology that software engineering tools can use to improve the underlying model.

15.5.3 Quality Analysis (D)

- Dacura (D1): In data engineering this tool can be used to produce data-quality tolerance requirements to constrain the data that is harvested, and these can be used as the basis for requirement in software engineering.
- RDFUnit (D1): The data engineering tools can be used to identify and analyse quality issues in the data. The results can give software engineers an insight of the underlying data set and provide feedback to tackle any quality issues.
- ORE (D1): The tool can provide ontology violations, suggest new ontology axioms (enrich) or suggest semi-automatic fixes for resolving violations. This can inform the requirement phase of software engineering, with information about data-quality issues.
- ORE (D2): Based on the ontology analysis carried out by the tool, design of the ontology can be improved.
- Dacura (D2): This defines statistical data-quality measures which must be met by the data. The tool supports the software engineering design phase by suggesting User Interface refinements to eliminate errors.

- Dacura (D5): This can help to identify software bugs that have been caused by data-quality errors and produce notifications for authors to correct these data errors and make software engineering systems more robust.
- RDFUnit (D5): In the maintenance phase, this tool can be used to identify constraint violations. Based on the analysis and design implementation, the violation results can be fed to a quality repair tool or human editors to fix the violations.
- Repair Framework and Notification (RFN) (E5): This tool checks data sets against specific data constraints (e.g. expressed as SHACL or ShEx documents) and provides semi-automatic repair strategies if violations are encountered. The tool is used in both implementation and maintenance phase because the defined data constraints influence the implementation of algorithms in the software engineering phase. Also as taxonomies are changed (additions, removals, merging with other sources), the data need to be checked for consistency with the new constraints.
- External Link Validation (ELV) (D5): This tool dereferences links from PoolParty taxonomies to "external" resources on the Web and provides statistics on invalid (broken) links. The synchronisation point is used on a regular basis for maintaining taxonomies created with PoolParty and informs the software engineering phase.

15.5.4 Evolve/Repair (E)

- ORE (E1): The tool can provide ontology violations, suggest new ontology axioms (enrich) or suggest semi-automatic fixes (for resolving violations). The analyses of the ORE findings drive the design and improvement of the underlying model in the software engineering life cycle.
- Dacura (E2): This tool can define statistical data-quality measures which must be met to support software engineering and suggest UI refinements to eliminate errors.
- ORE (E2): Based on the ontology analysis, this tool drives the design and improvement of the ontology, and this knowledge is transferred to model modifications in the software engineering phase.
- Repair Framework and Notification (RFN) (E4): The synchronisation point is used in both implementation and maintenance phase because the defined data constraints influence the implementation of algorithms, and, as taxonomies are changed (additions, removals, merging with other sources), the constraints need to be satisfied.
- Repair Framework and Notification (RFN) (E5): The tool is used in both implementation and maintenance phase because the defined data constraints influence the implementation of algorithms, and, as taxonomies are changed (additions, removals, merging with other sources), the constraints need to be satisfied. External Link Validation (ELV): This dereferences links from

PoolParty taxonomies to “external” resources on the Web and provides statistics on invalid (broken) links. The tool is used on a regular basis to suggest maintenance tasks in PoolParty.

15.5.5 Search/Browse/Explore (F)

- The Model Catalogue (F1): In the analysis phase of a model-driven software engineering, this tool is used to explore and gather metadata related to the system under construction. In the search browse and phase of the data engineering life cycle, this translates to searching and browsing the ontologies of data set.
- Booster (F5): In the maintenance phase of the software engineering life cycle, Booster can be used to create tools to extract data from a data store. The data may be used directly by users or by other tools via an API. In the data engineering context, Booster-generated tools may be used to provide a well-defined API and search data and gather data into the data store.

15.5.6 Extract (G)

- Dacura (G1): This can be used to produce data-quality tolerance requirements to constrain the data that is harvested and informs the software engineering analysis phase by defining what data is to be harvested.
- Confluence/JIRA Data Extractor (CJDE) (G1): This tool connects to software engineering management tool (Confluence/JIRA) installations (using the REST APIs) and extracts relevant requirement information and tickets (e.g. bugs, improvements) and creates RDF data. The RDF data is used to inform the data engineering life cycle.
- Confluence/JIRA Data Extractor (CJDE) (G3): This tool extracts relevant requirement information and tickets and creates RDF. This synchronisation is used on a regular basis in requirement design phase to understand the changes needed to the PoolParty tool.

15.5.7 Store/Query (H)

- Booster (H3): In the implementation phase of the software engineering life cycle, Booster-generated systems provide, create, read, update and delete functionality for data in a data store, as well as implement any user-specified actions, which can be accessed as triples via an API.

15.6 Related Work

Data-intensive systems require careful alignment between data engineering and software engineering life cycles to ensure the quality and integrity of the data. Data stored in such systems typically persists longer than, and may be more valuable than, the software itself, and so it is key that software development is sympathetic to the aims of “big data”: scalability to large volumes of data; distributed, large-scale research across multiple disciplines; and complex algorithms and analysis. These are normally described in the literature as the four V’s of big data: velocity, variety, volume and veracity [14].

In existing development methodologies, software and data engineering are considered as separate concerns [15]. Integrating these introduces a number of new challenges: software engineering aims of software quality, agility and development productivity may conflict with data engineering aims of data quality, data usability, and researcher productivity. Further challenges include federation of separate data sources, dynamic and automated schema evolution, multisource data harvesting, continuous data curation and revision, data reuse and the move towards unstructured/loosely structured data.

Auer et al. [12] identify challenges within the domain of life cycles for linked data projects. These include extraction, authoring, natural-language queries, automatic management of resources for linking and linked data visualisation. Typically seen as concerns for data life cycles, they all have a major impact upon software development: the authors mention component integration, the management of provenance information, abstraction to hide complexity and artefact generation from vocabularies or semantic representations.

Mattmann et al. [16] use their experience of data-intensive software systems across a range of scientific disciplines to identify seven key challenges which may be summarised as:

- *Data volume*: Scalability issues that apply not just to the hardware of the system, but may affect the tractability and usability of the data.
- *Data dissemination*: Distributed systems bring challenges of interoperability and can lead to complex system architectures.
- *Data curation*: Supporting workflows and tools for improving the quality of data, in a way that allows subsequent inspection or analysis.
- *Use of open source*: Complex technologies will depend upon reliable, reusable components supporting generic functionality.
- *Search*: Making the data collected available in a usable fashion to users, including access to related metadata.
- *Data processing and analysis*: Boiling down to workflows, tasks, workflow management systems and resource management components.
- *Information modelling*: The authors state that “the metadata should be considered as significant as the data”.

The authors split these challenges into further subcategories and point out the many interdependencies between these problems. Zaveri et al. [17] take a broader view, highlighting inadequate tool support for linked data-quality engineering processes. Where tool support does exist, these tools are aimed at knowledge engineers rather than domain experts or software engineers.

Anderson [18] agrees with this issue, describing a more wide-ranging lack of support for developers of data-intensive systems. He also identifies “the necessity of a multidisciplinary team that provides expertise on a diverse set of skills and topics” as a nontechnical issue that can be addressed by projects dealing with large, distributed data sets. A technical equivalent to this issue is to understand notions of iteration with respect to the data modelling—he argues that domain knowledge is required in order to understand data collection and curation. Subsequently, he also argues for technical knowledge in order to match frameworks with requirements, emphasising the need for a multidisciplinary team.

Some solutions to these challenges have been identified—most notably in the area of model-driven software engineering, domain-specific languages and generative programming. These approaches, in combination with linked data languages and schemas, enable self-describing data structures with rich semantics included within the data itself. Aspects of program logic previously encapsulated in software are now embedded in data models, meaning that the alignment between data and software engineering becomes even more important. But these approaches can lead to further problems. Qiu et al. [19] identify two issues: firstly, there is an interaction between domain experts and application developers, and, secondly, that change to schema code may not always impact application code in a straightforward manner.

15.7 Conclusion

The proposed methodology has been put into practice and observed for utility: however, it can be seen as a hypothesis that will be refined in future iterations. One extension will record and generalise the tasks required at each synchronisation point. These observations may be condensed into software and data engineering experience “pearls” [20], for later reuse. As the project progresses, new and previously unidentified synchronisation points may be added to the table. Similarly, overlapping synchronisation points may be merged, where there is duplication of effort. Synchronisation points may also be moved or removed, depending on changes or errors discovered in the requirements of either software or data engineering project management.

Meanwhile, the methodology is to be refined by monitoring how the methodology is used within the consortium, by way of surveys and workshops. Some analysis is needed on the correspondence between the use cases and the tools developed on the project. Further analysis may be necessary for user-oriented perspectives of the methodology on the effect of the synchronisation points on the development efforts of consortium partners.

This chapter has presented a methodology for combining software and data engineering project life cycles. The hurdles to a combined methodology have been discussed and include the differences in prescriptive and descriptive philosophies and the changing needs from project to project. An agile *meta-methodology* for combined project management has been proposed as a solution to the identified issues. The benefits of the methodology are clear: reuse of artefacts between domains and application of best practice from both domains transparent project management when multiple life cycles are involved. The application of the methodology in the interdisciplinary ALIGNED project has been presented as an initial case study.

References

1. Davies J, Gibbons J, Welch J, Crichton E (2014) Model-driven engineering of information systems: 10 years and 1000 versions. Sci Comput Program 2014:88–104
2. Davies J, Gibbons J, Milward A, Milward D, Shah S, Solanki M, Welch J (2015) Domain specific modelling for clinical research. In: Proceedings of the workshop on domain-specific modeling. ACM, New York
3. Dimou A, Kontokostas D, Freudenberg M, Verborgh R, Lehmann J, Mannens E, Hellmann S, de Walle RV (2015) Assessing and refining mappings to rdf to improve dataset quality. In: Proceedings of the 14th international semantic web conference
4. Feeney KC, O'Sullivan D, Tai W, Brennan R (2014) Improving curated web-data quality with structured harvesting and assessment. Int J Semant Web Inf Syst 2014:35–62
5. Schandl T, Blumauer, A (2010) PoolParty: SKOS thesaurus management utilizing linked data. In: The semantic web: research and applications: 7th extended semantic web conference, ESWC 2010, Heraklion, May 30–June 3, 2010, Proceedings, Part II, Springer, Berlin
6. Kontokostas D, Mader C, Dirschl C, Eck K, Leuthold M, Lehmann J, Hellmann S (2016) Semantically enhanced quality assurance in the JURION business use case. In: The semantic web. Latest advances and new domains: 13th international conference, ESWC 2016, Heraklion, Proceedings, Springer, May 29–June 2 2016
7. Royce WW (1970) Managing the development of large software systems. In: Proceedings of IEEE WESCON. Los Angeles
8. Larman C, Basili VR (2003) Iterative and incremental development: a brief history. Computer 2003:47–56
9. Martin J (1991) Rapid application development. Macmillan Publishing Co., Indianapolis
10. Fowler M, Highsmith J (2001) The agile manifesto. Software Dev 9(8):28–35
11. Selic B (2003) The pragmatics of model-driven development. In: IEEE Softw. Sept 2003, pp 19–25
12. Auer S, Bühmann L, Dirschl C, Erling O, Hausenblas M, Isele R, Lehmann J, Martin M, Mendes PN, van Nuffelen B, Stadler C, Tramp S, Williams H (2012) Managing the life-cycle of linked data with the LOD2 stack. In: The semantic web – ISWC 2012: 11th International Semantic Web Conference, Boston, 11–15 Nov2012, Proceedings, Part II, Springer, Berlin/Heidelberg
13. Reiter R (1978) On closed world data bases. Springer US, Boston
14. Hitzler P, Janowicz K (2013) Linked data, big data, and the 4th paradigm. Semantic Web 2013:233–235
15. Cleve A, Mens T, Hainaut J-L (2010) Data- intensive system evolution. Computer 2010:110–112

16. Mattmann CA, Crichton DJ, Hart AF, Goodale C, Hughes JS, Kelly S, Cinquini L, Painter TH, Lazio J, Waliser D et al (2011) Architecting data-intensive software systems. Springer, New York
17. Zaveri A, Rula A, Maurino A, Pietrobon R, Lehmann J, Auer S (2015) Quality assessment for linked data: a survey. Semant Web 2015:63–93
18. Anderson, KM 2015 Embrace the challenges: software engineering in a big data world. In: Proceedings of the First International Workshop on BIG Data Software Engineering, IEEE Press, Piscataway
19. Qiu D, Li B, Su Z (2013) An empirical analysis of the co-evolution of schema and code in database applications. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, New York
20. Bentley J (1986) Programming pearls. ACM, New York