Chapter 1

Engineering Ontologies with Patterns – The eXtreme Design Methodology

Eva Blomqvist
Linköping University, Sweden
Karl Hammar
Jönköping University, Sweden
Valentina Presutti
ISTC-CNR, Italy

When using Ontology Design Patterns (ODPs) for modelling new parts of an ontology, i.e., new ontology modules, or even an entire ontology from scratch, ODPs can be used both as inspiration for different modelling solutions, as well as concrete templates or even "building blocks" reused directly in the new solution. This chapter discusses how ODPs, and in particular Content ODPs[1], can be used in ontology engineering. In particular, a specific ontology engineering methodology is presented, which was originally developed for supporting ODP use. However, this methodology, the eXtreme Design (XD), also has some characteristics that set it apart from most other ontology engineering methodologies, and which may be interesting to consider regardless of how much emphasis is put on the ODP usage. Towards the end of the chapter some XD use cases are also reported and discussed, as well as lessons learned from applying XD. The chapter is concluded through a summary and discussion about future work.

1.1. ODP-based Modelling – Different Aspects of ODP Use

ODPs were originally proposed partly as a result of observing how difficult it is to reuse a large ontology. This observation even includes foundational ontologies

[1]In fact, throughout this chapter when mentioning ODPs, this mainly refers to Content ODPs if not specified further.

clearly designed for being reused as the basis for building other ontologies. Issues include that it is difficult to get an overview of such large ontologies, foresee effects of changes or extensions to them, and it is also rarely the case that you as an ontology engineer, or the set of requirements you have for your ontology engineering task at hand, will fully agree with all the ontological commitments that are made in such a large ontology. However, not reusing any well-established practices at all, and not aligning yourself at least partly to existing ontologies, will create problems in interoperability and potentially also understandability of your ontology. Hence, there is a trade-off between interoperability on one hand and overcommitment and conflicting requirements on the other hand, where ODPs as small "building blocks" offer one way to manage this trade-off. This is true regardless of whether reuse is discussed in a global Semantic Web context, or internally in an organisation, for instance. Hence, the idea of reusing, applying and sharing small patterns instead of complete ontologies, applies in many contexts. This is why, later when the XD methodology is discussed, it is mentioned that an important step is to decide what ODPs to use and how to create and manage your own ODPs, rather than there being a universally agreed set of ODPs that every project should use.

There are also many different types of ODPs, and they can be reused and applied in many different ways. Even when considering only what is called Content ODPs, i.e., ODPs that focus on modelling solutions on the conceptual level, and may constitute "building blocks" for your ontology, there are a variety of ways that these can be reused and applied. At one end of the scale, ODPs can be used similarly to design patterns in architecture, or how patterns many times are used in software engineering, i.e., as mere inspiration and a conceptual framework to keep in mind when designing your own solution. An example of this way of applying a pattern would be to read about its basic idea in a book, or an online catalogue, incorporate this idea into your own knowledge, and then proceed to design your artefact according to your own interpretation of that pattern, with any modifications you see fit. This way of reusing patterns is sometimes denoted *reuse by analogy*.

At the other extreme, some ODPs (in particular content ODPs) can be directly reusable through their OWL building blocks. This means that there is a small ontology available that represents the ODP, which you can directly import and use in your own ontology. This is similar to reusing an existing ontology, with the main difference that the ODP is usually small, has clear documentation of its capabilities, consequences, and so on, it is indeed designed for reuse, and ideally makes a minimal ontological commitment outside of its core purpose. In this case there is no way for the ontology engineer to make modifications to the ODP OWL-file, and you are affected by any changes that the ODP authors might make at a later date if you are resolving the ODP URI on the Web. Including the possibility of the ODP disappearing, i.e., that you at some point end up with a broken link in your ontology. However, the fact that ODPs are designed and published with reuse as their main purpose to some extent mitigates this risk, i.e., they change less frequently than other ontologies and are often hosted in ODP repositories, which reduces the risk of broken links.

Many people also follow some middle path between these extremes, poten-

tially reusing the OWL building blocks of a few well-known and stable ODPs directly, but then creating their own "ODP catalogue" for their project for the rest of their needs, or even model the rest of the ontology in a more monolithic way. There are also ODP-based ontologies that do not import any ODP OWL building blocks, but align to them through axioms of the ontology, e.g., expressing equivalences between locally defined classes and ODP classes. In the latter case it is up to the user of the ontology to decide how much of the ODPs to actually take into account, formally.

In [7] Gangemi and Presutti noted five types of operations that could be performed on ODPs when reusing them; *import, specialisation, generalisation, composition, and expansion. Import* denotes the reuse of the ODP as a component in the ontology being built, usually making use of the `owl:import` functionality. *Specialisation* can either take place in the mind of the ontology engineer, i.e., creating your own version of the ODP that specialises concepts and relations for your domain conceptually, but without actually importing any ODP "building block". On the other hand, if the ODP is actually imported, specialisation can be done by creating subclasses and subproperties of the ODP classes and properties, and specialising its additional axioms using the new classes and properties. *Generalisation* is a less common operation, but may occur when creating or extracting ODPs from an existing ontology, hence, generalising a set of classes and properties to be reusable also in other contexts. *Composition* on the other hand is frequently used when applying ODPs. Because, unless you are creating a very small ontology, one single ODP is rarely enough to solve all the requirements of your intended ontology, and since ODPs focus on small pieces of the overall solution, you will inevitably have to compose, i.e., combine, several ODPs or ODP specialisations in order to solve the overall problem. *Expansion* is related to a similar task, i.e., to cover those parts of your requirements that are in fact not solvable by any ODP, or to extend the ontological commitment of an ODP while specialising it, in order to correctly capture the domain knowledge you are modelling.

Previous studies have also shown that there are various different ways of carrying out the above operations. One study focused on the specialisation task [10], and showed that there are two main strategies for specialising ODPs. In the property-oriented strategy both classes and properties are specialised, i.e., both subclasses and subproperties are created and usually domains and ranges of the the subproperties are set to those new subclasses. One could view this as a more complete form of specialisation, since almost everything (except potentially some general axioms in the ODP) is specialised. Hence, this also opens up the possibility of simply removing the import of the ODP module at a later stage, while still retaining much of the semantics of the model. The other common strategy is the class-oriented one, where only classes are specialised, i.e., subclassed, and then restrictions are set on those classes in order to express their relation to the properties already defined in the ODP itself. Here, no new properties are defined, i.e., no subproperties of the ODP properties, and hence, the ontology built heavily relies on the imported ODP. Also hybrid variants are of course possible. The main advantage observed of the property-oriented strategy is that it makes the module created more self-contained, and more independent of the ODP, and

axioms can be added to describe the new properties more in detail. The main advantage of the class-oriented strategy is the data interoperability that it creates, based on that the original properties are the ones that will be used in data expressed according to the ontology. However, there is a downside in terms of reasoning complexity, since the class restrictions commonly used with this strategy will usually increase reasoning complexity.

1.2. The eXtreme Design Methodology

The eXtreme Design (XD) methodology [15, 16] was initially proposed as one scenario in the context of the overall NeOn methodology [15]. Nevertheless, XD is sufficiently self-contained to be considered an ontology engineering methodology by itself, in the sense that it incorporates both project initiation steps, requirements analysis, development, testing and release of an ontology. Here the original version of the methodology is described, with some additions since its original publication, while in the following section some possible adaptations to the methodology for use in specific contexts are discussed.

1.2.1. Background and Underlying Principles

XD was originally proposed as a reaction to the focus on waterfall-like methodologies in ontology engineering, intending to introduce a new and more agile way of thinking about ontology engineering. It was initially inspired by methodologies from software engineering, such as the eXtreme Programming (XP) [17] and experience factory [2] approaches. However, despite being inspired by these, there are some main differences between software and ontology engineering that do not allow for a direct transfer of methodologies between the fields. One such difference being the fact that ontologies are always "white box", i.e., a developer or tester needs to be aware of and concerned with the internal workings of a module, while in software engineering modules can at some point be considered "black box", and the engineer needs mainly to be concerned with an interface exposing some functionality. Another difference is also the focus on conceptualisation in the case of ontology engineering, which may introduce a number of non-functional requirements for ontologies, e.g., in terms of domain coverage and accuracy, including naming of concepts, for instance. Naming does not affect the technical ability of an ontology to answer queries or perform reasoning, but it is highly relevant for humans to understand and make sense of both the ontology itself and the output of any task it is used for.

Nevertheless, XD has been inspired by agile software engineering methodologies, and applies a set of principles similar to those of, for instance, XP. As discussed in a later section, how closely such principles are followed certainly may depend on the context where the methodology is applied.

First of all, XD needs high involvement of a "customer", i.e., someone who is going to use the ontology later on, either directly or indirectly. It should be noted that such a "customer" may not necessarily be the end user or person paying for the potential system to use the ontology (customer in an economical

sense). In fact, since ontologies are rarely used in isolation, but rather as a part of some application or software system, the appropriate notion of "customer" may very well cover also the software developers who's components are to exploit queries and inferences made by the ontology. Nevertheless, domain knowledge is of course of essence when designing an ontology, hence, it is at least as important to also involve domain experts as customers in the development process. The customers are involved in requirements engineering, they have to confirm the detailed requirements developed, as well as verify that the end result corresponds to those requirements, e.g., both in terms of functionality as well as domain coverage, appropriateness of terminology, and other non-functional requirements.

Second, XD is highly requirements-driven, i.e., the ontology should contain exactly what is needed and nothing else. In earlier publications [15] this was denoted "task-focused", meaning that ontologies built using XD are always focused on a certain set of tasks, rather than only being a representation of some knowledge domain, without any intended task for the formalisation of that knowledge. Note however, that tasks may be quite general, such as providing a schema for a set of data to be published on the web. Originally this was another reaction to some early ontology engineering methodologies that lacked such a requirement focus, and thereby allowed ontology engineering projects to go on for years and years, without ever knowing when the ontology would be "finished". In order for requirements to be realistic, they should also be based on "stories" that come directly from the customers, i.e., rather than invented by the ontology engineers themselves, which is a common mistake in ontology engineering as well as in software engineering.

Third, the XD methodology is iterative, and incrementally builds the end result. This means that the methodology will produce a tangible result early on, while then extending that result in each iteration. When XD was first proposed, this was also a novelty in the ontology engineering community, where many previous methodologies and ontology projects had been focusing on first achieving a complete coverage of requirements, then a complete domain coverage, e.g., in terms of domain terminology, and only thereafter starting to actually formulate some ontological axioms. XD has a short time to first release, hence, it is particularly appropriate as a methodology for "rapid prototyping" of ontologies. When used not only as a prototyping methodology, this of course has the consequence of instead increasing the amount of work that has to be spent on resolving conflicts and refactoring the ontology in later iterations, due to both changing requirements and the design choices made in the newer modules being built. However, as will be noted later in this chapter, using ODPs is actually one way of partly mitigating the risk of having too diverse modules in the end.

Nevertheless, the ontologies created using XD will be inherently modular, since they are created piece by piece, and integrating the pieces one by one. Modularity can be an advantage in terms of easier reuse of parts of the ontology, and opening up the possibility to only use parts of the ontology to perform some "local" inferencing and querying. However, technically the structure of the ontology might become quite complex. In particular if each module is given its own namespace - which would be ideal if only considering the (re)use aspects mentioned. This may not be ideal from a human communication perspective, since an

ontology could consist of tens or even hundreds of modules, it will become quite difficult to keep track of what namespace contains what module(s). In this case it may be more intuitive to simply use the "module" notion as a conceptual one, rather than technically separating them into separate ontologies.

Next, XD is test-focused, i.e., testing against requirements is a central part of the development. Whereas many agile software engineering methodologies are really test-driven, i.e., all tests are created even before the software is designed and built, this is not really possible when it comes to ontology engineering. The main reason is that since, as mentioned earlier, ontologies are "white box", writing a test requires knowledge of the internal structure of the ontology, hence, writing a test then implies assuming a certain internal structure, thus designing the ontology itself. For this reason, only generic tests, at a structural level, can be created before the ontology is designed, as proposed in [14], while selecting which tests are applicable and how to best test the actual domain conceptualisation has to be done later. However, XD still has a clear test-focus, in the sense that all (functional) requirements written should be testable, so that the fulfilment of those requirements indicates the completion of a module or the ontology as whole. Of course, then XD suffers from the same difficulty as software engineering when it comes to testing non-functional requirements. In the ontology case a non-functional requirement could be the appropriate use of domain terminology in naming elements of the ontology. Although this is hard to test, and there may not even be a clear answer since terminology is not always agreed even among a set of domain experts, it is important to agree on a procedure to confirm also that such requirements have been met.

Another of the core principles of XD is reuse, and in particular reuse through ODPs. Ontology reuse in general is difficult, due to both overcommitment of the reused ontologies, and the sheer size of ontologies to understand and reuse, as discussed previously. Reusing ODP building blocks, or even ODPs just as ideas of solutions, is much more flexible but still lets the ontology engineer benefit from previous solutions. In addition, XD encourages also the development of ODPs, both specific and generic, which could be the starting point of a "component library" of reusable ontology modules, for instance within an organisation. Such a library can then constitute a "common language" for ontology design in that organisation, c.f. the notion of a pattern language [1]. However, this could also be useful even if applied only within a single ontology development project, in order to reduce the need for refactoring and alignment in the module integration phase.

In addition, the divide-and-conquer paradigm that is inherent to XD leads to a natural modularisation of both the problem, and its solution, which facilitates distributed ontology development, and assists in scoping the modelling issues that are addressed within a single iteration. However, to handle this incremental and potentially distributed ontology development, integration and refactoring become essential tasks. To some extent one could view XD as pushing some of the hard decisions of ontology engineering towards the end of development, instead of solving them before starting to model, i.e., conflicts and inconsistencies are only dealt with after they become a concrete problem in the integration of new modules, rather than beforehand. Taken to its extreme, developers of a certain

module are not to take into account any requirements outside their small story and their set of requirements. However, in reality, and in particular when using a shared catalogue of ODPs, there may still be some communication around suitable design choices among development teams, which in turn may reduce the need for later refactoring.

The latter puts some focus on the collaboration model of XD, which promotes pair design, comparable to pair programming in XP, but tries to limit the interaction between design pairs during module development, in order to keep their focus and limit them to the scope set by their particular requirements. Nevertheless, since integration and refactoring is such a crucial part of the methodology, and this is not necessarily performed by the same set of people, documentation and proper communication of motivations of design solutions is essential. XD is suitable for distributed development, e.g., by geographically distributed teams that collaborate online, however, it is important to set up appropriate communication channels for sharing solutions, ODPs, and discussing refactoring issues.

In the following sections, the steps of the methodology are described in detail. Roughly the methodology can be divided into three parts; (1) a project initiation and scoping step, which is only performed once, at the start of the project (although may have to be revisited if conditions change), (2) a development loop that iteratively produces new modules, and (3) an integration loop that adds the increments to the overall solution. An overview of the methodology workflow can be seen in Fig. 1.1. In the following three subsections, first the project initiation and scoping steps are discussed, and thereafter details of the development and integration loops.

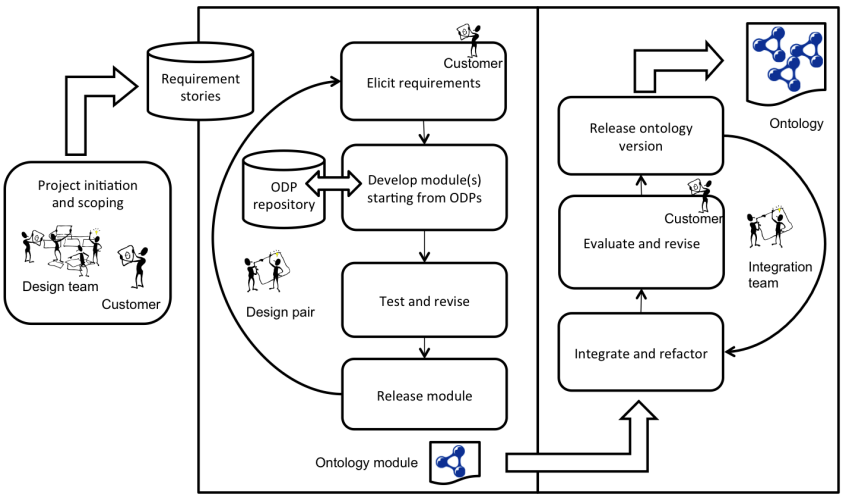

Figure 1.1. Overview of the XD workflow.

1.2.2. Project Initiation and Scoping

As with any development project, some general agreements have to be reached, if not given by the project specification itself. This can be things like staffing and distribution of roles among project participants, setup of an appropriate technical environment and decisions on representation languages and frameworks, agreement on the exact procedures to be followed within the project, including release plan and integration strategies, project scope and priorities, reuse opportunities, as well as a timeline with deadlines and milestones. At a high level an ontology engineering project does not differ much from any other development project. Nevertheless, some of the mentioned things to consider has a specific meaning in the context of ontology engineering and XD, which are worth mentioning.

One thing that sets ontology engineering apart from, for instance, many software engineering projects, is the need for a deeper understanding of the target domain, even among the developers and not only at the management level. This is due to that the knowledge to be modelled in the ontology usually needs to be understood to a larger degree by the ontology engineers in order to, for instance, use correct terminology in the ontology, and make appropriate design choices that correctly solves the set of requirements. Again, this boils down to the fact that an ontology is "white box", hence the users (whether software developers of the surrounding system, or end users of the system) may have to in turn understand and approve of the inner workings of what is constructed. The consequence of this is that there is a greater need for developing a shared understanding of the domain, its terminology, the intended tasks of the ontology and so on, in an ontology engineering project than in, for instance, many software projects. Therefore this should be taken seriously already from the project start, and this activity is also to be revisited throughout the project in order to clarify any potential misconceptions. Commonly, a shared tool is set up for the purpose of documenting and discussing both domain knowledge and requirements, e.g., the user stories, so that domain experts can closely follow and also participate in any discussions, even if distributed in terms of time and location. Such a tool could constitute a simple wiki, or a more advanced project management system coupled with requirements analysis tooling.

Additionally, scoping is very important for ontologies, but also very hard to clearly define in terms of the knowledge domain to be modelled. Here the task-focus of XD can be very helpful, allowing to focus on the generic tasks that the ontology should support, rather than the domain coverage in terms of concepts, attributes and terminology. This could for example be specified through its use by other system components, its roles in a system to be built, while more or less considering the ontology and its associated querying and inference software as a black box from the system point of view. Of course, there may also be ontology engineering projects with less clear scope, and where it is not yet clear what the ontology might be used for, but then it may be hard to use XD as the main methodology without some adaptation.

Further, before starting the actual development, one needs to agree on the starting point of the project, e.g. in terms of any existing resources to take into account, or even reuse, what ODP catalogue(s) to use, and how to manage the

shared set of ODPs that will emerge during the XD process. It is rarely the case that ontologies are constructed completely from scratch. Usually there are at least some legacy systems or terminologies to take into account, e.g., anything from implicitly shared vocabularies within and organisation, via legacy databases, to standards and already existing ontologies. How each such resource is to be managed has to be determined at the start of the project. Procedures that need to be made explicit could include to what extent ontology modules and tests are to be documented, what naming conventions are to be use, where files are stored and how versioning is to be managed, how the integration process is to be managed, quality assurance processes, and when and what to include in an ontology release.

At this stage, also other non-functional requirements of the ontology may be listed. These can be overarching requirements, such as how well the domain terminology needs to be represented in the naming of concepts and properties, how much documentation the ontology should contain and how that documentation should be written to serve later purposes, e.g., to display information in a user interface, any trade-offs between domain coverage and functionality that should be observed, requirements on the overall architecture of the ontology, the OWL-profile to be used etc.

Finally, before starting the development loop, some user stories need to be developed. User stories will later lead to the development of the functional requirements of each ontology module. Stories can be formulated in different ways, e.g., as examples of data for which the ontology is to act a as a schema, or describing some functionality that is to be realised based on the ontology (for the latter c.f. the "original story" in Table 1.1). The important thing is to keep them short and focused, i.e., on one concrete part of the domain knowledge, one specific task. If stories are too big they will have to be broken up into smaller stories before starting the development loop, in order to avoid the situation where one development pair would more or less develop the whole ontology. A typical story might contain anything from 1-2 sentences up to about two brief paragraphs of text. Additionally, stories need to be quite specific in order not to allow for too much interpretation by the ontology engineers. Stories can be written in collaboration between "customers" and ontology engineers, but should be driven entirely by the needs of the customers.

An example story from a research project where XD was applied [2] can be seen in Table 1.1. The context of the ontology is an "intelligent bathroom" environment that should proactively serve the user with context-dependent information related to current user needs and preferences. Since the original story contained quite a few aspects and tasks of the resulting ontology, it was in this case broken down into 10 smaller stories (only a few examples are shown in the figure for space reasons) that were made more specific, identifying exactly what was the task of the system in each step. These 10 stories were then used as the basis for the development of a set of ontology modules that would together support the functionality described in the original story.

All stories should then be organised in terms of priority, and possible dependencies between them are identified and made explicit. It is therefore suitable

[2] http://www.iks-project.eu/

Table 1.1. Example of the breakdown of a story that was originally too large, in terms of covering too many aspects and tasks of the ontology to be built.

Original story:	It's Thursday morning. I get site-specific weather information when I am brushing my teeth in the bathroom. Based on weather information and my calendar, free-time event suggestions are given (e.g. "Today, 8 p.m. - Sneak Preview at CinemaOne). Do you want to order tickets?
Substory 1:	I am alone in the bathroom. I am standing facing the mirror with the electric toothbrush in my hand, hence, the system recognises that I am brushing my teeth.
Substory 2:	I was brushing my teeth in the bathroom on Thursday morning. Morning is between 6 and 10am according to the current user.
Substory 3:	I am living in Berlin and I like to get the local weather displayed in the morning as soon as I am brushing my teeth.
...	...
Substory 10:	The system asks me if I want to order tickets to a proposed event if there are tickets available. I am presented with a set of ticket options, and get to select the time of the show, the seat I want at the movie theatre. Finally, I pay by credit card.

for each story to be described by means of a small card, like the one depicted in Table 1.2, which includes the unique title of the story, a list of other stories that it depends on, a description in natural language, i.e., the story itself, and a priority value. The example story in Table 1.2 comes from a course on ontologies in the legal domain[3]. Typically these stories can be collected using a form in a wiki setting, or a more elaborate requirements management system may be used. The two examples given here in Table 1.1 and 1.2 both phrase the story as a concrete example, i.e., containing elements that would be part of the knowledge base rather than the ontology. Although stories can very well be expressed in more general terms, it may often be easier for domain experts to provide concrete examples of data and how the system should react to that data, rather than providing descriptions of generic situations. Of course this comes with the risk of missing something generic that does not happen to be covered by the concrete example.

Since XD is agile and iterative, it is not necessary to develop all user stories beforehand, but an initial "backlog" is to be accumulated before starting the development process. This is to ensure that an appropriate prioritisation can be made between the initial set of stories. Simply picking up the first one being written may result in development starting with a very low-priority story that otherwise may not even have been included in the ontology at all. At this stage, it is also important to constantly update the agreement with the "customers", unless otherwise specified by formal constraints, such as a contract. As the set of stories is allowed to emerge and evolve over time, it is important to also update the agreement of what is actually going to be implemented. For this purpose, there are two main points of agreement between the "customers" and the project management; one is the agreement on what stories are to be prioritised, and in the end even what proposed stories to implement at all, and the second one is at

[3] http://ontologydesignpatterns.org/wiki/Training:Legal_Ontology_Design

Table 1.2. A proposed template for user stories, filled with an example story from the legal domain.

Title	Founding of legal entities
Dependencies	...
Priority	High
Story	FIAT (Fabbrica Italiana Automobili Torino) was founded in Turin in 1899 by Bricherasio, Goria, Biscaretti, Ferrero, Ceriana-Mayneri, Racca, Scarfiotti, Damerino, Agnelli and the Bank di Sconto e Sete. The deed of constitution of FIAT, showing the Savoy coat of arms and the original company name on its cover, was drafted by the public notary Ernesto Torretta, signed by all the founders, and registered on the 11th July 1899.

the level of agreeing on the detailed requirements of each story (see further in the next section).

1.2.3. Module Development Loop

Once some user stories have been collected and prioritised, the concrete development of the ontology can begin. As mentioned previously, this is done incrementally, one module at a time. Ideally, each story will correspond to one (or a small set of) ontology modules, however, the situation may also occur that some stories are considered too overlapping, so that their solutions have to be merged.

Nevertheless, the development of the stories can be done in parallel by as many design pairs, i.e., pairs consisting of two ontology engineers, as the project has access to. Pairs may of course communicate with each other, but ideally such communication is reduced to a minimum, and the focus of each pair should mainly be on their own story only, regardless of the requirements developed from other stories. This is important in order to avoid pairs getting stuck on issues that should actually be resolved later on, instead of developing what they feel is the best solution for their small sub-problem.

The module development loop is represented in Figure 1.1 by the left rectangle, where a design pair loops through the activities of requirements elicitation, module development, testing, and module release, for one story at a time. The first step is for the design pair to pick up a new story to work on. How this selection is done may vary, but priority of the story, dependencies on previous stories treated by the pair, and the skills and competencies of the pair, may impact the selection.

Once selected, the pair should perform requirements elicitation from their story. Of course, this process may need considerable involvement by the customers (who wrote the story) in order to interpret its intended meaning and ensure an appropriate coverage of this particular subset of the domain. If the story was formulated as a concrete example, e.g., as exemplified in the previous section, it is now time to generalise it. One way to perform such generalisation is to first reformulate the story sentences into *instance-free* sentences, i.e., to for each mention of a named individual or example attribute value, replace that with a

term representing the expected type of this individual or a potential attribute that could hold that value. Table 1.3 shows an example of such a transformation, based on the example story from the legal domain presented earlier.

Table 1.3. Example of how a set of phrases from a user story may be generalised.

Original phrase	**Example of instance-free phrase**
FIAT (Fabbrica Italiana Automobili Torino) was founded in Turin in 1899 ...	Legal persons are founded in a certain location at a certain time ...
... by Bricherasio, Goria, Biscaretti, Ferrero, Ceriana-Mayneri, Racca, Scarfiotti, Damerino, Agnelli and the Bank di Sconto e Sete.	... by a set of legal persons.
The deed of constitution of FIAT, ...	Legal entities are created by a deed of constitution ...
... showing the Savoy coat of arms and the original company name on its cover, ...	... which consists of text and pictures, ...
... was drafted by the public notary Ernesto Torretta, ...	... and is drafted by a public notary, ...
... signed by all the founders, and registered on the 11th July 1899.	it is signed by the founder persons and a certain point in time.

Once the story text has been sufficiently generalised, a set of requirements should be elicited from it. These requirements commonly fall into the following three categories:

- Competency Questions (CQs)
- Contextual Statements (CS)
- Reasoning Requirements (RR)

CQs are probably the most well-known category of ontological requirements, which was recognised already at the very beginning of the knowledge engineering tradition [8]. CQs express typical tasks of the ontology, i.e., typical queries it should be able to answer, and are usually expressed as natural language sentences, e.g., questions. However, through applying XD it has been noted repeatedly that CQs on their own do not always suffice in order to clearly specify what is required from the ontology, especially in the following two respects:

- Are there any constraints that should be enforced over this knowledge, or any common-sense notions that are to be introduced to complement the knowledge needed to answer the CQ? - Answers are CS
- Is all the information needed to answer the CQ going to be entered explicitly into the knowledge base, or is there some inferences required either in order to derive the answer to the CQ or that should be derived as a consequence of the response? - Answers are RR

Note that both of these questions refer to the CQ, hence the CQs are the requirements that set the scope of the module to be built and drive the need for additional requirements. However, CS and RR are sometimes needed in order to precisely specify the additional axioms of the entities mentioned in the CQs that are needed in order for the ontology to perform a certain task. Considering

a CS, the task may be consistency checking, or identity resolution - in addition to answering the CQ. While considering an RR, the task may for example be classification of instances, in order to then be able to answer the CQ based on the inferred knowledge. In Table 1.4 some example requirements are presented based on two of the example stories seen previously.

Table 1.4. Example breakdown of stories into detailed requirements.

Story	
I am alone in the bathroom. I am standing facing the mirror with the electric toothbrush in my hand, hence, the system recognises that I am brushing my teeth.	
CQ	**Associated CS or RR**
Who is where in this indoor location?	-
What sensor data is known about this user's context?	-
What is the user doing now?	RR: Activity inferred based on the available context information, from a fixed set of activities.
Story	
FIAT (Fabbrica Italiana Automobili Torino) was founded in Turin in 1899 by Bricherasio, Goria, Biscaretti, Ferrero, Ceriana-Mayneri, Racca, Scarfiotti, Damerino, Agnelli and the Bank di Sconto e Sete. The deed of constitution of FIAT, showing the Savoy coat of arms and the original company name on its cover, was drafted by the public notary Ernesto Torretta, signed by all the founders, and registered on the 11th July 1899.	
CQ	**Associated CS or RR**
Who founded a legal person?	CS: The "who" must also be a legal person.
When was a legal person founded?	CS: Each legal person was founded at exactly one point in time.
Where was a legal person founded?	CS: Each legal person was founded at exactly one location.
Who signed the deed of constitution of a certain legal person?	CS: Each deed of constitution is signed by at least one founding person.
What is contained in a deed of constitution?	CS: Content can be of types text of images.
Who prepared (drafted) a deed of constitution?	CS: Each deed of constitution was drafted by one or more persons.
When and where was a deed signed and registered?	RR: A deed is valid when it has been both signed and registered.

Before leaving the requirements elicitation activity, the set of requirements should be "signed off" by the customer, i.e., an agreement should be reached that these requirements are necessary and sufficient for considering the story as covered by a solution. This means that when these requirements are confirmed through appropriate testing the solution module can be considered complete.

The following development step constitutes the actual modelling, i.e., creating a solution covering all the requirements of this story. Depending on the size of the set of requirements of a story, and how disparate they are, it may at this stage be a good idea to select only one or a few of the CQs to treat initially, creating an incremental building process for each module as well. Once a small set of coherent CQs have been selected, the first task is to look for any existing ODPs that may match the requirements at hand. This matching task is supported by the fact that

most Content ODPs are annotated with a set of CQs in themselves. However, due to the gap in abstraction, i.e., ODPs being usually quite abstract solutions while the requirements at hand are usually much more concrete and domain-specific, providing good tool support for this matching process is difficult and currently it is therefore mainly a manual task. ODP repositories, such as the ODP portal[4] can be browsed and searched for potentially relevant ODPs, and some tools also provide search functionalities to perform keyword search over ODPs and their annotations [12]. Nevertheless, usually the ODPs found then have to be manually assessed and compared to find the best match for a particular situation. Commonly certain abstract notions, such as events, participation, states and so on, can be represented in several alternative ways, whereas several ODPs exist for such notions. In such cases the ODPs need to be carefully examined, consequences considered, and finally the solution (if any) that best matches the requirements at hand can be selected.

Once an ODP has been selected, the next step is usually to specialise that ODP for the domain problem at hand. In rare cases an ODP may be suitable to use as-is, but this is not commonly the case. As described earlier, there can be several ways to perform the specialisation, e.g., both by reusing the ODP simply as an abstract template and source of inspiration, or by reusing the actual building block, and if reusing the actual building block there are also various strategies for specialisation. Depending on the tool used for modelling, there may be more or less support for this task in the tool. Usually ontology engineering environments support `owl:import` axioms, which can be used to import ODPs as components in your ontology module. However, some tools do not at the time of writing support imports, such as the WebProtégé tool, however, for this tool in particular there is a specific XD plugin being built to partly remedy this situation [12]. In addition to specialisation, covering all the selected requirements usually requires either some composition of several specialised ODPs, or an extension to the ODPs . If a considerable portion of the solution is not supported by any ODP, one may also consider to extract and potentially generalise that solution, and propose it as a new ODP . Taking the time to do so, and sharing it at least with the fellow ontology engineers of the same project may potentially give rise to less problems in the integration of modules later on, since other design pairs may then reuse the same generic solution for their specific modelling problems.

An example ODP specialisation, covering the first CQ of the story from the legal domain in Table 1.4, can be seen in Figure 1.2. The `ParticipantRole` ODP[5] has been used, and specialised by adding classes specifically representing legal roles, legal persons, and legal events, as well as the central class (`LegalParticipant Role`) representing the "n-ary relation" that connects a certain legal person to a certain legal role in a certain event. Assuming that "founding" is a legal event for which this specialised model will now be used, then the CS follows from the range restriction set on the `legalPersonParticipating` property, i.e., all participating objects pointed to by tis property will be inferred to be legal persons. Whether this is exactly the intended behaviour for the CS may have to be confirmed with the customer, but let us assume it is for the sake of this example.

[4] http://ontologydesignpatterns.org/

[5] http://ontologydesignpatterns.org/wiki/Submissions:ParticipantRole

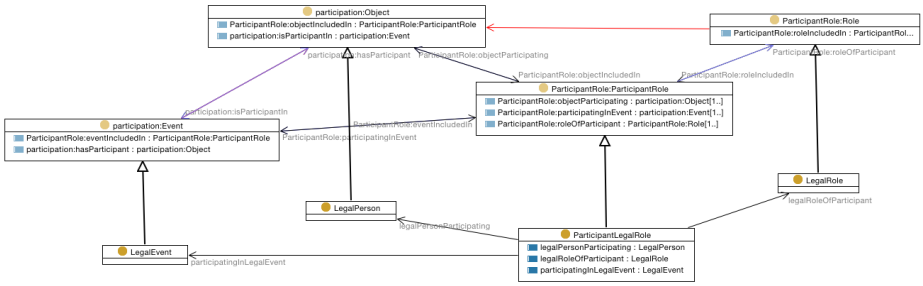

Figure 1.2. An illustration of a specialisation of the ParticipantRole ODP for the legal domain, concerning participation in legal events, such as the founding of a legal entity. Illustration is using the UML-inspired OWL notation of TopBraid Composer. The four classes at the top of the figure constitute classes from the ODP, while the bottom four constitute the specialisation, including the three specialised properties that can be seen connecting these classes.

Once a module has been developed, it should be tested against all its requirements, c.f. [5]. CQs are usually tested through SPARQL queries , i.e., creating a test module, importing the module that has been built and adding test data, then writing and running SPARQL queries corresponding to the CQs over the module. Although this seems quite straight-forward, deciding if the model appropriately solves the CQs may require a bit of deliberation. In some cases it may be possible to write the SPARQL query, but the query turns out to be overly complex, or contains implicit assumptions that could instead be made explicit in the model. Hence, both the query itself and the retrieved results should be analysed in order to decide whether the module has passed the test or not. A test is considered successful if an error is found, i.e., data is missing or different from the expected results. This helps finding mistakes in the model or parts of the specification that were overlooked during the design. In Figure 1.3 the module with some added test data is illustrated, and Figure 1.4 contains a SPARQL query that could be used as a test case for testing the CQ with the test data in the figure. If the original story was written in an exemplary form, then realistic values and test instances may be taken from it.

Reasoning requirements are often as straight-forward to test as CQs, i.e., the ontology engineer simply has to enter some test instances and run an appropriate inference engine over the ontology with the added test data and check the inferred statements against an expected set of statements. The important thing to look for, however, is not only the expected inferences but especially the unexpected ones. It is the unexpected inferences that indicate a problem in the design.

Finally, the contextual statements are the most difficult to check. Preferably they should be accompanied by some explanation of the purpose for which they should be expressed in the ontology, whereas that purpose could be tested. Some CS are also expected to generate inconsistencies under certain conditions, or added inferences, such as inferring `owl:sameAs` axioms holding between individuals. In the latter case they can be tested in a similar manner as the RRs.

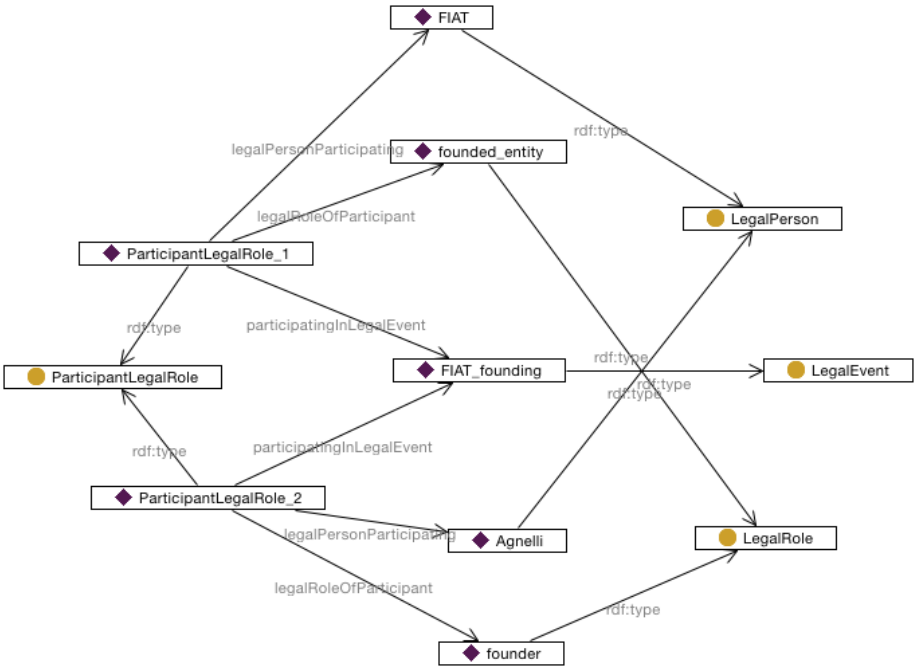

Figure 1.3. An RDF graph representing some test data expressed using the example ODP specialisation in Figure 1.2. The boxes with diamonds represent individuals, while the ones with circles represent classes, i.e., the types of the individuals, in the graphical notation of TopBraid Composer. The data graph could be read as "FIAT (a legal person) participated as the founded entity (legal role) in the FIAT founding event" and "Agnelli (a legal person) participated as a founder (legal role) in the FIAT founding event"

```
SELECT ?person ?founded
WHERE {
?participation1 p:legalPersonParticipating ?person .
?participation1 p:legalRoleOfParticipant :founder .
?participation1 p:participatingInLegalEvent ?event .
?participation2 p:legalPersonParticipating ?founded .
?participation2 p:legalRoleOfParticipant :founded_entity .
?participation2 p:participatingInLegalEvent ?event
}
```

Figure 1.4. An example SPARQL query that could constitute the test case for the first CQ of the legal story in Table 1.4, assuming the model as in Figure 1.2 (represented by prefix `p:`) and the example data as in Figure 1.3. Given the example data, the query should return the instance pair `Agnelli` and `FIAT`, i.e., representing the fact that Agnelli was a co-founder of FIAT.

Whatever test cases and test data is developed, it is important to document and save all of this for later use in the integration phase. Integration of the module into the overall ontology may affect the result of the tests, which should be rerun after integration, c.f. regression testing of software, where previous tests are run again to ensure that modifications or newly integrated software has not affected the already integrated modules. Additionally, after any refactoring needed in the integration phase, the test cases (potentially modified if the module has been refactored) should also be ran again, with the same test data, in order to check that the same results are still achieved.

Once all test cases for all the requirements have been executed, and no further issues have been discovered, the module can be released. The module should of course be appropriately documented, according to the project-specific guidelines set up at project initiation, e.g., by using annotation properties inside the ontology file.

1.2.4. Integration Loop

The integration loop is represented by the righthand side of Figure 1.1. Integration may be performed by a dedicated integration team, or by the design pairs themselves. The advantage of having a dedicated team taking over the released modules is that this team can specialise in integration and refactoring, and also maintain an overview of the overall ontology so far. It also forces the design pairs to document their modules in such a way that they can be immediately understood by others, which may not be the case if they are responsible for integrating their module themselves.

If the integration is performed between a module and a large ontology it may be hard to find all the suitable points of alignment manually. For this task there are numerous ontology alignment systems available, which to some extent have been tested for use within XD [6]. However, so far XD tooling, such as the WebProtégé plugin [12] that will be discussed later, only apply a naive matching approach for proposing alignments, i.e., string matching of names and labels to find potentially equivalent classes. This means that a large part of the alignment task may have to be done manually by the integration team, relying on their expertise.

Additionally, they are responsible for making decisions on refactoring, and implementing such refactoring. For example, if an overlap between two modules are found, even if the design solutions are perfectly compatible one has to make a decision on what strategy to use in order to align the modules. One strategy could be to keep the classes as they are in each module and add equivalent class axioms between them. This has the advantage of keeping the modules self-contained, making it easier to use modules independently for certain reasoning tasks, and facilitates reuse of single modules in later ontology engineering projects. However, the resulting ontology may not be very intuitive to a human, due to the presence of duplicate classes (potentially with different names), which may increase the risk of misuse of the ontology, and which may additionally increase the complexity of querying for data without first materialising all inferences. The situation may also occur when something is missing, in order to connect modules, where the

integration team then would have to perform some modelling in order to create the "glue" to fit the modules together.

What is even more difficult of course is the situation when modules apply completely incompatible designs, where one module really needs complete refactoring in order for both solutions to be incorporated in the overall ontology. There is no universal solution to this problem, each project needs to find the strategy that seems most suitable for their needs. However, any team applying XD needs to be aware of that this is one of the most crucial, and difficult, steps of the methodology, which will need quite a bit of time and effort, and potentially discussions throughout the engineering team, and with the customer as well.

Fortunately, the use of ODPs is a crucial facilitator that makes this approach actually feasible in practice. With a limited ODP catalogue to start from, and potentially an additional emerging set of project-specific ODPs that are shared within the project, design solutions usually tend to converge rather than diverge as the project progresses. The ODPs constitute the common ground on which to build modules, and also constitute a common language for easier communication about solution alternatives. Once the ontology engineers become familiar with the ODP catalogue used, the ODP names are commonly used to signify design alternatives. A typical discussion among ontology engineers could be as follows:

- "I am thinking of using `AgentRole` for this module, I know you did another one involving roles recently, what ODP did you use?"
- "I needed roles to vary over situations so I decided to use `ParticipantRole` instead."
- "Then I'll use that as well, so our modules are compatible."

Without actually talking about the details of the modelling choices they make, the two ontology engineers have agreed on a common design by means of referring to ODPs that they both know. This is the original idea of a pattern language as presented in architecture by C. Alexander [1].

As soon as a new module has been integrated into the overall ontology, it has to be evaluated . At this point, the integration team should reuse all the test cases for all the currently integrated modules and make sure they can be successfully completed also using the integrated ontology, i.e., perform regression testing. In addition, some new test cases should be created, in order to evaluate the effects of the integration, e.g., the added alignments. Finally, the ontology should be applied in its intended usage setting, e.g., within a software system, verifying the functionality that it is supposed to provide according to the stories covered so far. At this stage, it is important to involve the customers again, whether they are the developers of the surrounding software or end users, in order for them to agree with and "accept" the verification results. By involving the customers also any non-functional requirements can be verified, such as correct use of terminology, understandability of the model etc. Only once all such verifications, and potential revisions as a result of that, have been completed successfully a new ontology version is released.

The release process may involve some formal delivery of the ontology, e.g., by making it available online, and it may include to provide a new version IRI for the

ontology (c.f. the versioning guidelines of W3C[6]). How often releases are made may also depend on the project setup. If modules are delivered and integrated very frequently, then it may be more reasonable to provide a new release of the ontology after integrating a set of new modules, rather than a single one.

1.3. Tools and Support Infrastructure

At the time of writing, there is no ontology engineering environment that provides complete support for the whole XD process. However, there are various tools that provide support for certain parts of the process, which are described here. Additionally, some related infrastructure, e.g., in terms of pattern repositories that may also support the process, are also briefly introduced.

There are a number of ontology engineering environments available today, where the most commonly used probably are Protégé Desktop[7], WebProtégé[8], and TopBraid Composer[9]. Additionally, ontologies are also sometimes created through specifying them from an API, e.g., the OWL API[10] or Jena[11], which then generates the OWL file, however, in this section the focus is on tools with graphical user interfaces, and which are not only intended for expert users. One additional ontology engineering environment worth mentioning is the NeOn toolkit[12], which was built by the NeOn project and for a few years after that still maintained by the NeOn foundation. However, as far as the authors are aware the NeOn toolkit is no longer being maintained and the latest version was released in 2011. Despite the fact that the NeOn toolkit actually has a plugin directly supporting XD, it will not be describe here, both due to the uncertain status of the NeOn toolkit itself but mainly the fact that most of the functionality provided by that plugin has now been reimplemented in the XD plugin for WebProtégé.

First of all, it should be mentioned that most of the tasks of the XD methodology can be performed using either a general-purpose ontology engineering environment, such as Protégé or TopBraid Composer, with the addition of some support tools for specific purposes, such as requirements management, debugging, version management etc. This is indeed, in the authors' experience, how most projects have been set up when using XD. A typical setup could consist of the following:

- a **wiki** for project communication, documentation, requirements management, and collection of project specific ODPs,
- a **Git** or **SVN** repository for version control of the modules and the overall ontology,
- an **ontology engineering IDE** like Protégé or TopBraid Composer for the modelling of modules and integration of modules,

[6] https://www.w3.org/TR/owl2-syntax/#Versioning_of_OWL_2_Ontologies
[7] http://protege.stanford.edu/products.php#desktop-protege
[8] http://protege.stanford.edu/products.php#web-protege
[9] http://www.topquadrant.com/tools/modeling-topbraid-composer-standard-edition/
[10] http://owlapi.sourceforge.net/
[11] https://jena.apache.org/
[12] http://neon-toolkit.org/

- a **reasoner** and **SPARQL** engine (potentially the ones shipped with TopBraid Composer) for testing,
- and the online **ODP repository** at ontologydesignpatterns.org as the main source of ODPs.

In order to further support some of the specific tasks of XD, the eXtreme Design for WebProtégé (XDP) extension[13] has been developed [12]. XDP is a reimplementation of many of the functionalities originally provided by the XD Tools plugin for the NeOn Toolkit, but also provides a set of novel new developments[14], including a new search engine for finding suitable ODPs [11], some specific support for the different specialisation strategies (c.f. Section 1.1 and [10]), and an ontology alignment component for providing suggestions of points of integration after ODP specialisation [11].

Additionally, in [6] some existing ontology alignment and debugging tools are tested and analysed, which could further support the integration and testing tasks within XD. Further, the recent Protégé plugin for applying generic structural test cases for OWL ontologies [14], may also be useful for testing purposes. For refactoring of ontologies, the tool suite for pattern-based transformation of ontologies[15] originating from the Patomat project may be used [18]. The Patomat tools allow for expressing transformation rules to transform an ontology, or an ontology module, from the design proposed by one ODP to a structure conforming to a different ODP. Since this requires some manual effort in terms of writing transformation rules it may not be suitable for a one-time effort, but for a large ontology engineering project where multiple modules need to be transformed, this may be a very useful tool to apply. It would also be possible to set up your own transformation services based on the OPPL language[13], or use the OPPL Protégé plugin[16] for specifying transformations and templates directly.

Finally, in order to apply XD the project has to have access to appropriate ODPs. There are several ODP repositories available for ODPs of different types, but for Content ODPs the two most frequently used repositories are the ODP portal's list of submitted Content ODPs[17] and the Manchester ODP repository[18]. At the time of writing the ODP portal contains a list of 114 submitted Content ODPs. However, it should be noted that quality assurance of ODPs is not standardised, and the different portals may apply varying criteria for publishing an ODP. One quality assurance opportunity is to submit the ODP to the WOP workshop series[19], where the ODP would undergo a peer review process. The ODP portal additionally provides various other ODP types, in addition to Content ODPs, as does the Manchester repository.

[13] Online demo: http://wp.xd-protege.com, video walkthrough: https://youtu.be/ZRH6vGXocqU, code: https://github.com/hammar/webprotege

[14] Note that as at the time of writing WebProtégé does not support `owl:imports`, import and specialisation has been implemented through duplicating ODP entities in the target ontology.

[15] http://owl.vse.cz:8080/patomat/

[16] https://sourceforge.net/projects/oppl2/files/

[17] http://ontologydesignpatterns.org/wiki/Submissions:ContentOPs

[18] http://www.gong.manchester.ac.uk/odp/html/

[19] http://ontologydesignpatterns.org/wiki/WOP:Main

1.4. eXtreme Design Use Cases and Lessons Learned

In this section a brief description of some of the lessons learned from using ODPs, and the XD methodology for ontology engineering, are presented. First, a few projects and use cases where XD was applied are mentioned, both providing a brief history of some of the first large-scale applications of XD as well as providing an example of the variation of contexts and kinds of ontologies XD has been applied to build. Next, a number of experiments and user studies that have been carried out are summarised, which have analysed and pointed at the benefits of using ODPs and XD, and then a set of points of adaptation are described, where XD has sometimes been adapted to practical constraints with a successful result. Finally, some experiences in using ODPs and XD as tools for teaching and learning ontology engineering are presented.

1.4.1. Example Use Cases

XD has been frequently applied in various contexts over the past decade. In this section this usage is exemplified by briefly mentioning a few of the projects where XD has been applied in various forms. The intention is neither to provide a full listing of all projects where it has been applied, nor to describe each case in detail, but merely to exemplify the variety of ontologies and project contexts where the methodology has been used. Some of these projects were also used for studying aspects of XD more in-depth, as described in the next section.

XD was originally proposed in the context of the NeOn project, and ODPs and an initial version of XD was then applied to construct ontologies for one of the project use cases in the fishery domain. One of the project use case partners was the Food and Agriculture Organisation (FAO) of the United Nations, who were building a system for fish stock monitoring all over the world. The ontologies of this use case were constructed according to an early version of the XD methodology, still including the focus on requirements, modularisation and ODPs. The resulting network of fishery ontologies, i.e., ontology modules, was evaluated and published by the FAO[20], and act mainly as standardised vocabularies for information sharing and integration. This project also resulted in a set of "fishery ODPs" that are listed in the ODP portal[21].

The XD methodology was also applied in a use case of the IKS project[22], which was previously mentioned. The overall project focused on introducing semantic technologies into the domain of Content Management Systems (CMS), while the particular use case where XD and ODPs were applied focused on an ambient intelligence application, i.e., an "intelligent bathroom" environment that should proactively serve the user with context-dependent information related to current user needs and preferences. A physical bathroom environment was set up in a lab environment, where also the software using the ontologies was tested with actual users. In contrast to the NeOn use case, the ontologies created here were focused on reasoning tasks, in order to support the system "intelligence",

[20] http://aims.fao.org/network-fisheries-ontologies
[21] http://ontologydesignpatterns.org/wiki/Community:Fishery
[22] http://www.iks-project.eu/

rather than merely acting as vocabularies for data. Hence, it was mainly in the context of this project that the different types of requirements (CQs, CS and RR) were elaborated, and different testing methods for them were introduced. A set of ontology modules, representing the ontology for supporting a few larger user stories that were tested in the bathroom environment are still available for download[23].

More recently, XD is being applied in a number of ongoing projects ranging from the creation of criminal profiling ontologies for criminal intelligence analysis[24], via ontologies for automated test case generation in software engineering (project OSTAG), to the development of Decision Making ODP prototypes in the context of a W3C incubator group[25], and various Linked Data projects. In several of these projects, XD was slightly adapted, e.g., in terms of personnel, roles, and project setup, while still maintaining the core ideas of requirement and ODP focus, as described further in Section 1.4.3.

1.4.2. Empirical Evaluations of ODP Usage and XD

In order to establish ODP usage and the XD methodology as a viable support to ontology engineers, a series of initial experiments and observational studies were conducted over the course of several years. The results of these studies have been reported in [3, 4, 9].

Conclusions include that ODP usage indeed improved several quality aspects of the resulting ontologies as expected, e.g., overall a lower error rate, higher understandability due to increased number of comments etc., although there are also a few pitfalls concerning misuse of ODPs, as noted in [9]. While it was not possible to quantitively see any difference in the modelling time spent to solve a certain modelling problem, nevertheless, as noted in [9] the ontology engineers themselves feel that they solve the tasks faster. One potential explanation for this is that the time it takes to find, understand, and select an appropriate ODP without any specific tool support, and without prior experience with the ODPs in question, roughly corresponds to the time saved in the actual modelling phase when reusing the building block of that ODP. However, this is still an unconfirmed hypothesis.

Concerning XD, observations and questionnaires with ontology engineers using the methodology has led to the conclusion that most of them found the methodology very useful. An interesting note is that many subjects expressed that they were already working in a similar manner, e.g., in a divide-and-conquer process when addressing a modelling problem, but still their results improved considerably when XD was explicitly introduced. In this case a potential explanation could be the formalisation of the testing process, i.e., it could be the case that people think they test all their requirements properly, but in reality they may not actually be doing that if they are not following a structure approach for requirements management and testing. Yet again, this is an unconfirmed hypothesis.

[23] http://www.ontologydesignpatterns.org/iks/ami/2011/02/
[24] Project VALCRI - http://valcri.org/
[25] https://www.w3.org/2005/Incubator/decision/wiki/Draft_Final_Report

1.4.3. Adaptations of the Methodology

So far the XD methodology has been described as it was originally proposed, only with the addition of some more details and a few clarifications compared to how it was described in [15, 16]. However, during the past 10 years a lot of experiences have been collected while using the methodology, some of which are described in section 1.4.2, but some that has also made us notice a few common variations of the methodology. Such variations are often due to external factors or project specific preferences that prevent from applying XD exactly in its original form.

One of the first observations made was that most ontology engineering projects are still quite small. There are of course a few large multi-decade efforts, like the Gene Ontology, Cyc, etc., and a number of medium sized ones. However, the large bulk of ontology engineering projects are still run by just a few people over a time period of a few months to a few years, typically as a sub-project of a larger effort, e.g., a software project, or a linked data publishing project. In such a setting an XD configuration with several design pairs working in parallel and a dedicated integration team may not be realistic. It may even be the case that an ontology is created by one individual, e.g., in the context of a larger software project. What adaptations need to be made to XD to be applied in such a setting?

In the projects matching this description that have been observed, XD has very successfully been applied also there. The requirement of pair design may have to be relaxed, and one or two ontology engineers work separately on creating the modules. The advantages of such a setting is that everyone involved has a good overview of the overall ontology at all times, which makes integration easier since problems are anticipated already when designing the modules. However, the disadvantage may be that it is easy to slip back into a mindset where you take on too large challenges at a time, trying to design the overall ontology at once, instead of focusing on one single modelling issue at each point in time. However, also in this case the use of ODPs helps the ontology engineer to keep focused on one single problem at a time, since ODPs are inherently small and modular. Additionally, running XD in "single developer-mode" may have consequences on documentation and testing, since there is no external scrutiny of documentation nor of test setup an results during the development. This makes it even more important to in this setting have frequent contacts with customers and potentially add external reviews to the evaluation phase before the release of an ontology version.

Another very common situation is to have distributed engineering teams. Nowadays many teams are distributed both geographically, and as a consequence may also be highly distributed over time zones. So even if the team is larger than one or two ontology engineers, it may not be feasible to apply pair design as it was originally intended, i.e., two ontology engineers in front of one screen. Two main ways to deal with this issue have been observed; either pair design is completely left out of the methodology, and each ontology engineer works individually, or a compromise is made and engineers still work in pairs but using a more asynchronous method of collaboration, such as sending solution suggestions back and forth or collaborating on the same WebProtégé project but editing it at different points in time. Although the effects of this have not been evaluated in detail, experiences show that despite the shortcomings of the asynchronous

communication, it is still worth the effort in terms of improved quality of the resulting model, compared to an ontology engineer working on his or her own.

A further observation is that originally XD was proposed with only two roles of the project participants; customer and ontology engineer. However, in reality, there can be a variety of specialisations of these roles, depending on the characteristics of the project but also on the skills and competencies of the project staff. On the customer side, there is usually at least two types of people; domain experts and software developers. Since ontologies are rarely used in isolation, there is is usually someone (other than the ontology engineers themselves) building, or setting up, the software that is going to use the ontology. Whether it is a simple storage solution to serve a SPARQL endpoint to the web, or whether it is a complex AI system to use the ontology for various reasoning tasks, these software developers need to get at least some understanding of how the ontology is being built, what design solutions it contains, and what functionality it supports. Additionally they will have requirements of the ontology, such as what namespaces it uses, how versions are to be handled, typical queries etc. Communication with this kind of "customer" is however very different from the communication that needs to take place with a domain expert, who may not have any technical knowledge at all. Recognising this difference, and involving different kinds of customers in different ways throughout the XD workflow is important, but it is currently not specified in detail in the methodology itself.

Similarly for the ontology engineers of the project, some may be more suited for performing testing, while other may be suited for designing modules or integrating them. The experiences of the authors indicate that the integration loop is the most challenging part of the methodology, where you have the least support from ODPs. Therefore it may be better to assign the most experienced ontology engineers to the integration team, while more inexperienced engineers can do quite well on module development, with the help of a proper ODP catalogue.

Finally, despite the fact that XD is heavily focused on reuse, it does not really detail how to manage other reuse than the selection and application of ODPs. As mentioned earlier, an ontology is rarely constructed completely from scratch. Concerning this, mainly projects that reuse existing ontologies have been observed, e.g., W3C standard for instance, where these have simply been used as a starting point of the overall ontology, and complementing modules have then been created and integrated with that starting point. However, this is also a point of future work for improving XD, to develop more detailed guidelines for integrating existing resources.

1.4.4. ODPs and XD as a Learning Tool

In addition to using ODPs and XD in actual ontology engineering projects, there is also quite an extensive track record of using them for teaching purposes. Experiences of the authors indicate that ODPs are highly suitable as a way of introducing various trade-offs and design choices and their consequences to inexperienced ontology engineers. Participants need some basic knowledge of modelling in general, and of the languages involved, but ODPs then have the potential of creating a fast-track from basic knowledge of OWL to a reasonable set of modelling skills,

in order to take part in realistic large-scale ontology engineering projects.

ODPs and XD have been taught in various settings[26] over the past decade. While it is commonly necessary to start by introducing the basics, and the basic idea behind ODPs, practical modelling using ODPs for modular ontology engineering is then introduced very quickly, and already after 1-2 days of training the participants are usually ready to take on a small "ontology engineering project". The latter is usually conducted as an XD project, where the participants are divided into pairs and work according to the XD methodology on a shared development project. This both practices their collaboration skills, but also allows the participants to uncover and analyse shortcomings in their own ontology designs. For instance, when students are to integrate modules built by other students, they often realise the importance of documentation, since it is not as easy as they expected to understand someone else's model. Thereby the authors are convinced that XD is a very suitable methodology to apply when teaching ontology engineering.

1.5. Summary and Future Work

This chapter presented how ODPs can be used in the ontology engineering process, and in particular the XD methodology for ODP-based ontology engineering was introduced. XD is an agile and iterative ontology engineering methodology, that incrementally builds an ontology as a composition of a set of ODP-based ontology modules. Both the use of ODPs as such, and the XD methodology, have been studied in a number of settings, allowing us to confirm a set of positive effects, such as reduced error rate in the resulting ontology, improved documentation and consequently increased understandability of the resulting ontology, as well as a number of subjectively perceived benefits, such as easier and faster development of ontology modules. Applying the XD methodology also allows to have a working increment early in the project, which reduces the risk of misinterpreted requirements, as the (partial) solution can be confirmed and validated in its intended usage scenario very early on. Due to this, XD is also particularly suitable for rapid prototyping of ontologies.

As experiences from working with XD during a number of years show, there are a number of adaptations of XD that have been useful in the realistic settings where it has been applied. In particular, relaxing the criteria of pair design, and using mainly online collaboration instead of face-to-face interaction has worked well in settings where practical constraints have made this the most feasible solution. To further detail the roles of participants, e.g., so that more experienced ontology engineers are assigned specific tasks, such as managing the integration loop, have also proved to be beneficial, rather than to treat all design pairs equal in the project. However, how to more precisely specify the suitable competencies for the various roles and tasks is still future work.

Additional future work consists of providing more detailed guidelines for reusing existing ontologies, and even non-ontological resources, in the XD pro-

[26] For an (incomplete) list of courses and their content, see: http://ontologydesignpatterns.org/wiki/Training:Main

cess, as well as improving the tool support for the methodology. Despite the fact that there exist tools for most tasks in the XD process, only a few of them are integrated into an ontology engineering IDE, and there is still a lack of specific support for the XD workflow in these IDEs. Additionally, since XD relies heavily on the use of ODPs, the result of an XD project may only be as good as the ODPs available and used. In a larger, more long-term ontology engineering project, it is reasonable to set up your own emerging ODP catalogue, while in smaller projects one usually relies entirely on ODPs available in online catalogues. For this reason, the quality and coverage of such catalogues is a crucial point of improvement in order to apply XD more successfully in smaller projects, especially if there is a lack of ontology engineering expertise in the development team.

Bibliography

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press, 1977.

[2] V. Basili, G. Caldiera, and D. Rombach. The experience factory. In J. Marciniak, editor, *Encyclopedia of Software Engineering.*, pages 469–476. Wiley, New York, 1994.

[3] E. Blomqvist, A. Gangemi, and V. Presutti. Experiments on Pattern-Based Ontology Design. In *Proceedings of the 5th International Conference on Knowledge Capture (K-CAP 2009), September 1-4, 2009, Redondo Beach, California, USA*, pages 41–48. ACM, 2009.

[4] E. Blomqvist, V. Presutti, E. Daga, and A. Gangemi. Experimenting with eXtreme Design. In *Proceedings of EKAW 2010 - Knowledge Engineering and Knowledge Management by the Masses, Lisbon, October 11-15*, LNCS, pages 120–134. Springer, 2010.

[5] E. Blomqvist, A. Seil Sepour, and V. Presutti. Ontology testing - methodology and tool. In *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012, Galway City, Ireland, October 8-12, 2012*, volume 7603 of *LNCS*, pages 216–226. Springer, 2012.

[6] Z. Dragisic, P. Lambrix, and E. Blomqvist. Integrating ontology debugging and matching into the extreme design methodology. In *Proceedings of the 6th Workshop on Ontology and Semantic Web Patterns (WOP 2015) co-located with the 14th International Semantic Web Conference (ISWC 2015) Bethlehem, Pensylvania, USA, October 11, 2015.*, volume 1461. CEUR Workshop proceedings, 2015.

[7] A. Gangemi and V. Presutti. Ontology Design Patterns. In *Handbook on Ontologies, 2nd Ed.*, Int Handbooks on Information Systems. Springer, 2009.

[8] M. Gruninger and M. S. Fox. The role of competency questions in enterprise engineering. In *Proceedings of the IFIP WG5.7 Workshop on Benchmarking - Theory and Practice*, 1994.

[9] K. Hammar. Ontology design patterns in use - lessons learnt from an ontology engineering case. In *WOP 2012: Proceedings of the 3rd Workshop on Ontology Patterns, in conjunction with the 11th International Semantic*

Web Conference (ISWC) 2012, CEUR Workshop Proceedings, volume 929. CEUR Workshop proceedings, 2012.

[10] K. Hammar. Ontology design pattern property specialisation strategies. In *Knowledge Engineering and Knowledge Management - 19th International Conference, EKAW 2014, Linköping, Sweden, November 24-28, 2014. Proceedings*, volume 8876 of *LNCS*, pages 165–180. Springer, 2014.

[11] K. Hammar. Ontology design patterns: Improving findability and composition. In *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, volume 8798 of *LNCS*, pages 3–13. Springer, 2014.

[12] K. Hammar. Ontology Design Patterns in WebProtégé. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015)*, volume 1486. CEUR Workshop proceedings, 2015.

[13] L. Iannone, A. Rector, and R. Stevens. Embedding knowledge patterns into owl. In *The Semantic Web: Research and Applications - 6th European Semantic Web Conference, ESWC 2009 Heraklion, Crete, Greece, May 31–June 4, 2009 Proceedings*, volume 5554 of *LNCS*. Springer, 2009.

[14] C. M. Keet and A. Lawrynowicz. Test-driven development of ontologies. In *The Semantic Web. Latest Advances and New Domains 13th European Semantic Web Conference, ESWC 2016, Anissaras, Crete, Greece, May 29 – June 2, 2016. Proceedings*. Springer, 2016.

[15] V. Presutti, E. Bomqvist, E. Daga, and A. Gangemi. Pattern-based ontology design. In *Ontology Engineering in a Networked World*, pages 35–64. Springer, 2012.

[16] V. Presutti, E. Daga, A. Gangemi, and E. Blomqvist. eXtreme Design with Content Ontology Design Patterns. In *Proceedings of the Workshop on Ontology Patterns (WOP 2009), collocated with the 8th International Semantic Web Conference (ISWC-2009), Washington D.C., USA, 25 October, 2009*, volume 516. CEUR Workshop proceedings, 2009.

[17] J. Shore and S. Warden. *The art of agile development*. O'Reilly, 2007.

[18] O. Zamazal and V. Svatek. Patomat – versatile framework for pattern-based ontology transformation. *Computing and Informatics [online]*, 4(2):305–336, 2015.