# 3

# Methodology

**James Welch[1], Jim Davies[1], Kevin Feeney[2], Pieter Francois[1], Jeremy Gibbons[1] and Seyyed Shah[1]**

[1]University of Oxford, UK
[2]Trinity College Dublin, Ireland

## 3.1 Introduction

Software engineering is concerned with the development of reliable computer applications using a systematic methodology. Data engineering involves the collation, organisation, and maintenance of a dataset, or data product, and may be seen as the dual of software engineering. The two processes are typically treated as separate concerns – largely as a result of different skill sets. However, there is often a great deal of overlap: dependable software is reliant on consistent, semantically correct data; processing data at scale requires high-quality tools and applications.

For most enterprises, the data they hold may well be their most valuable asset. Day-to-day operations will be dependent on data concerning customers, payments, and stock. It is vital that this data is of high quality: any loss of integrity or inconsistencies with operating practices or business processes, may be costly, and in many cases irreparable. Furthermore, the ongoing success of the business is increasingly reliant on analysis of the data: historical reporting, predictive analytics, and business intelligence. These latter processes, along with decreasing costs for storing and managing data, drive an increase in scale: minimising human effort is vital, and new Big Data tools and techniques are required to manage ever-larger datasets.

For some organisations, the data may be the primary artefact or the product in itself. From research enterprises to social networks, the value of the data stems from its quality, coverage, and completeness. These curated datasets may be the product of many smaller ones, perhaps different in structure or domain, and linked to create new, richer datasets. For these

combined datasets, the ability to version and update individual components is critical: users of the data require up-to-date input, new features, and access to corrections and clarifications. Tool support must be sympathetic to changes in requirements and the acquisition of new data, and must scale accordingly.

It therefore follows that Software Engineering and Data Engineering are closely related. Mission-critical software is reliant on high-quality data, and the construction and maintenance of large datasets is dependent on secure, reliable software. Many of the key challenges are common to both disciplines: correctness, scale, and agility; tools and techniques for improving software quality may also result in improved data quality and vice versa.

The increase in popularity of "Big Data" analytics means that solutions to these challenges are required more than ever. The rise in data-intensive applications – those systems that deal with data that is large in scale, complex, or frequently changing[1] – has brought about a requirement to abandon traditional methodologies and explore new processes and techniques. A broader range of software applications for processing data, including visualisation, natural language processing, and machine learning, have provided new areas for innovation, and the integration of a range of software components around an underpinning data corpus has become a typical system architecture.

Engineering processes for both data and software are also required to be sympathetic to the so-called "Five V's of Big Data": velocity, volume, value, variety, and veracity. The speed at which data can be acquired – manually through the efforts of large groups, or automatically through complex applications – can impact the processes of data curation, enriching, and analysis. The ever-increasing amount of data collected – which can include static "historical data" and changing contemporaneous data – can reach scales challenging existing software scalability. The perceived value of data captured requires precision software, and rigorous data engineering processes, to ensure continuing accuracy and integrity. The ever-greater heterogeneity of data to be handled creates semantic issues, which must be resolved when linking and analysing data. Finally, the quality or trustworthiness creates further semantic issues – understanding the meaning, provenance, and accuracy of data is vital to realising its worth, and all phases of both software and data engineering processes need to take this into account.

Modern approaches to software engineering consider automation for agility and correctness, formal techniques for reliability and iterative approaches to improve delivery time and adapt to requirements. Data

[1] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, 2016.

engineering as a discipline is less mature, although certain phases of an iterative process have been identified, and dependencies between phases can infer a natural development life cycle. However, both life cycles remain independent, and finding an integrated process, which considers both software and data in parallel, remains a considerable challenge.

The content in this chapter is adapted from a paper submitted (in January 2018) to Elsevier's Journal of Information Sciences.

## 3.2 Software and Data Engineering Life Cycles

### 3.2.1 Software Engineering Life Cycle

Modern software development methodologies can be seen as refinements to the original waterfall process for hardware systems development. First conceived as a "stagewise" model,[2] an instantiation targeting software development is typically summarised by the diagram in Figure 3.1.

In this most basic process, progress flows one way, through each of the stages, and one phase cannot begin until the previous phase has been completed. Each of the stages can be "signed off" by either the customer or the developer in such a way that completion of a phase can be recognised and made final. For example, the requirements for the system determine the scope

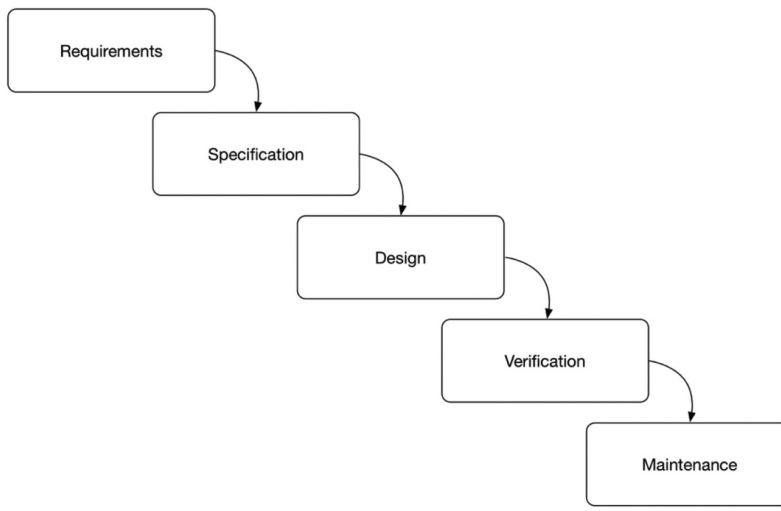


**Figure 3.1** The waterfall process for software development.

[2]H. D. Benington, Production of large computer programs, IEEE Annals of the History of Computing 5 (1983), pp. 350–361.

of the specification; the completed specification document may be seen as a contract for the design work.

The first major problem with the waterfall model is that the execution of one phase of design may influence the previous stage. This is particularly apparent in the verification stage: issues in verification will require further effort in design; design may be said to be unfinished until verification is complete. This may also hold true in the case of specification: the process of producing a clear, precise specification may uncover ambiguities or inconsistencies in the requirements provided.

One solution to this problem is to allow feedback from one phase to modify earlier decisions. This leads to a modified version as proposed by Boehm,[3] in which backward arrows lead from one phase to the preceding one (see Figure 3.2). Although this allows for some notion of iteration in development, allowing decisions made in each phase to be revisited, it suffers from another flaw, that is, estimating delivery time (and therefore cost) can be very difficult. Without specific bounds on revisiting decisions, overall implementation can take unspecified amounts of time, leading to frustration for both developer and customer.

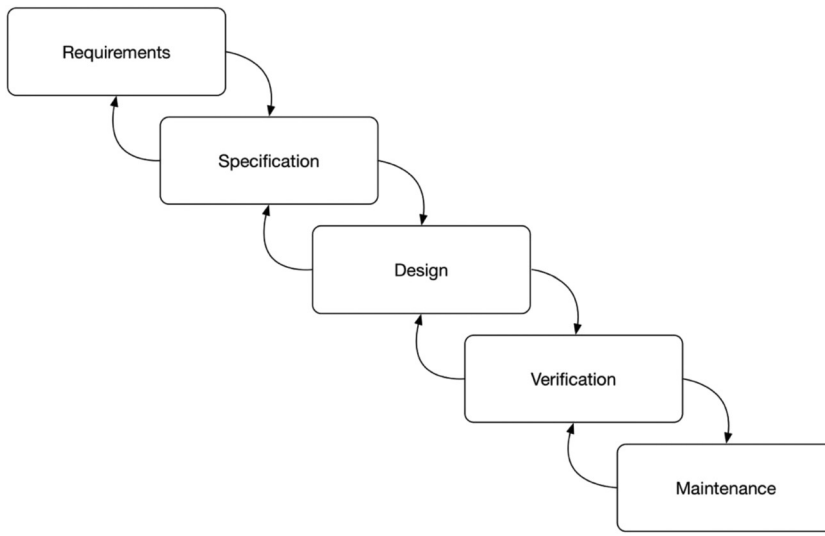


**Figure 3.2** A modified waterfall process.

[3]B. W. Boehm, Software Engineering, IEEE Transactions on Computers 25 (12), pp. 1226–1241, 1976.

This uncertainty can be exacerbated by another common problem in software development: customers often do not know, or understand, precisely what they want until they have had a chance to see it, or interact with it. Business rules that may seem fixed at the time of requirements and specification may need revising in light of constraints in subsequent design or implementation stages. A good software engineering process must be sympathetic to revisiting even the earliest requirements decisions after design and implementation are underway, but still be amenable to stable project management in order to allow predictable costs and timescales.

More modern approaches to these problems can take two forms. The first of these is more technical, and directed at the actual design and implementation process: by reducing the length of time taken to get from requirements to implementation, decisions can be revisited quickly, and with less development effort. Prototyping allows the customer or user to get a feel for the solution earlier, permitting the requirements or specification to be revisited sooner in the overall implementation process. Automation in the implementation phase can reduce the effort involved in updating implementations to match updated requirements.

The second approach is another update to the software engineering life cycle, allowing multiple iterations of the traditional model, typically reducing the retrograde steps in the previous model in favour of completing an implementation and starting a new requirements and specification iteration sooner. Figure 3.3 shows a typical iterative software development life cycle.

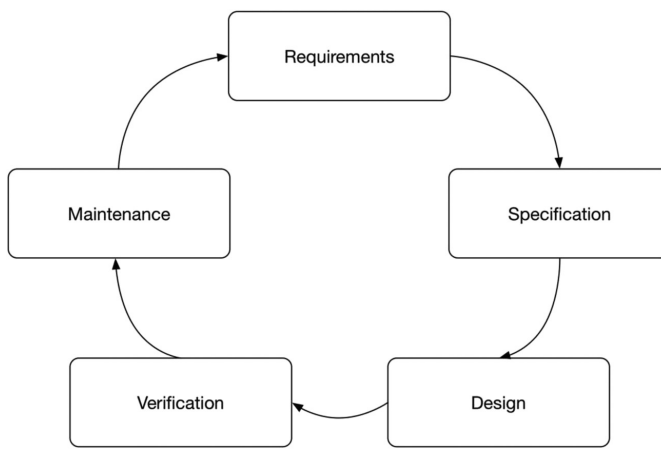


**Figure 3.3** An iterative software development process.

The iterative model allows a more flexible approach to contracts and timelines: short cycles of the entire process allow prioritisation of features; early implementations can be used as prototypes and complex details can be saved for future iterations when there may be more clarity. Cycles are typically kept to a predefined length; at the start of each cycle, the scope of each phase is determined, managing time and cost expectations. Although system-wide requirements will be gathered throughout the whole cycle, some analysis will be performed at the start of each cycle in order to confirm the scope for the next cycle. Overall, time and cost estimation can be managed more effectively[4] and revised at the end of each cycle.

Another advantage to the iterative approach is that it changes the nature of the maintenance phase. Typically, during the life of the software, functionality will need adjusting to match evolving business requirements. With the standard waterfall model, the final phase of maintenance is often insufficient to deal with updated requirements, and the whole process needs to begin again; an iterative approach takes this into account, and maintenance can be merged in as part of the overall development and re-evaluation cycle.

The "Manifesto for Agile Software Development"[5] proposes 12 principles for such a development process, including to "satisfy the customer through early and continuous delivery of valuable software", to "welcome changing requirements, even late in development", and to "deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale". The iterative approach is typically referred to as an "agile" approach, although the principles as set out for an agile process extend beyond the software life cycle itself and provide guidance for the way in which developers work as a team and interact with their customers.

Managing an iterative process effectively can still be difficult: although individual cycles can be fixed in duration, and development effort within the cycle may be reasonably estimated, it can still be difficult to manage priorities and overall development direction. A number of variations on the iterative, "agile" process have been proposed, and frameworks built around

[4] A. Begel, N. Nagappan, Usage and perceptions of agile software development in an industrial context: An exploratory study, in: First International Symposium on Empirical Software Engineering and Measurement, pp. 255–264, IEEE, 2007.

[5] K. Beck, Manifesto for agile software development, http://agilemanifesto.org, accessed: November 2017 (2001).

them, for example, Scrum',[6] Kanban,[7] and Extreme Programming,[8] all of which can help with cost estimation, reducing the time spent on verification and enhancing code quality.

An agile approach can also be counter-productive for building certain types of software where solutions are complex and irreducible. Such solutions require a high degree of planning and design and architectural decision-making in advance. An iterative development methodology can restrict the solution space to one in which development time may be reasonably estimated, where progress may be demonstrated at the end of each iteration and where prioritisation stays consistent.

### 3.2.2 Data Engineering Life Cycle

As an emerging field of research, the processes of data engineering used in industrial applications are still relatively immature. The LOD stack LOD2[9] is a collection of integrated tools supporting a life cycle for creating and managing Linked Data. Auer et al.[10] proposed an iterative process for developing linked open datasets. Eight core activities of Linked Data management are identified and managed as phases in an iterative life cycle, consistent with the principles of Linked Data:

- storage/querying: retrieving and persisting information to be included as part of the dataset;
- manual revision/authoring: processes for manual curation of content;
- interlinking/fusing: creating and maintaining links between datasets;

---

[6] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Vol. 1, Prentice Hall, 2002.

[7] M. O. Ahmad, J. Markkula, M. Oivo, Kanban in software development: A systematic literature review, in: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on, IEEE, pp. 9–16, 2013.

[8] K. Beck, Embracing change with extreme programming, Computer 32 (10), pp. 70–77, 1999.

[9] S. Auer, V. Bryl, S. Tramp, Linked Open Data–Creating Knowledge out of Interlinked Data: Results of the LOD2 Project, Vol. 8661, Springer, 2014.

[10] S. Auer, L. Bühmann, C. Dirschl, O. Erling, M. Hausenblas, R. Isele, J. Lehmann, M. Martin, P. N. Mendes, B. van Nuffelen, C. Stadler, S. Tramp, H. Williams, Managing the life-cycle of linked data with the LOD2 stack, in: P. Cudre-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, E. Blomqvist (Eds.), International Semantic Web Conference, pp. 1–16, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

- classification/enrichment: creating and maintaining links between data and models of data (which themselves may be linked and part of the dataset);
- quality analysis: testing for data completeness and correctness;
- evolution/repair: correcting invalid data resulting from a quality analysis phase via either manual or automated processes;
- search/browsing/exploration: making data artefacts available to domain experts or to users beyond the original authors;
- extraction: producing or publishing profiles or projections of data to be used in other applications.

Figure 3.4 shows an iterative life cycle incorporating these stages.

These stages of data engineering have been identified primarily for building tool support and integrated frameworks, encouraging compatibility of independent tools within a particular framework. As with the software engineering process, feedback from one phase may be fed into another. For example, the models linked during the classification or enrichment stage will determine the scope of the quality analysis stage; any errors found during quality analysis may need to be resolved in the evolution/repair phase.

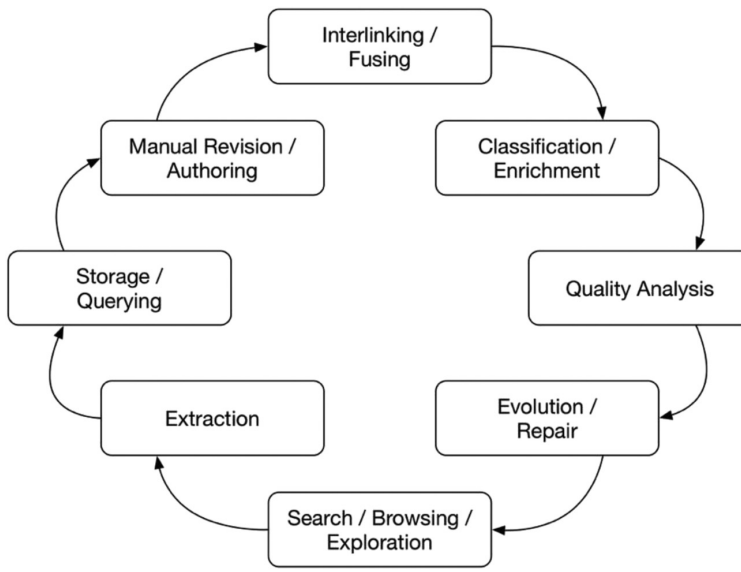


**Figure 3.4** A data engineering life cycle.

## 3.3 Software Development Processes

In order to design a combined software and data engineering process, we will first consider some modern approaches to software engineering at scale and how phases of the data engineering life cycle might be integrated or merged. As explained in Section 3.2, automation is often seen as key to improving both the speed of software delivery and the correctness of the delivered product. In this section, we will consider three cross-cutting techniques of software engineering and discuss their advantages and disadvantages. We first consider model-driven approaches to software development, and the trade-off between automation and customisation. We then look at formal techniques, in particular formal specification, validation, and verification. Finally, we discuss test-driven development and its role in an iterative development process.

### 3.3.1 Model-Driven Approaches

MDE describes a development process in which the components of the final software artefact are derived – either manually or automatically – from models that typically form part or all the specifications or requirements of the system. Rather than writing software that understands the data itself, software is written that understands the modelling language and is capable of handling updates to the model. Such software can be reused in different applications within a similar domain, minimising the time spent on the implementation phase and capturing common repeating patterns that would otherwise have to be repeated on each cycle of an iterative development.

MDE is a promising starting point for our combined methodology: by choosing well-suited models that fit the application domain, updating software to match evolving data can be achieved by simpler updates to a model. With suitable tool support, this methodology may also allow ordinary business users to manipulate these models and help bridge the gap between requirements and specification.

As implied above, MDE approaches fall on a sliding scale between a fully automated generation process, and something much more manual, allowing greater flexibility and customisation. An overview of some of the possible approaches and discussion of their practicality follows below.

The first MDE approach can be described as “full automation”: everything is modelled – including future-proofing – and machine learning is used to learn how to change the model from the flows of data as their format changes over time. This approach has significant advantages in terms of

maintenance cost: once the system is deployed and operational, minimal further intervention is required. However, although learning how to self-adapt a model is theoretically possible, it remains impractical for real-world applications. Another problem is the generation of training datasets for the machine-learning component: this is currently beyond the scope of most data engineering projects.

A second approach that appears more practical is where a full model of system behaviour is manually produced, but a fully functional software implementation can be generated automatically from the model. The modelling language should be designed in such a way that a broad range of likely future developments and feature requests can be handled without any custom code needing to be written. If such tools are written with evolution and upgrade in mind, they may be used for rapid prototyping, as part of an iterative agile process or as a technique to manage and enable software change beyond initial deployment. If the tools for editing and managing models are good enough, such changes may even be carried out by business users and deployed instantly, rendering the whole process cost-free from a technical resources point of view.

Although feasible within particular domains of application, this approach does not work universally: there can be no theoretical basis for automatically implementing arbitrary behaviours and functionalities. However, subsets of the overall problem are tractable, and such modelling languages – also referred to as DSLs – exist with usable tool support. The UML[11] is the most significant attempt to create a complete modelling language. It does not have a formal semantics itself, but can be given one for a specific purpose, and there are many tools based on subsets of the language. UML has been successfully deployed for building large, complex model-driven systems. However, in practice, the development and testing of the models takes considerable amount of time and effort to get right. Such systems are most appropriate for domains in which a lot of effort is spent moving data through highly stereotyped workflows that do not change rapidly over time and where significant resources can be allocated to testing and managing model updates.

The remaining modelling approaches do not attempt to model behaviours, limiting their scope to data. The third approach is where a complete data model, containing a full specification of all the classes and properties that

[11] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modelling Language Reference Manual, Pearson Higher Education, 2004.

are present in the data, is used to constrain or guide the manipulation of data corresponding to that model. This approach has the advantage that constraints on data are easier to define and use than those upon behaviours. A complete data model can be used to generate a large proportion of software components in an information system – for example, the data storage mechanism and user interfaces.

The disadvantages to this third approach are that although generation processes have been formally solved and public standards such as OWL are available, in practice, automated software generation from such models is still very hard and requires tools to be built from scratch. Most importantly, the conceptual framework and the assumptions underlying the logic of OWL need to be changed. Existing tools for model management are typically focussed on knowledge engineers with specific goals and as such are not really suited to business users.

A fourth approach is that of partial data modelling: where a subset of the information domain is specified – limited to ad-hoc or incomplete positive or negative constraints on the data. Here the assumption is made that the model is not exhaustive, that there are states of the data that are not addressed in the model. This technique has a particular advantage in processing large datasets: where data are messy and do not necessarily conform to any model, we can identify and filter out the most important problems caused by the lack of structure. A model may be incrementally built, adding rules to specifically address any issues with the data as they are encountered.

A disadvantage with this approach is that the incompleteness of the model prevents most automation techniques. Another is that the models are built up by accumulation of ad-hoc rules and become difficult to manage over time, invariably becoming a barrier to agility, and may become inconsistent. Changes to the model may result in large changes to the data – or worse, required changes to the data may go unnoticed or their calculation or derivation may be infeasible from the model.

### 3.3.2 Formal Techniques

The use of formal methods in the development of programs has been the traditional practice for those systems that may be seen as safety-critical: typically those systems whose failure could endanger human life. Such formal techniques include the mathematical derivation of program code from precise specifications, the logical proof that code exactly implements specifications in the form of contracts, or the exhaustive verification of software to show that

unwanted behaviours are precluded. Each suffers from the same problems: that formal techniques are slow and expensive, and do not scale to large complex software systems. A rigorous, mathematical approach will require developers with very specialised skills and experience.

However, there have been some successful applications of formal techniques in practical software development. Automation can solve problems of scalability, but a completely automatic process is impossible in the general case. One solution is to restrict the problem domain: pattern matching can be applied to the specification and particular refinements applied; proof libraries and verification results can be stored for reuse. Another solution is to focus automation on part of a stepwise process; for example, automatically generating method stubs or proof obligations for manual completion.

In many cases, formal techniques are associated with a more traditional waterfall method development. This can be because there is a need for a detailed, comprehensive specification before the mathematical process can begin – requiring that much of the solution is explored before any programming starts. Hall[12] described the development life cycle of the specification itself: from Learning through Production and Simplification. These stages are necessary within any development method, but in a formal code derivation process, these must typically happen before any code has been written. This may result in an overall speed increase, but does not incorporate the fundamental component of an iterative process: feedback – the user's response to an initial implementation.

However, the construction of a complete, precise specification is not without merit. The explication and analysis of the problem space is invaluable when developing code, most importantly when a team of developers require a shared understanding. Human-readable documentation is also important for giving context and addressing subtleties not obvious from the plain mathematical statement. By addressing both specification and requirements in this way, developers have a clearer sense of direction, customers can make better judgements on the suitability of a solution, and managers can better manage expectations of time and cost.

### 3.3.3 Test-Driven Development

A test-driven (or "test-first") software development proceeds in an iterative fashion, but relies on a short development cycle, focussed on building

[12] A. Hall, Seven myths of formal methods, IEEE Software 7 (5), pp. 11–19, 1990.

functionality to meet requirements, rather than specification. At the start of each iteration, acceptance tests are written to validate the implementation of the next round of features: the expectation is that these new tests will initially fail. Minimal changes to the code are made in order to get the test suite completely passing; once all tests pass, the feature development is complete. An optional refactoring phase can be used to tidy the code, whilst maintaining a full suite of passing tests.

As well as measuring the suitability of the latest iteration of development, tests also provide a valuable restraint on regressions: that previously correct functionality is not broken by the latest updates. This can give users confidence in the stability of the software and reduce the burden for developers.

An agile test-first approach can lead to high-quality, timely software. However, some of the caveats about agile, iterative development also apply: maintaining long-term objectives whilst focussing on short-term goals can be difficult. Finding appropriate levels of code coverage requires experience: total coverage is often impossible; tests covering trivial or non-realistic cases can waste developer time, but too few tests may lead to a reduction in quality.

The test-driven approach to software has obvious parallels in the development of large datasets: the quality analysis phase of development can be used to measure the correctness of the other phases – in particular those of manual revision, interlinking, and enrichment. Tools for finding inconsistencies in data – and highlighting areas of concern – are readily available and well understood by data engineers.

## 3.4 Integration Points and Harmonisation

Although the processes for software engineering and data engineering discussed so far are complementary, it is more than likely that in the development of a data-intensive system, there will be dependencies between the two processes. In general, an integration point corresponds to any pair of points in the software and data engineering life cycles where specific artefacts and processes should be shared. In this section we enumerate three different forms of integration point: overlaps, synchronisation points and dependencies; we discuss the importance of each, and consider the difficulties in spotting them. We conclude the section by examining potential barriers to harmonising the two processes, in terms of terminologies, development roles, models, and tool support.

### 3.4.1 Integration Points

The first type of integration point between the data and software engineering processes is that of a natural overlap. This will be particularly prominent at the start of the project: for example, where the initial implementation of the software may run in parallel with a manual curation of the initial dataset. Similarly, in some projects, a phase of testing the software for correctness may coincide with a phase of quality analysis for the data: bugs in the software may be a cause of inconsistencies in the data; errors in the data may uncover issues in the software. In general, overlapping phases such as these can indicate a requirement for software engineers and data engineers to work together to ensure successful conclusions in both life cycles.

More generally, we can consider synchronisation points: where phases in both cycles are required to start, or finish, at the same time. This could be due to a release of software coinciding with the linking of a new dataset. It may be due to external pressures: the implementation of software and manual update of data to match new business processes; the completion of a cross-cutting software and data concern before a member of staff leaves the organisation.

More generally still, it is important to consider dependencies between phases in cycles. Typically this can mean that a phase in one cycle must finish before another starts, but may simply be that one phase must reach a certain level of completion. One example where a software engineering phase might depend on a data engineering phase would be when data quality analysis must be completed before the requirements for the next iteration of software development can be signed off. An example where a data engineering phase may depend on a software engineering phase might be where a particular software feature must be tested and deployed before some manual data curation may start.

Such integration points may happen regularly with every iteration – for example the requirement to migrate data to match the deployment of new software, or may happen irregularly, for example in response to changes in business processes, the implementation of new features, or updates to external data sources. Thus it becomes important to regularly review known integration points and assess the potential for new integration points in the future. As this requires insight into both data and software engineering development plans, along with an understanding of overall roadmaps and business direction, the integration analysis will involve many stakeholders across a range of disciplines or technical competencies.

As with any project management activity, care should be taken to ensure that dependencies can be appropriately managed. It is conceivable that in rare cases, cyclic dependencies appear: this may indicate that data and software engineering phases need more carefully defining – split up or merged – or that requirements and design need revising. Generic tool support for such project management is readily available, but specialist tooling – as discussed in Section 7 – is really only available for software development processes.

The nature of each integration point needs investigation to explore the best way of addressing it. For example, although some straightforward dependencies may be seen to be sufficiently addressed by a simple sign-off process, the criteria for completion must be agreed beforehand. More complicated dependencies, especially where an overlap in phases is concerned, may require more substantial collaboration between data engineers and software engineers, perhaps with intermediate checkpoints and combined requirements.

### 3.4.2 Barriers to Harmonisation

There are a number of barriers to the easy combination of software and data engineering processes. Although both processes have foundations in computer science and information engineering, the two disciplines have different terminology, and different reference or metamodels. The participants in each will also vary: roles may not have obvious counterparts in the other discipline, and the people carrying out each role will have different backgrounds and skills. Highlighting barriers and potential pitfalls is important so that they can be anticipated and worked around.

An integration point will usually indicate some shared resource between software and data engineering: typically a requirement, a model or a metamodel. It can be important to recognise this shared resource and ensure that both data engineers and software engineers share a collective understanding. A common barrier is that of terminology: although engineers may typically share a common language in the domain of application, with differing skills and backgrounds, software and data engineers may have different technical terminology. An example of this is shown in Figure 3.5 – showing a standard equivalence between terms of abstraction in different domains: data engineering, model-driven software engineering and more general programming.

Based on the scope of the project, however, the equivalence may not be as direct as those shown. For example, in a particular project, one specific

| Meta-level | Data engineering | Software engineering | Programming |
|---|---|---|---|
| M3 | Schema, Ontology Language | Meta-metamodel | Grammar notation |
| M2 | Upper Ontology | Metamodel | Language Grammar |
| M1 | Domain Ontology, Schema | Model | Program definition |
| M0 | Triple, Dataset | Instance, Object | Program runtime |

**Figure 3.5** Comparison of terminology in software and data engineering.

Upper Ontology may be used as a Model in software engineering, which may be represented at Program Runtime in practice. The abstraction level at which each artefact is expected to be used when shared between software and data engineering processes should be documented as part of the process, and any changes in notation – for example, a process used to turn UML software models into an OWL ontology – should be automated if possible.

As well as the differing terminologies, the models themselves may differ. In order to facilitate interlinking, data engineers typically make good reuse of existing models – for example Dublin Core (DC)[13] for generic metadata, Friend Of A Friend (FOAF)[14] for social relationships, or PROV[15] for data provenance information, are all commonly reused or extended. This extension is an essential part of the data engineering process, allowing dataset linking. In software engineering, however, reuse of such pure data models is less common: reuse happens in terms of libraries of functionality. While there are some libraries that do implement standard data models,[16] most are typically restricted to the most trivial – for example hash maps – or the domain-specific – for example models of Microsoft Word documents.[17] Without common models for software and data, harmonising the two development processes will prove difficult.

Enumerating the participants involved in each of the two processes can also highlight potential hurdles. A wide variety of roles may be involved: in software engineering, these might be systems or software analysts, developers and testers; in data engineering these might be data architects, data

[13]S. L. Weibel, T. Koch, The Dublin Core metadata initiative, D-lib Magazine 6 (12), pp. 1082–9873, 2000.

[14]D. Brickley, L. Miller, FOAF vocabulary specification 0.91 (2007).

[15]P. Groth, L. Moreau, PROV-overview. an overview of the PROV family of documents, project Report, April 2013.

[16]C. Mattmann, J. Zitting, Tika in Action, Manning Publications Co., 2011.

[17]The Apache Software Foundation, Apache POI, http://poi.apache.org, accessed: November 2017 (2017).

harvesters and data consumers. There may be roles which can, or should, be shared across the two processes: requirements engineers, system administrators, technical or development managers. Users may be technical or domain experts; they may be users of the software, the data, or both. It is important that interaction between roles is between both sides of the process: software developers should understand the concerns of data quality analysts, for example, and the data architects should collaborate with the software architects.

Another area where software and data engineers can be divided is on the use of tools for managing the development process. In software development, the usual practice is to use an issue-tracking or defect-tracking tool, such as Atlassian Jira,[18] or JetBrains YouTrack.[19] Such tools can help orchestrate an iterative process: plugins are available to manage agile variants such as Kanban or Scrum. Technical problems can be managed through this process too: issues can be raised directly by users, taken through a workflow from prioritisation through development to testing by the developers, and "signed off" as complete by management or the original users. Customisable workflows allow this process to be adapted according to particular development processes or business culture.

Typically, such tool support for data engineering processes does not exist, in part due to the relative immaturity of formalised processes, and in part due to the wide variety of workflows for data curation, some of which will be specific to particular domains. In some cases, customisable tools such as Jira can be re-purposed, and plugins developed, but data engineers – especially the domain experts, who may be non-technical – can often be reluctant to use such tools aimed at software developers. Processes can often be managed in a more ad-hoc fashion without tool support or building additional bespoke support into data curation tools.

Having identified a number of potential barriers to integrating two different engineering processes, we can consider approaches to success. Collaboration and harmonisation between two typically distinct teams in an organisation requires a detailed understanding of the other process and those

[18] J. Fisher, D. Koning, A. Ludwigsen, Utilizing Atlassian JIRA for large-scale software development management, Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2013).

[19] JetBrains, JetBrains YouTrack, https://www.jetbrains.com/youtrack/documentation/, accessed: November 2017 (2017).

participating in it; of compromise in terms of terminology and modelling; a sympathy for those solving orthogonal problems within the same space; and shared sets of resources and tools for collaboration.

### 3.4.3 Methodology Requirements

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 persist 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 Five V's of Big Data: velocity, volume, value, variety, and veracity.

In existing development methodologies, software and data engineering are considered as separate concerns.[20] Integrating these will introduce 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, multi-source data harvesting, continuous data curation and revision, data reuse and the move towards unstructured/loosely structured data.

Auer et al. identified challenges within the domain of life cycles for Linked Data.[21] 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 mentioned component integration, the management of provenance information, abstraction to hide complexity, and artefact generation from vocabularies or semantic representations.

---

[20] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, 2016.

[21] S. Auer, J. Lehmann, A.-C. N. Ngomo, A. Zaveri, Introduction to linked data and its lifecycle on the web, in: Reasoning Web. Semantic Technologies for Intelligent Data Access, pp. 1–90, Springer, 2013.

Mattmann et al.[22] used 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 pointed out many interdependencies between these problems. Zaveri et al.[23] took 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 agreed with this issue,[24] describing a more wide-ranging lack of support for developers of data-intensive systems. He also identified "the necessity of a multidisciplinary team that provides expertise on a diverse set of skills and topics" as a non-technical issue that can be addressed by projects dealing with large, distributed datasets. A technical equivalent to this issue is to understand notions of iteration with respect to the data modelling – he argued that domain knowledge is required in order to understand data collection and curation. Subsequently, he also argued for technical knowledge

[22] C. A. Mattmann, D. J. Crichton, A. F. Hart, C. Goodale, J. S. Hughes, S. Kelly, L. Cinquini, T. H. Painter, J. Lazio, D. Waliser, et al., Architecting data-intensive software systems, in: Handbook of Data Intensive Computing, pp. 25–57, Springer, 2011.

[23] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, S. Auer, Quality assessment for linked data: A survey, Semantic Web 7 (1), pp. 63–93, 2016.

[24] K. M. Anderson, Embrace the challenges: Software engineering in a big data world, in: Proceedings of the First International Workshop on BIG Data Software Engineering, pp. 19–25, IEEE Press, 2015.

in order to match frameworks with requirements, emphasising the need for a multi-disciplinary team.

Some solutions to these challenges have been identified – most notably in the area of model-driven software engineering, DSLs, 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.[25] identified two issues: firstly the interaction between domain experts and application developers, and secondly that change to schema code may not always impact application code in a straightforward manner.

## 3.5 An ALIGNED Methodology

This section outlines the proposed methodology for combined software and data engineering. We describe it as "lightweight", because the technique requires some initial setup and maintenance, and its exact form can be heavily determined by the exact software and data engineering processes, by the tools available and the technical members of the team. However, in this methodology, we propose a general framework for process management, an iterative methodology, and a number of guidelines or recommendations for successful integration. We conclude the section by considering tool support for such a process.

### 3.5.1 A General Framework for Process Management

In Section 5, we outlined a number of potential barriers to harmonising the data and software engineering processes. Our general framework is concerned with reducing the effect of these issues, as well as providing an iterative methodology that is suitably adaptive in response to changes in context. The framework is split into two phases: the first, a "setup" phase, involves some analysis of the preferred engineering processes, the shared resources and integration points, and the impact of any tools, project roles or terminology where managing integration points will prove problematic. The second phase

[25] D. Qiu, B. Li, Z. Su, 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, pp. 125–135, ACM, 2013.

is the iterative development, where the outputs of the setup phase are under a process of continuous revision, such that problems can be foreseen at the start of each cycle.

The setup phase is broken into four consecutive steps – the first of which is to perform some basic analysis on the preferred software and data engineering processes. This will be most greatly influenced by the skills of the technical collaborators, the preferred management style, and the requirements laid down by the users. As part of the guidelines later in the section, we strongly recommend iterative development approaches to both software and data, and for the remainder of the section assume processes similar to those outlined in Figures 3.3 and 3.4 – generic iterative approaches corresponding with an agile approach. However, specific projects may choose, for example, a specific software testing phase apart from the more general software maintenance; or a detailed requirements phase within the data engineering life cycle.

At this point, we can assume that there is some shared understanding of the requirements – not necessarily a full detailed consensus, but a general appreciation for the tools and techniques required to produce a satisfactory solution. This is not an unreasonable assumption, as in most cases some contractual negotiations will have preceded a team starting on a development, or the new development will be part of a rolling series of features given to an in-house team of engineers working on a particular project.

The second step of the setup phase is to consider the resources that should be shared between software and data engineers. Typically, this will include requirements or specification in the form of models, or perhaps metamodels, that can be shared rather than creating two incompatible versions. Unifying terminology and semantics is important here: if software and data engineers have differing interpretations of the same model, any potential advantage may be lost. Creativity in this part of the process may result in gains later on: other potentially sharable resources may include test suites and other quality analysis tools, technical and user-facing documentation, and project management tools or support. As with all analysis carried out in this setup phase, it can be revised in later iterations, and so any decision taken here need not be final.

The third step is to consider the integration points for this particular project, in the context of the decisions made in the previous two analyses. Given iterative approaches to software and data processes, and a list of shared resources, it is possible to build a grid, similar to that shown in Figure 3.6. The software engineering life cycle steps are enumerated along the top, and the

| Data Engineering | Software Engineering | | | | |
|---|---|---|---|---|---|
| | Requirements | Specification | Design | Verification | Maintenance |
| Manual revision/ Author | | | | | |
| Inter-link/fuse | | | | | |
| Classify/Enrich | | | | | |
| Quality Analysis | | | | | |
| Evolve/Repair | | | | | |
| Search/Browse/ Explore | | | | | |
| Extract | | | | | |
| Store/Query | | | | | |

**Figure 3.6** An incomplete grid for analysing integration points.

data engineering life cycle steps are enumerated on the left-hand side. Each box in the main part of the grid therefore corresponds to a potential integration point – for example, the first column in the first row represents a potential synchronisation between the requirements phase in software engineering with the manual revision/authoring phase in data engineering.

The grid can now be populated with two pieces of information. The first is to highlight any squares in which a potential integration point is possible – this will be based on the shared resources analysed in the previous step. For example, if a data schema is to be shared, then any changes made as part of the specification phase could impact some or all the data engineering phases. Similarly, any shared test cases which are updated as part of the quality analysis phase in data engineering, will affect the verification phase of the software development process. The second piece of information is the tooling that can be used to facilitate the integration at each point in the grid. In Section 7, we outline some of the tools built by the ALIGNED project that can be used to support and manage these integration points, but appropriate tools may be found off-the-shelf, repurposed from software or data engineering, developed in-house, or built for this specific development. As the need for data-intensive systems development increases, it is expected that such tools will be more widely available.

It should be obvious at this point that any identified integration point without specific tool support may need addressing. In many cases, simple awareness could be sufficient: highlighting such unsupported integration points and ensuring greater effort on collaboration at these points in the process. Alternatively, new tools could be sourced, or processes adjusted to minimise potential integration.The fourth and final step in this setup phase is

to consider the other barriers to harmonisation, in the context of each integration point. Software and data engineers involved in the project should come together to consider how their terminology, standard models, developer roles and tools can be made compatible in order to ensure maximum integration at each feasible point.

### 3.5.2 An Iterative Methodology and Illustration

Once the setup phase is complete, a more traditional iterative development can begin. In the setup phase, an iterative process for each of the software and data engineering components was selected. In our methodology, these may now continue independently in parallel, but constrained by the integration points previously discussed: overlap, synchronisations, and dependencies. To ensure that these integration points may be sufficiently addressed, it is our recommendation that the cycles are aligned, or are coincident at a particular phase in each cycle – this will be determined by the integration points, and the shared resources.

To illustrate, we consider a typical scenario encountered by our ALIGNED project use cases. In this scenario, the software engineering process is approximately equivalent to the iterative methodology in Figure 3.3, and the data engineering process can be seen as similar to that defined in Figure 3.4. The key shared resource is a complex data model, used as a reference by the data engineers, but also forming part of the software model: data modification functionality, business rules, and additional internal data points are added to the external-facing data model, and used as a specification document for the software engineers.

In such a process, updates to the data model can occur as part of the storage/querying phase of the data engineering activity, where new data are added to the existing data corpus, or as part of the specification phase of the software engineering activity, where new requirements give rise to updates in the intended functionality of the system. This forms the key integration point: there is an overlap in process here, as both software and data engineers should agree on any updates to the data model, and neither may continue until the updates made are complete and consistent. It is important that such a key integration point is well managed: problems here could result in wasted time and effort in curating a dataset against an incorrect model, developing software against an invalid or inconsistent schema, or managing a difficult merge operation between two parallel versions of the same data model. However, managed properly, having a shared data model is worth the

effort: a reduction in duplication can save time and money; automation based on this model can be shared; a common understanding can lead to a more coherent, better designed solution.

In this scenario, we insist that iterative processes in software and data engineering may now continue independently, but must synchronise on this overlapping event: storage/querying and specification. Figure 3.7 shows an example of such a parallel, synchronising process. In theory, this means that the iterations of each process should be the same length, and while in some projects this may be feasible, in others, where a particular phase may be more burdensome, this may prove to be overly restrictive. In such situations, it may be possible to relax this guideline, by simply insisting that the iterations synchronise whenever a change affecting both processes is made to the data model. For example, after a major release of a combined software and data product, minor, or patch releases may be made to the software if no changes are made to the data model, or any changes made do not affect the current iteration of data engineering. This will allow the software engineers to iterate a few times within a single iteration of the data engineering process, ensuring that data engineers have time to satisfactorily complete their iteration, and that software engineers are not kept waiting before beginning a new iteration.

Such synchronous iterations must be managed with care – those managing the projects must be made aware of any potential delays, since a delay to one process will impact the other. In software engineering, developers may be used to working within time-bounded "sprints" – in which the scope of a release may be reduced in order to ensure that completion is not delayed.

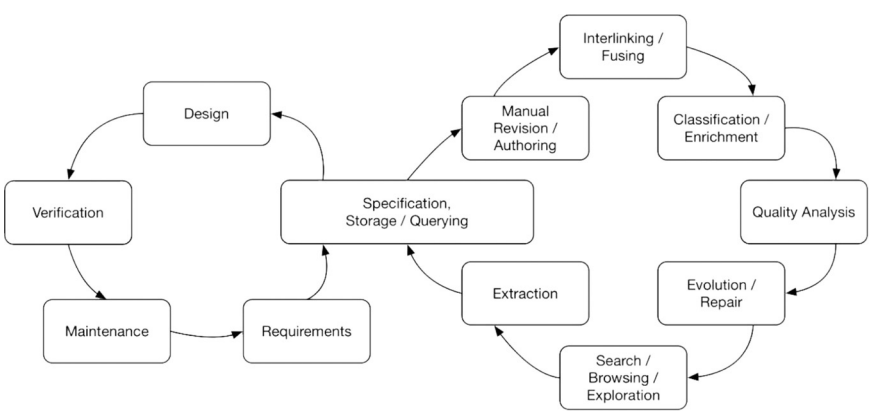


**Figure 3.7** A parallel life cycle with synchronisation.

In data engineering, such practices are less common, and so some training may be required to ensure all technical staff understand the restrictions. In developments where software and data iterations coincide, but are of differing lengths, care must be taken to ensure that any additional iterations do not impact the shared resources. For example, in the scenario outlined above, any additional software iterations for a minor or patch release must not update the shared part of the data model, for otherwise the current data engineering iteration may be inconsistent with the software that will next be deployed.

## 3.6 Recommendations

The iterative approach outlined above can provide a framework for combining software and data engineering processes, in such a way that a certain amount of autonomy can be maintained in two quite separate disciplines, but also in a way that can improve consistency and efficiency in the delivery of a solution made up of two closely coupled components. We now give some recommendations, based on our experience on a number of use-case projects, for ensuring that integration points are managed efficiently, and to maximise collaboration between software and data engineers.

Our first recommendation is that models are shared between software and data specifications, wherever possible. As previously discussed, this increases the opportunities for reuse and helps ensure that software and data remain consistent. We further recommend that these models are formalised in such a way that removes ambiguities, reducing the chance of inconsistent assumptions being made by software and data engineers.

Second, we recommend that development is driven by these shared models, in an automated fashion wherever possible. This reduces the chance of error in development and can ensure consistency such that developers can rely on the solutions produced in a parallel iteration.

Third, any solutions for either software or data should be rigorously tested, where tests are also developed – automatically if possible – directly from the model. Sharing or reusing test components can prove efficient, as well as ensuring consistency between data and software.

Fourth, tool support should be used to effectively manage the iterative process on both software and data sides. As discussed in Section 5, software engineers are used to using project management software to coordinate and administer an agile process, but such tools are not commonly used in data engineering applications. Such tools would need specialist support for

managing the integration points, and a wider range of developer roles and responsibilities.

Our final recommendation is that whenever meetings are held to discuss the iterative process – in particular the planning and feedback stages – these meetings should be attended by representatives of all solution stakeholders. The purpose for this is twofold: so that integration points and shared resources can be carefully managed; and so that the overall roadmap and architecture can be maintained whilst engineers focus on small iterations addressing short-term goals.

These five recommendations are derived from the combined experience of the project use cases, but in every project, their priorities differed, according to the experience of the development and project management teams, the tools available, and the particular iterative steps used in each development.

### 3.6.1 Sample Methodology

As an illustration, in this section, we look at the synchronisation points required for the ALIGNED use cases.

Table 3.1 outlines the usecase-oriented view of the synchronisation between Data and Software Engineering life cycles. Each entry of the table represents a synchronisation point within in the project. The use cases will be

**Table 3.1** A usecase-oriented synchronisation table for the ALIGNED project

| **Data** | **Software Engineering** | | | | |
|---|---|---|---|---|---|
| **Engineering** | **Requirements** | **Specification** | **Design** | **Verification** | **Maintenance** |
| **Manual revision/ Author** | | PS5.1, PS5.2, JURION [WKD1] | PS5.1, PS5.2 JURION [WKD2] | JURION [WKD3] PS1.4, [Seshat1] | |
| **Inter-link/ fuse** | | PS4.1, PS4.2, DBpedia [DBP1] | PS4.1, PS4.2, PS4.4 | | DBpedia [DBP2] |
| **Classify/ Enrich** | | | | | [Seshat2] |
| **Quality Analysis** | | JURION [WKD4] DBpedia [DBP3] | | PS1.1, PS1.2, PS1.3, PS2.3, PS3.1 | PS1.1, PS1.2, PS1.3, PS2.3, DBpedia [DBP4] |
| **Evolve/Repair** | | | | PS5.3 | PS3.1, PS3.2 |
| **Search/Browse/ Explore** | | PS5.1, PS5.2 | PS5.1, PS5.2 | DBpedia [DBP5] | |
| **Extract** | | PS4.1, PS4.2 | PS4.1, PS4.2 | DBpedia [DBP6] | |
| **Store/Query** | | PS5, JURION [WKD5] | | JURION [WKD6] DBpedia [DBP7] | |

used to enact the methodology with the tools in Section 7. The following summary describes the high-level features of each intersection point, in terms of use cases:

- Manual Revision/Author
  - A2: [WKD1] In the schema change use case (JS7), it is reflecting the situation that when a schema change is introduced and forwarded to the software manager in the SE life cycle, which initiates a process of validating the suitability of the model for use in SE. [WKD2] In the bug reporting governance use case (JS8), when a bug is reported and the software analyst finds that the bug is caused by a data error, he informs the DE expert to fix the data error via manual revision. [PS5.1] Develop plugins for Confluence and JIRA [PS5.2] Make use of collected process-related data.
  - A3: [WKD2] In the bug reporting governance (JS 8) use case, the SE designer can eliminate scenarios where a data-caused bug could occur in the future by sending additional constraints to the DE side, where these constraints are integrated to the schema. [PS5.1] Develop plugins for Confluence and JIRA [PS5.2] Make use of collected process-related data.
  - A4: [Seshat1] We will implement of graphical user interface software to author and edit data and the data will be communicated and captured in the DE life cycle.
- (B) Interlinking/Fusing
  - B2: [PS4.1] Extract data from Confluence and JIRA [PS4.2] Create RDF data from the extracted data. [DBP1] Refers to the fact overlap and conflict evaluation (DS1.3) and in the interlink evaluation (DS3.2). DS1.3 refers to validation by fusing data from different DBpedia language editions and Wikidata in order to identify overlaps and conflicts. DS3.2 refers to tools that validate external links to other datasets.
  - B3: [PS4.1] Extract data from Confluence and JIRA [PS4.2] Create RDF data from the extracted data [PS4.4] Link Development Process Data with Data Model Integrity Information.
- (C) Classify/Enrich: There are few synchronisation points where DE use cases exploit SE tools, possibly because classification in DE is a well-studied task.

  - A5: [Seshat2] The graphical user interface software widgets on the SE side will be continuously updated and maintained as the DE schemas evolve.

- (D) Quality Analysis
  - D2: [DBP3] Quality analysis for mapping (DS2.1), ontology (DS2.2) and instance data (DS3.1).
  - [WKD4] When a quality-related schema change is introduced and accepted in the DE Life Cycle, the changes are communicated to the SE Life Cycle, where the software is accepted. There is a protocol for accepting quality changes.
  - D4: [PS1.1] Constraints for Internal Actions [PS1.2] Rules for Reasoning and Inferencing [PS1.3] Constraints for Specific Schemas [PS2.3] Validate Thesaurus Against Schema.
  - D5: [DBP4] Schemas refers to reports, generated by the automated mapping validation tool (DS5.1) and erroneous fact report to the Wikimedia community (DS5.2). [PS1.1] Constraints for Internal Actions [PS1.2] Rules for Reasoning and Inferencing [PS1.3] Constraints for Specific Schemas [PS2.3] Validate Thesaurus Against Schema.
- (E) Evolve/Repair
  - E4: [PS5.3] Integrate Data Constraints Information with PPT Data Migration and Deployment Strategy.
  - E5: [PS3.1] Formulation of Constraint Violation Repair Strategies [PS3.2] Creation of Repair User Interfaces.
- (F) Search/Browse/Explore
  - F2: [PS5.1] Develop plugins for Confluence and JIRA [PS5.2] Make use of collected process-related data.
  - F3: [PS5.1] Develop plugins for Confluence and JIRA [PS5.2] Make use of collected process-related data.
  - F4: [DBP5] These integration points use the generation of DataID as a core and auto generate tool for browsing and querying based on the DataID file. Browsing is achieved by auto generating a download page for a DBpedia release and querying by providing a Docker image that contains the release stored in a triple store.

- (G) Extract
  - G2: [PS4.1] Extract data from Confluence and JIRA [PS4.2] Create RDF data from the extracted data.
  - G3: [PS4.1] Extract data from Confluence and JIRA [PS4.2] Create RDF data from the extracted data.
  - G4: [DBP6] Extraction of two additional Wikimedia projects: Wikimedia Common (DS1.1) and Wikidata (DS1.2), implementing tools in the SE domain that extract the data.
- (H) Store/Query
  - H2: [PS5.1] Develop plugins for Confluence and JIRA [PS5.2] Make use of collected process-related data [PS5.3] Integrate Data Constraints Information with PPT Data Migration and Deployment Strategy.
  - [WKD5] This integration point appears in the schema change (JS 7) use case. Once the schema change is in place in the DE Life Cycle, new instance by the DE expert to the SE expert. The new data are used to execute test scenarios on how the new schema is affects the existing software, to formulate new requirements for the design and implementation phases.
  - H4: [DBP7] These integration points use the generation of DataID as a core and auto generate tool for browsing and querying based on the DataID file. Browsing is achieved by auto generating a download page for a DBpedia release and querying by providing a Docker image that contains the release stored in a triple store.

A1–F1: The planning phase of the software engineering life cycle does not contain any synchronisation points. Possibly because there are few tools for this stage (in general) artefacts produced at this stage are informal and documentary, and not useful to Data Engineering processes.

## 3.7 Sample Synchronisation Point Activities

As example, tools from the synchronisation table and details of the changes made are included below, in the Model Catalogue tool and Semantic Booster. The aim of the following sections is to demonstrate the methodology using the example tools. The implication of iteration in the life cycles is also discussed.

### 3.7.1 Model Catalogue: Analysis and Search/Browse/Explore

The Model Catalogue Tool has been developed for use cases supporting model driven software engineering. The main purpose is to capture, document, and disseminate models including software systems, data standards and data interchange formats, amongst others. The interface of the Model Catalogue is shown in Figure 3.8. The tool helps end users to analyse the available models and understand requirements for capturing new data against existing models. In the standard version, models can be imported

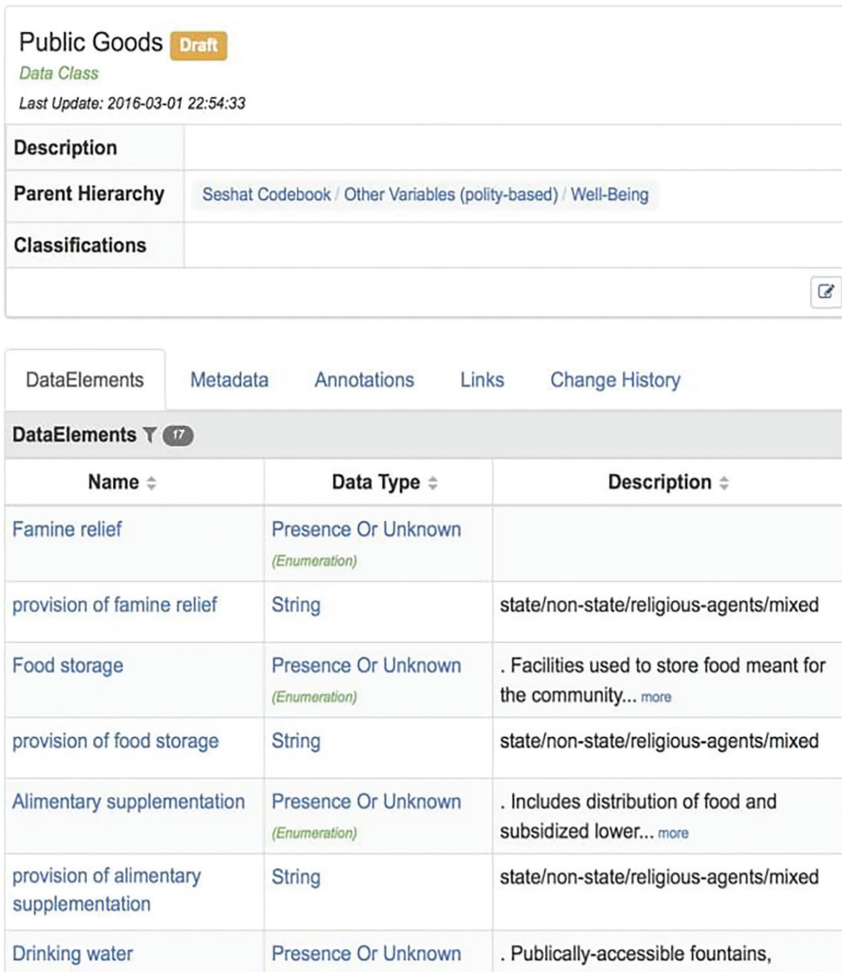


**Figure 3.8** Model catalogue interface: browsing the SESHAT code book.

from formalisms such as UML and XSD. Models may be interlinked and reuse elements from related models. Some of the output formats include Booster for software generation and Microsoft Word for documentation of the model.

The catalogue tool has been adapted to support similar data engineering use cases, and thus bridge between data engineering and software engineering domains. The main addition has been the import and export of models in standard data engineering formats, such as RDFS and OWL. For the data engineer, the tool can be used to explore how their data models are used in practice in software. The models can be updated and changed without relying on software engineers to create new versions of software. Models exported using the catalogue will retain interlinks between models in the two domains. This allows more streamlined integration of semantic metadata into working software.

The synchronisation point is bi-directional. The models can capture a software model from a data engineering model or use the model to capture data in the data engineering domain. Multiple iterations of the software engineering and data engineering life cycles will typically result in new versions of the model; the changes will need to be synchronised after each iteration. A feature to compare the changes in models in the model catalogue is planned to support this activity.

### 3.7.2 Model Catalogue: Design and Classify/Enrich

In model-driven development, the model catalogue tool also supports the creation of new models for capturing emerging designs for data standards, software systems, and so on. The tool supports definition of new data classes and data elements that form the basis of data models. The tool has features such as model versioning, annotation, collaborative editing and communication between developers. The models can be built using existing models in the catalogue or imported from partial models that exist in semi-structured and human-readable formats such as spread sheets, CSV or text documents.

The model catalogue has been adapted for data engineering activity: classify data and enrich data models by linking elements with existing model elements. Model classes can be refined and developed in the catalogue, capturing new and emerging structures in a data model, which leads to more precise understanding of the domain. Data engineers can use the catalogue to link between concepts in separate data engineering standards, and decide where links are semantically appropriate.

Similar to the "Analysis and Search/Browse/Explore" synchronisation point, this sync point is bi-directional. Iterations of the software and data engineering life cycles can result in new versions of the models. The model catalogue compare feature will support synchronisation of independent changes in models across both life cycles.

### 3.7.3 Semantic Booster: Implementation and Store/Query

Booster is a tool for the model-driven generation of information systems. High-level specifications are developed in Booster notation, which models the system implementation. Booster performs a series of translations and refinements on the model to generate a working system and Application Programming Interface (API) backed by a standard relational database. A user interface to the system is provided as an example of how the API may be used. The tool is used in the software engineering life cycle at the implementation phase.

Semantic Booster is a set of modifications to the Booster framework to support some data engineering life cycle activities. The changes add support for semantic annotation to standard Booster specifications, as shown in Figure 3.9: Example Semantic Booster System with Annotations. The Booster translations have been adapted to present the data as triples, with a SPARQL endpoint. The data in such a Booster system can be accessed and queried using standard data engineering toolsets. In combination with the design activity supported in the model catalogue, data engineering tools can be generated automatically using semantic Booster.

This synchronisation point is unidirectional at the model level, as MDE provides an implementation for the data engineering domain. The created implementation will be used by subsequent stages in the data engineering domain. As the mapping into triple form created by Booster uses a live version of the data, subsequent data engineering life cycle phases will access the version of latest data. Any modifications to the data must performed via the Booster generated API.

### 3.7.4 Semantic Booster: Maintenance and Search/Browse/Explore

In model-driven software development, maintenance and adaption of existing systems is a challenging task. Any changes to a Booster specification must be reflected in the implemented system, which can require re-generation of

examplesystem.boo2

```
1 @prefix rdfs: http://w3.org/.../rdf-schema#
2 @prefix snomed: http://snomed.info/sct/
3
4 @mappings
5 //a snomed `Information system software'
6 examplesystem  snomed:706594005
7 Patient snomed:116154003
8 Patient.name rdfs:label
9 Patient.name snomed:371484003
10 Patient.nhsno snomed:395451000000101
11
12 system examplesystem
13
14 class Patient {
15   attributes
16     name : String
17     nhsno : String
18 }
19
20 class Clinic {
21   attributes
22     name : String
23 }
```

**Figure 3.9** Example semantic booster system with annotations.

the implementation. The Booster approach ensures that any data entering a system are always validated to conform to the constraints. A large or complex change to the model involves the migration and validation of existing system data. Previous experiments with Booster have shown that for some model edits, existing data can be migrated automatically.

The data in Semantic Booster are presented as triples. Using the Booster mechanism for migration, automated migration of triple data in the Booster system becomes possible. Once data have been migrated, tools from the data engineering world can be used to validate the migration for compliance with the semantic rules of the model.

This synchronisation point is bi-directional. In subsequent iterations of the software and data engineering life cycles, the model catalogue will capture changes to the model. Booster will use the changes to automate migration of data stored in the Booster system; the data will be presented both in the API of Booster and as triples.

## 3.8 Summary

### 3.8.1 Related Work

That software and data engineering life cycles should be more closely integrated are not a new observation: Cleve et al.[26] took a more concrete approach and also proposed a number of contemporary challenges in system evolution, based on higher levels of tool support; better tooling for co-evolution of databases and programs; more agile coding techniques; and aligning data orientation through Object-Relational Mappings.

A more general-purpose approach to integrating life cycles elicits a number of broader challenges: software-engineering aims of software quality, agility and development productivity may conflict with data engineering aims of data quality, usability, and user productivity. Such is the importance of this integration work, the NESSI has identified "Collaborative Service Engineering based on the convergence of software and data" and "Integration of Big Data Analytics into Business Processes" as EU research priorities.[27] Further challenges relating more specifically to Big Data applications have been identified by Chen and Zhang:[28] in particular, those relating to data capture and storage, curation and analysis are of relevance here: hardware as well as software limitations can impact the effectiveness of Big Data techniques and highlighted opportunities may be missed.

Auer et al.[29] identified challenges within the domain of life cycles for Linked Data. 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 on software development: the authors mentioned component integration, the management of provenance information, abstraction to hide complexity, and artefact generation from vocabularies or semantic representations.

---

[26] A. Cleve, T. Mens, J.-L. Hainaut, Data-intensive system evolution, Computer 43(8), pp. 110–112, 2010.

[27] NESSI, Strategic research and innovation agenda, Tech. rep., NESSI, version 2.0, April, 2013.

[28] C. P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, Information Sciences 275 pp. 314–347, 2014.

[29] S. Auer, J. Lehmann, A.-C. N. Ngomo, A. Zaveri, Introduction to linked data and its lifecycle on the web, in: Reasoning Web. Semantic Technologies for Intelligent Data Access, pp. 1–90, Springer, 2013.

Mattmann et al.[30] used their experience of data-intensive software systems across a range of scientific disciplines to identify seven key challenges:

- 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 pointed out the many interdependencies between these problems. Zaveri et al.[31] took 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[32] agreed with this issue, describing a more wide-ranging lack of support for developers of data-intensive systems. He also identified "the necessity of a multidisciplinary team that provides expertise on a diverse set of skills and topics" as a non-technical issue that can be addressed by projects dealing with large, distributed datasets. A technical equivalent to this issue is to understand notions of iteration with respect to the data modelling – Anderson argued that domain knowledge is required to understand data collection and curation. Subsequently, he also argues for technical knowledge

[30] C. A. Mattmann, D. J. Crichton, A. F. Hart, C. Goodale, J. S. Hughes, S. Kelly, L. Cinquini, T. H. Painter, J. Lazio, D. Waliser, et al., Architecting data-intensive software systems, in: Handbook of Data Intensive Computing, pp. 25–57, Springer, 2011.

[31] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, S. Auer, Quality assessment for linked data: A survey, Semantic Web 7 (1) pp. 63–93, 2016.

[32] K. M. Anderson, Embrace the challenges: Software engineering in a big data world, in: Proceedings of the First International Workshop on BIG Data Software Engineering, pp. 19–25, IEEE Press, 2015.

in order to match frameworks with requirements; emphasising the need for a multi-disciplinary team.

Some solutions to these challenges have been identified – most notably in the area of model-driven software engineering, DSLs, 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.[33] identified two issues: firstly the interaction between domain experts and application developers, and secondly that changes to schema code may not always impact application code in a straightforward manner. In this document, we attempt to tackle these two issues explicitly.

## 3.9 Conclusions

We have described a flexible methodology for integrating software and data engineering life cycles, identified a number of barriers to harmonisation, and made recommendations in order to better implement the combined methodology, and reduce the impediments. The methodology reflects the observed practices and experiences of the ALIGNED consortium – across a range of application domains, development practices, and experiences, both for the development of new solutions and the evolution of existing ones. We outlined the application of the methodology in each of the use cases, describing the particular challenges and requirements faced by each, and how the use of the methodology has improved development practice. We also described a number of tools built by the ALIGNED project partners that have been adapted to fit the integration points in the methodology, showing how they may be repurposed, or similar tools may be adapted for application to data-intensive systems.

The use cases presented here represent a small fraction of the potential application domains: further work is to apply the methodology in a wider range of projects, with a different selection of tools, and with different development teams. Further validation may be obtained from more qualitative or quantitative validation: although it is rare for two system developments to

[33] D. Qiu, B. Li, Z. Su, 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, pp. 125–135, ACM, 2013.

be directly comparable, experienced developers may be able to evaluate the effectiveness of the methodology against previous practice.

As discussed above, the software engineering life cycle is relatively mature and is broadly similar in all developments, but the data engineering processes are less well-defined, and may be more varied in further real-world applications – perhaps differing by domain or toolsets used. Further investigation is necessary to ensure that the methodology presented here is applicable to different data engineering practices.