Improving the Quality of Software Design through Pattern Ontology

by

Marc Guy Boyer

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2011

Thesis advisor

Dr. Vojislav Mišić

Author

Marc Guy Boyer

Improving the Quality of Software Design

through Pattern Ontology

Abstract

Software engineers use design patterns to refactor software models for quality. This displaces domain patterns and makes software hard to maintain. Detecting design patterns directly in requirements can circumvent this problem. To facilitate the analogical transfer of patterns from problem domain to solution model however we must describe patterns in ontological rather than in technical terms. In a first study novice designers used both pattern cases and a pattern ontology to detect design ideas and patterns in requirements. Errors in detection accuracy led to the revision of the pattern ontology and a second study into its pattern-discriminating power. Study results demonstrate that pattern ontology is superior to pattern cases in assisting novice software engineers in identifying patterns in the problem domain.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the support of the following persons and institutions.

First and foremost, my thanks to Dr. Vojislav Mišić whose wit, good sense, and deep insights into the art and science of software engineering were a constant source of help and encouragement to me.

My thanks to Dr. David Scuse, Dr. Pourang Irani, and Dr. Jason Leboe for accepting to sit on the thesis examination committee and to Dr. John Bate for accepting to be the chair for the thesis defense.

My thanks also to Dr. Vojislav Mišić, the Department of Computer Science, the Faculty of Science, the Government of Manitoba, the National Sciences and Engineering Research Council of Canada, and the National Aboriginal Achievement Foundation for their financial support.

This thesis is dedicated to Christopher Alexander, the father of the modern pattern movement and the inspiration behind the present work on using pattern ontology to improve software design.

Christopher Alexander, *A Timeless Way of Building*, 1979

Suppose now, that for a given act of building, you have a pattern language, and that the patterns in this language are arranged in proper sequence. To make the design, you take the patterns one by one, and use each one to differentiate the product of the previous patterns.

§

Again, just as before, the process is sequential. Only now the patterns operate not on a mental image, but on the building itself, as it is being built. Each pattern defines an operation, which helps to differentiate, and to complete, the building as it grows: and when the last patterns are introduced into the growing fabric, the building is complete.

§

Again, the patterns operate upon the whole: they are not parts, which can be added—but relationships, which get imposed upon the previous ones, in order to make more detail, more structure, and more substance—so the substance of the building emerges gradually, but always as a whole, at each stage of its growth.

Chapter 1

Introduction

In software engineering, practitioners make every effort to design software that meets both the functional and quality requirements of end-users. Function, because software must solve problems in the real world; quality, because software that does not run properly on a computer is not of much use to anyone.

Designers often integrate domain patterns into the solution model to make it reflect real-world entities and relationships. This facilitates the solving of end-user problems within the context of the application domain as it evolves over time. Unfortunately, software not designed with technology constraints in mind often runs poorly on a computer. This may force designers to use design patterns to restructure the model to improve the software's run-time performance. An unwanted side-effect of this refactoring is the reduction of the model's ties to real-world things, as technical-quality patterns taken from the realm of technology begin to displace the domain patterns initially collected from the problem domain.

This forces software engineers into making a difficult choice. Do they keep the domain patterns in the model for the sake of software maintainability even if this means reduced software performance? Or do they restructure the model to fit design patterns that increase software per-

formance but at the expense of its maintainability? For example, should designers model even the smallest parts of a skyscraper (e.g., bolts) as objects, knowing full well that no computer could possibly load such a detailed object model? On the other hand, should they really be adding layers of isolating, adapting, and sharing mechanisms foreign to skyscrapers into the model of a skyscraper just to get it to work on a computer?

At the present time, software engineers fall into three camps on this question. In the first camp are those that insist that we link model elements directly to the properties of real-world things, whatever the cost. A second camp argues that models should be structured for quality to run well on computers, come what may. A third camp seeks to reconcile these two positions; their goal is to use pattern-assisted design techniques to integrate both domain and design patterns into the software model, and without prejudice to either one.

This thesis belongs to the third camp in that it examines how design patterns may be used to increase the run-time performance qualities of the software model (in this thesis, the "quality" of the model) without compromising its faithful representation of the things in the problem domain (in this thesis, the "accuracy" of the model). It is well-known that the design patterns found in the pattern catalogs offer reusable structural solutions to known design-quality problems. Our goal is to show that we can also use these patterns to accurately detect ontological regularities (in this thesis, the "domain patterns") in the domain and so to transfer them into the technical realm.

In our opinion, designers should use design patterns not only to refactor substandard models but to build both technical quality and domain accuracy into their initial software models as well. To reach this goal however requires that we change how we describe design patterns so that they become useful in initial design-phase activities as well.

But why so?

From the findings of the computer and cognition sciences we learn that accurate pattern detection depends primarily on two things. In the first place, each pattern must be distinct from other

patterns so that we can detect it specifically in instances. Secondly, the terms used to define a pattern must match the terms used to describe real-world things for the analogical transfer of knowledge from the source (in our case the domain pattern described in the requirements) to the target (in our case the design pattern) to occur.

Unfortunately, existing design-pattern descriptions fail on both counts. First, existing descriptions use technical terms to describe the patterns. These technical terms are unlike the domain terms used to describe things. This makes it difficult for designers to map the design patterns to the regularities detected in things. Second, the design patterns overlap once the abstraction powers used in the design process eliminate the technical terms that differentiate them. Choosing the right pattern to model the domain knowledge becomes difficult when more than one design pattern can match the functional regularities being detected there.

From these observations flow our first prediction, that designers using design patterns as currently described are likely to detect specific patterns in requirements with low accuracy.

As a solution to this problem we propose to describe design patterns differently. We first identify a set of basic structural and behavioral pattern properties present in existing design patterns. We name these key design ideas in ontological terms that cross application domains. We then use these design ideas to differentiate the design patterns in a hierarchical ontology of patterns, that is, in a pattern ontology.

We expect pattern ontology to facilitate the use of design patterns in the design phase of software engineering for two reasons. First, describing design patterns in ontological terms should facilitate the mind's analogical transfer of pattern knowledge from the problem domain to the technical realm. Second, differentiating design patterns in a hierarchical tree using very specific design ideas should facilitate the mind's detection of these ideas and of the design patterns attached to them in real-world things.

From this flows our second prediction, that designers are likely to detect specific pat-

terns in requirements more accurately using pattern ontology than using traditional design-pattern descriptions.

In this thesis, then, we first critique the approaches used to mimic domain knowledge in models, to structure models for quality, and to use patterns in support of the design- and domain-pattern integration process. We study the findings of the computer and cognition sciences to learn how to detect specific patterns in things and how to transfer these patterns to a different knowledge domain with accuracy. Based on these findings we present a theoretical model that explains why pattern ontology should result in a more accurate detection of patterns in requirements than traditional design-pattern descriptions. We present an initial pattern ontology and the method used to construct it as proof of the feasibility of this new approach to design-pattern definition.

We present two studies with novice designers that validate the hypotheses above. In the first study we investigate whether pattern ontology permits a more accurate detection of design ideas and of patterns in requirements than traditional design-pattern descriptions. In a second study we investigate whether calculated changes in the design ideas used to define the pattern ontology can improve pattern-detection outcomes. Study results show that using traditional design-pattern descriptions does indeed lead to low accuracy in pattern detection, and that using pattern ontology has the opposite effect.

The thesis makes several important contributions to ongoing research in the field of software engineering:

1. A theoretical model that sheds light on the cognition mechanisms underpinning accurate pattern detection and pattern-knowledge transfer in software design.

2. The introduction of ontological analysis to design-pattern definition.

3. The identification of several key design ideas that can efficiently describe and differentiate the design patterns currently published in the pattern catalogs.

4. Experimental studies that demonstrate that using pattern ontology results in a more accurate detection of design ideas and of patterns in requirements than using existing design-pattern descriptions.

The rest of the thesis is divided into six sections as follows. Chapter 2, Related Works, reviews software-engineering, human-cognition, and pattern-recognition research done in the computer and cognition sciences. Chapter 3, Theoretical Model, uses research findings to explain why designers should detect patterns in requirements more accurately using pattern ontology than using traditional design-pattern descriptions. Chapter 4, Problem, Solution, and Hypotheses, briefly outlines the problem, solution, and hypotheses of the thesis. Chapter 5, Experimental Method, describes the studies undertaken to test the hypotheses. Chapter 6, Findings, analyzes study results in the light of the theoretical model and of the hypotheses. Finally, Chapter 7, Conclusion, reviews the results and contributions of the thesis and points to future research that is possible in the area of pattern ontology.

Chapter 2

Related Works

When designing software, software engineers have two key objectives in mind. The software must help end-users solve a problem that they are having in the application domain. The software must run efficiently on computer technology. Engineers must meet both objectives if their software is to be both useful and useable to end-users.

If the designer structures the software according to how entities exist in the application domain, then software maintenance becomes easier. This is true because when entity interactions change in the application domain so can the software structures that mimic them. This is the key benefit of designing software for domain accuracy (section 2.1, Designing for Model Accuracy). Another approach is to structure the software using design patterns so that it runs more effectively on computer technology. In this case, it is the non-functional structures integrated into the model that will predominate (section 2.2, Designing for Model Quality).

Unfortunately, the ontological patterns of the real world and the technical patterns of the computer world are often dissimilar and may conflict. To ask the software engineer to choose between model accuracy and model quality is unfair however because both are required in the software. A third option then is for the engineer to use pattern-assisted design techniques to integrate

both design patterns and domain patterns into the software model at the same time (section 2.3, Pattern-Assisted Design).

This last approach has merit because the software will then model the structural elements both of the application domain and of the technical realm. A prerequisite of this approach however are design patterns described in a way that the designer can accurately detect them in software requirements. Existing design-pattern descriptions do not fit the bill, if what both novice and experienced designers say about them is true (section 2.4, Design Patterns). Hence we need to describe design patterns differently if we are to realize the promise of pattern-assisted software design.

The maintainability and structural quality of a software model will depend then primarily on the accuracy of the designer's identification of design patterns in the requirements: the closer the match between the domain and design patterns, the better the resulting software will be. We look to the cognition sciences (section 2.5, Pattern Transfer in Human Cognition) and to the computer sciences (section 2.6, Pattern Recognition in Machines) for insight into the mechanisms that underlie accurate pattern recognition in humans and in machines.

We will use this research in the next chapter (chapter 3, Theoretical Model) to predict the accuracy of detecting patterns in requirements using traditional design-pattern cases or the proposed pattern ontology. A description of thesis hypotheses and of the experimental method used to validate them will then follow.

2.1 Designing for Model Accuracy

One of the goals of software design is to define a software solution for an end-user problem in an application domain. This solution must model domain knowledge accurately if end-users are to use it to solve real problems. Matching model structures to domain patterns can also make the software easier to maintain because as domain realities change so can the solution model that

represents them.

Domain analysis. Designers may use domain analysis, for instance, to define problems and solutions in the context of the application domain (Andrade et al., 2004). They may use a problem-sensitive modeling language (Andrade et al., 2006), a formal language (Evermann and Wand, 2005), or even domain-analysis tools (Lisboa et al., 2010) to assist them in their work. As a general rule domain models are useful only to the extent that they use the language and duplicate the patterns of the problem domain.

Expert systems. Designers may use expert systems instead to model expert knowledge of problems and solutions in a domain (Liao, 2005). End-users can then navigate through these reasoning systems to solutions already known to work for problems in the targeted domain.

Ontology. Ontology can also capture the real-world relationships of the entities that exist in the application domain (Pinto and Martins, 2004). An additional benefit of these structures is that they give designers the key semantic terms to use to describe how the entities are bound together or differentiated in their original real-world context.

Simulation models. Business-process model elements (Dijkman et al., 2008) or Petri nets (Bernardeschi et al., 2001) can also represent domain knowledge. Designers can use these more dynamic models to observe the changing behavior of represented domain entities over time.

Contracts. Designers may also specify desired domain outcomes for the software using `shall` statements (Daniels and Bahill, 2004) or more formal contracts (Cheon et al., 2005). Designers can use these contracts to confirm that software outputs do in fact meet the domain requirements of end-users, or even to drive the software-development process itself (Beck, 2002).

Prototyping. Another option is to use prototyping. In this case domain elements are shown on a visual interface that end-users are to interact with at some point (Arnowitz et al., 2007). This can help designers target the right domain knowledge to model for the end-users who are to eventually use the software.

Use cases. Designers may describe the interactions of end-users and software with use cases (Eriksson et al., 2008) or scenarios (Sutcliffe, 2003) instead. Making explicit the specific domain knowledge that end-users expect to work with can increase the likelihood that the software will be useful in that application domain.

Knowledge decomposition. Alternatively, designers may use knowledge-decomposition techniques like problem frames (Jackson, 2005; Seater et al., 2007), requirements engineering (Li, 2008), or requirements specification (Redondo et al., 2005) to specify more precisely in the initial requirements the domain knowledge describing the problem and its real-world context.

Object-oriented design. Object-oriented design is also an option. Here designers link the attributes, operations, and associations of real-world things to the classes that are to represent them (Schach, 2002). Domain-driven design is a good example of this approach (Evans, 2003). Using an object-modeling language can also help designers correct model associations that are logically incorrect and so cannot possibly exist in the real world (Barbier and Henderson-Sellers, 2000).

Agile methods. Finally, agile methods transfer domain knowledge directly into software code without the use of an intermediary model (Martin, 2003). In this case, it is the software artifact itself that represents and encapsulates all that is known about the entities and the domain.

These approaches all produce a solution model that accurately reflects the realities in the application domain. This tight coupling of model and domain patterns increases model maintainability but it also makes refactoring the model along technical lines more problematic. Domain-centric approaches may force the designer to sacrifice model quality for model accuracy. A different approach to software design is needed if we are to avoid this dilemma altogether.

2.2 Designing for Model Quality

An equally important goal of software design is to structure the solution model so that the software runs well on a computer. Software must not only solve problems in a domain. It must also run reliably and efficiently on a computer if end-users are to get any benefit from it.

Formal methods. Designers can improve model quality by following formal software-development methods (Bjørner, 2000; Miller et al., 2006). Most formal methods are adaptations of the well-known `Waterfall` method which defines the activities necessary to the systematic elaboration of software.

Grammars. Restricting the words or forms permitted in requirement specifications can also help. Grammar rules like Z-notation (Spivey, 1992), set theory and Venn diagrams (Al-Karaghouli et al., 2000), F-logic (Yang and Kifer, 2006), or natural-language rules (Georgiades et al., 2005) can help designers define more exactly and up-front what end-users require.

Model diagrams. Modeling solutions in diagrams constrained by composition rules—for example, by the Unified Modeling Language—may also increase model correctness (Burton-Jones and Meso, 2006) and quality (Nugroho and Chaudron, 2008). Helping designers avoid elementary logic errors when doing design is the principal benefit of this technique.

Validation tests. Designers can also use validation tests to verify that a model mirrors the problem domain (Nelson and Monarchi, 2007). Feedback loops do not say how to change a model to increase its quality but they are still useful to designers as model-error detection mechanisms.

Design-concept analysis. Designers must also have a correct understanding of design concepts (for instance, what is meant by the term `cohesion`) before using them to evaluate and improve the quality of a model. Uncertainty as to what a quality looks like is unlikely to result in a model that contains it (Mišić, 2000). Some even call for precise, axiom-based definitions of design concepts so that designers can detect them more consistently (Morasca, 2008).

Quality attributes. Techniques exist to evaluate the quality of model architecture as well. Designers can use code tactics (Bachmann et al., 2002), process-analysis (Eguiluz and Barbacci, 2003), and architecture review (Bass et al., 2003) to identify software-model parts that may need to change to support a targeted quality.

Pattern catalogs. Following the lead of the Gang of Four (Gamma et al., 1995), practitioners (Fowler, 1997, 2003) and pattern communities (Henninger and Corrêa, 2007) have defined scores of pattern cases that address specific design-quality failings. Designers should use existing pattern catalogs if at all possible (Cutumisu et al., 2006). Reusing industry-tested patterns is preferred to the invention of new design patterns whose quality characteristics are still unknown.

Refactoring to patterns. Finally, designers can follow expert directions on how to incrementally restructure model parts to fit a specific design pattern (Kerievsky, 2005). Following the steps that other designers have used to integrate a design pattern into a model increases the likelihood that the resulting model will contain it.

Methodology, modeling rules, quality-analysis techniques, and design patterns can all help the designer to improve the quality of a solution model. On the other hand, imposing quality rules on a model—and in the worst-case scenario, even refactoring it—may also displace the domain patterns already implanted there. Quality-centric approaches may force the designer to sacrifice model accuracy for model quality. A better approach might be to structure the solution model using only the design patterns that match the domain patterns detected in the requirements. This would remove the need to sacrifice model accuracy for model quality, and vice-versa.

2.3 Pattern-Assisted Design

Perceiving quality and accuracy as conflicting goals in software design may also result in software that is increasingly difficult to maintain over time. If a designer continually changes model

structures from domain patterns to design patterns—then back again, depending on the needs of the moment—it will not be long before previously-discernible patterns (domain or technical) are lost in the mix. Redesign, a.k.a. refactoring, is a reactive approach to integrating domain and design patterns, and it is certainly not the optimal way to design software for the long term.

Providing software engineers with design patterns that match domain regularities can neatly sidestep the quality-accuracy dichotomy completely. If the design patterns picked to structure the model and the patterns detected in the problem domain are a good fit, the initial solution model will adapt well to changes in the domain and will have desirable performance qualities too. Such is the promise of the pattern-assisted approach to software design.

Aspects. For instance, designers may use system aspects to detect specific system elements (for example, `user roles`) in the problem domain (García-Duque et al., 2006). The benefit of scanning the requirements for one design property at a time is that it permits a more exact detection of the specific property in question.

Pattern discovery. Designers can use a similar strategy to identify complete patterns in requirements (Muller et al., 2007; Hsueh et al., 2009). Designers can use patterns to build quality into the solution model at design time (Wania and Atwood, 2009). This means that they need not limit the use of patterns to the refactoring phase of software development only.

Decision-support. Another option is to use a knowledge base of design rules to support designer modeling decisions. Such a system can prompt designers with timely advice on how best to integrate quality patterns into a solution model under construction (Antony et al., 2005).

Expert system. Designers may also find useful expert systems that map problem forces to the patterns that resolve them. These systems can guide designers in their choice of the best pattern to use to fix specific quality failings detected in the model of the domain (Moynihan et al., 2006).

Ontology-driven. Finally, designers can use an existing ontology of a domain to guide their creation of new models in this same domain (Fonseca and Martin, 2007; Soffer and Hadar,

2007). In this case designers use proven domain patterns to correct errors in domain-knowledge accuracy detected in the new model.

The approaches above use expert knowledge about design and domain to define credible links between the two. Designers can use this bridge of expertise to link known structural patterns to detected but not-yet-identified domain regularities. Domain-centric and quality-centric design techniques never venture outside their own area of expertise; pattern-assisted design techniques do; hence their utility to designers seeking to implant both functional and quality structures into their models from the start.

However, are design patterns—currently seen as the best way by far to integrate desirable run-time qualities into the software model—not currently geared for use in quality-centric design only? This is the primary goal of refactoring after all. If so, of what practical use are such patterns to software engineers confronted on a daily basis not only with the technical models but also with the ontological realities of the application domain? Is there not growing evidence also that design patterns as currently described are becoming increasingly impossible to use? We consider these and other equally important questions about design patterns as currently defined next.

2.4 Design Patterns

In the late 1970's Christopher Alexander introduced the use of patterns to the field of architecture design (Alexander, 1979). Following his lead the Gang of Four championed the use of design patterns in software engineering in their seminal work, "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma et al., 1995).

Since then both academia and industry have encouraged pattern use in designing software, primarily because patterns seem to solve many of the problems often faced by software engineers. For instance, patterns link design theory to modeling practice (van Diggelen and Overdijk, 2009).

They have known quality characteristics (Gross and Yu, 2001) that designers can use to meet competing end-user requirements (Hsueh and Shen, 2004). Patterns also make software models more resilient to change (Aversano et al., 2007; Izurieta and Bieman, 2007).

The arguments that support pattern use in software engineering are convincing. The utility to novice and experienced designers alike of design patterns as currently described is less so.

2.4.1 Utility to Novice Designers

In theory, design patterns should help novices to acquire design expertise (Lewis et al., 2004) and to practice the analytical skills used in designing software (Muller et al., 2007). Design patterns are schemas that novices can activate to better understand real things (Kohls and Scheiter, 2008). They help novices remove cognitive barriers that impede the efficient acquisition of design and of domain expertise (Kolfschoten et al., 2010). They enhance the analogical-reasoning skills that novices need to transfer domain patterns into the technical realm (Muller, 2005).

In practice, new designers often find existing design patterns confusing and hard to use: a hindrance rather than a help in design activities. This seems to be true both of students (Chatzigeorgiou et al., 2008) and of people with industrial experience (Vokáč et al., 2004). Novices cannot easily apply patterns to problems nor integrate domain knowledge into them (Jalil and Noah, 2007). Nor does understanding a design pattern necessarily result in its accurate identification in requirements described in non-technical terms (Boyer and Mišić, 2009).

It is true that these difficulties may have their source in deficient novice skill. Novices may lack the expertise to abstract or specify knowledge (Wagner and Deissenboeck, 2008), to use natural language to describe a domain correctly (Frederiks and van der Weide, 2006), or to avoid the many errors that threaten the modeling effort (Walia and Carver, 2009). Modeling the complexity of real-world interactions can also be a challenge to new designers (Batra and Wishart, 2004).

On the other hand it is hardly the fault of novices if existing pattern descriptions do not

take into account how people new to design detect patterns in things. At present design patterns seem to be written chiefly to help experienced designers refactor already implemented software. This may explain in part why inexperienced designers find it difficult to use these descriptions to identify patterns in problem-domain realities, or in the software requirements or models that represent them. Using a pattern-description format like that of Shvets (2008), one that uses more familiar language and examples to describe the patterns, may ultimately prove more useful in helping people to detect design-relevant regularities in the problem domain.

2.4.2 Utility to Experienced Designers

In addition there is ample evidence that even experienced software engineers are finding it increasingly difficult to work with design patterns as currently defined. These difficulties seem to stem mostly from pattern proliferation, pattern overlap, and pattern isolation.

Pattern Proliferation

First of all, mastering all design patterns or finding the right one to solve a problem has become more difficult with the increase in the number of pattern variants available. Henninger and Corrêa (2007) have identified the existence of at least 2,241 distinct pattern cases produced by 170 different groups in the pattern-language community. This is simply too many patterns for any designer to ever master or use.

To help contain pattern proliferation, Agerbo and Cornils (1998) suggest that we focus on core design ideas rather than on producing additional design variants. Rost (2004) goes further and says that it is the key structuring idea—and not its many variations—that is the real pattern; he recommends placing patterns in a hierarchy with the most invariant structural ideas at the top. Kniesel et al. (2004) add that it is the more generic aspects of design patterns that often define the most basic structural ideas.

Using formal methods to define pattern-structure elements can also help pattern authors identify duplicates. Some possibilities are the use of pattern-language grammar (Zdun, 2007), pattern templates (Gamma et al., 1995; Buschmann et al., 1996), directed hierarchies (Riehle and Züllighoven, 1996), graph structures (Tsantalis et al., 2006), and static-pattern specification (Kim and Shen, 2008).

Identifying the design ideas essential to software design and using them to structure or to suppress pattern variants should reduce pattern proliferation. Finding a method to identify these key design ideas is especially critical given that the authors above give no indication of how to do so.

Pattern Overlap

A second problem with existing design patterns is that they may be too similar to use in the detection of specific patterns in things. The Gang of Four thought for example that `Wrapper` and `Adapter` were synonyms for the same pattern (Gamma et al., 1995). Studies in automated design-pattern detection have also shown how difficult it is to identify specific design patterns using the model structure of patterns alone (Antoniol et al., 2001; Guéhéneuc et al., 2010).

From this Philippow et al. (2005) argue that the differences between patterns come not from their technical specification but rather from the domain knowledge that the patterns are meant to represent. In fact Noble et al. (2002) insist that design patterns have a metaphoric dimension that allows designers to map them to the differentiated patterns that exist in things; for example, we would use the `State` pattern and not a `Mediator` to model property changes in a `Sheep`.

Important pattern properties can also serve as pattern differentiators on the technical side of things. The Gang of Four for example categorize patterns according to their `creational`, `structural`, or `behavioral` intent (Gamma et al., 1995). Ram et al. (2000) suggest using structural properties like `adaptability` or `extendibility` to differentiate them. McNatt and Bieman (2001) propose to use the strength of pattern coupling as the pattern differentiator instead.

Some authors even define a complete property set that they then use to highlight pattern similarities and differences. For instance, Smith and Stotts (2002) combine elemental patterns like `CreateObject` and `Redirect` to describe different patterns. Rypáček et al. (2006) use a set of operators like `send` and `join` to define how different patterns interact with other model elements. Zdun and Avgeriou (2008) use architectural primitives like `callback` and `indirection` to differentiate patterns by their behaviors.

Describing design patterns using properties that have the power to differentiate both domain regularities and technical structures should eliminate pattern overlap. Unfortunately the properties identified to date are present in too many patterns and so lack the discriminatory power required for precise design-pattern differentiation. As with design ideas, we also need a method to identify the key pattern properties, in this case those with the highest pattern-discriminating power.

Pattern Isolation

A third problem with design patterns is that they are becoming increasingly isolated within distinct pattern cases. Henninger and Corrêa (2007) for instance point out that few of the 2,241 patterns that they reviewed even consider inter-pattern relationships. They suggest that this makes combining patterns to solve more complex domain problems that much more difficult.

Several authors suggest ways to reduce pattern isolation. Noble (1998) for example explains how design patterns `use`, `refine`, or `conflict with` other patterns. Tahvildari and Kontogiannis (2002) show how complex patterns use more primitive ones like `Composite` within layers. Henninger and Corrêa (2007) propose to use ontology to show the semantic associations that link patterns together.

Expecting pattern authors to describe inter-pattern `use` relations may be a bit unrealistic however because no one can really foresee the inter-pattern interactions that a designer may need to model for a given problem domain. More realistic might be to ask pattern authors to give the `is-a`

relation between their design patterns and those already defined in a base pattern set, because `is-a` relations are based on definition and not on use and so should never change.

In particular, pattern authors would need to identify the key design ideas or pattern properties that a new pattern has in common with an existing one, and also those that make it different and unique. For example, do the `Factory Method` and `Abstract Factory` patterns not share a common `factory` function, and is not the difference between them slight if not superficial?

Pattern proliferation, overlap, and isolation all seem to stem from a common cause: that of faulty design-pattern definition. Describing design patterns as isolated pattern cases in particular makes it hard for us to suppress pattern variants, to show the similarities and differences between them, and to reduce their semantic isolation. For these reasons and for those given in the preceding section describing design patterns differently may be a prerequisite to their continued use in software-design activities by novice and experienced software designers alike.

2.4.3 Criteria for Key Pattern Properties

To describe design patterns differently though we must first identify the key pattern properties or design ideas to use in our pattern descriptions.

These pattern properties must meet a strict set of criteria. They must differentiate design patterns so that each is unique and easily recognizable. They must form a hierarchical structure in which pattern variants can inherit the more generic properties of their pattern parents. They must be detectable in requirements and in things so that designers can use them to transfer domain patterns into the design patterns of the technical realm. Above all the properties chosen must have the highest pattern-detection power since this is what will make them most useful to software engineers.

In the next section, we examine research into human cognition. Our goal is to gain insight into what pattern properties should look like if they are to facilitate the transfer of knowledge from the application domain into the technical realm. We leave to the section after that a discussion of

the principles that underlie pattern differentiation.

2.5 Pattern Transfer in Human Cognition

At the heart of software design we find designers. It is the human mind that analyzes the problem in the light of its domain and that engineers a solution to it in a software model. Understanding how the mind transfers information across knowledge domains is a prerequisite then to defining design patterns that support and do not oppose the mind's transfer of domain knowledge into the technical realm.

We consider first how the mind constructs conceptual models of the things in the world. We then investigate how the mind uses patterns and analogical processes to transfer information across knowledge domains. A brief discussion of the implications of these findings on the choice of the pattern properties to use to describe design patterns follows.

2.5.1 Conceptual Models

The mind first constructs a conceptual model of the things in the world, because only then will it have at hand the concepts that it needs to understand that world.

Mandler and McDonough (2000) show that the mind uses surface features (e.g., `red` or `square`) and then simple concepts that categorize features (e.g., `color` or `shape`) to understand things. Learning the relations that link concepts leads to a conceptual network (Hills et al., 2009) that uses ideas to both group and distinguish concept collections (Hammer et al., 2009). The value of a concept to the mind depends primarily on its power to categorize and so to support the mind's recognition and understanding of things (Vanoverberghe and Storms, 2003).

This network of concepts also has a hierarchical dimension: cognitive economy requires that the mind re-use ideas present in more abstract concepts when defining new concepts (Cohen,

2000). This is possible because abstraction reduces the feature specificity of ideas (e.g., `Fido` vs. `dog` vs. `mammal`) which in turn increases the number of other concepts that an idea can relate to (e.g., both `dog` and `cat` are `mammals`).

The mind can use this growing and ever-more complex model of reality to understand the deeper structural regularities present in things (Dibbets et al., 2002; Wiemer-Hastings and Xu, 2005; Westbrook, 2006). Predicting the presence of properties in a thing is also possible when many of a known-model's structural parts are found to already exist within it (Rehder and Hastie, 2004; Coley et al., 2004).

It is possible to assist the mind in its construction of these conceptual models by inserting already-validated models of reality into the mind through education. Doing so can give focus to the knowledge-acquisition process and also improve the robustness of a person's detection of exemplars (i.e., of patterns) in things (Son et al., 2008).

In software engineering, design patterns are the conceptual models that the designer's mind should use to identify the domain patterns present in the things of the problem domain. Acquiring knowledge of design patterns through education can also help software engineers avoid the difficulties of acquiring pattern knowledge through trial and error.

2.5.2 Analogical Processes

Once the mind has mastered the patterns that govern things in one domain, it can use them to acquire knowledge about the things that exist in another. The analogical processes that the mind uses to detect known patterns in still-unknown things are described next.

The mind first applies a known model of reality—a category, a concept, or a more complex pattern—to domain knowledge (Liikkanen and Perttula, 2009), the model used depending on what we are seeking to learn about the domain (Ford, 2004). Doing so immediately populates the model with the domain knowledge that matches the model concepts (Duncan, 2007).

The mind examines the model's alignment with domain regularities (Gentner and Bowdle, 2001), and uses contextual information to filter out concepts not relevant to the problem at hand (Glucksberg et al., 2001; Poitrenaud et al., 2005). It does so because focusing on only relevant concepts can increase the accuracy of the analogical match (Albers, 2007; Chaigneau et al., 2009). Using concept collections that have crystallized into exemplars may also reduce the amount of focusing effort needed (Pierce and Chiappe, 2009). An expert has the ability to define these exemplars and to use them to quickly identify the patterns in things (Popovic, 2004).

Pattern matching usually occurs on concept structures and especially those that are more abstract for two reasons. First of all, many concepts (e.g., `took`) are used across knowledge domains. Hence it is the structural links between concepts (e.g., `bought` = `took` + `paid` vs. `stole` = `took` + `not paid`) that we must use to distinguish things (Gentner and Kurtz, 2006; Alam, 2009). Second, domain specifics are too different to be matched (e.g., `robot` is not `human`). Hence the mind must look to the more abstract concepts (e.g., `attains goals logically`) attached to the domain specifics (e.g., `robot algorithms` and `human rationality`) for the similar concept structures that allow cross-domain knowledge transfer to occur (Mandler, 2000).

Once the mind finds a structural match, it attaches the domain-specific knowledge describing the unknown thing to the concept structures already present in the mind (Chiappe and Kennedy, 2001). This continues down to the level of specificity that the similarity between the concepts in the two domains permit. Analogical processes transfer knowledge across domains only to the extent that the concepts common to both domains permit; after that, human ingenuity must come into play.

In software engineering, a design pattern encapsulates a specific combination of design concepts. The designer's mind applies different design patterns to the problem domain—or to the model that represents it—looking for a match. Patterns are populated with more specific domain knowledge once a match on the higher-level pattern structures is found. This continues until no further matches are possible and the designer must turn to invention to complete the software model.

2.5.3 Implications for Design Patterns

The findings of the cognition sciences have two important implications for the continued use of design patterns in software engineering.

First of all, in describing design patterns we must use pattern properties that include concepts common to both the application domains and the technical realm. The mind requires concepts that are semantically similar to be present in differing domains for information transfer to occur (Kintsch and Bowles, 2002). Without common concepts the domains remain isolated and no mental process exists that can bridge the knowledge gap between them. If software engineers find existing design patterns hard to use it is likely because the concepts now used to describe them obstruct the natural knowledge-transfer mechanisms of the human mind.

Second, we must choose the right concepts to use for the knowledge-transfer task. Using pattern properties that lack the analogical power to transfer knowledge from domain to model may result in contrived models that are not accurate reflections of reality (Pierce and Chiappe, 2009). Finding a method to identify the design concepts that have high analogical power may prove critical to the continued use of design patterns in software design.

Another reason why designers may find existing design patterns hard to use is the current lack of precision in their definitions. If designers are unclear as to what makes each design pattern unique, they will find it hard to use design patterns to identify specific patterns in things: overlapping patterns do not make for precise and accurate pattern recognition.

In the next section, we examine research into pattern differentiation and recognition in the computer sciences. Our goal is to gain insight into how to use the discriminatory power of pattern properties to improve the definition and differentiation of the patterns used in software design.

2.6 Pattern Recognition in Machines

In machine-based pattern recognition a classifier compares the properties of a data instance to those of a defined pattern class and if they are sufficiently similar, identifies the data instance as a member of the pattern class.

If patterns overlap however the classifier may place a data instance in the wrong class. Classification errors will then prompt the use of other pattern properties that better differentiate the patterns and that result in more accurate instance classification. Best are the properties that permit the accurate detection of distinct patterns even in noisy data instances where a pattern is more difficult to discern.

The human mind is regulated by the same logical constraints as machines (von Rooij and Wareham, 2008). As a result the rules that govern pattern use by machines should apply to a large extent to the mind's use of design patterns as well. Hence we examine next the techniques used in the computer sciences to partition the pattern space, to identify high-value pattern features, and to evaluate the discriminatory power of the concepts used in describing patterns.

2.6.1 The Science of Pattern Recognition

Computer scientists have observed that information features that are co-related tend to form pattern clusters (Omran et al., 2007), and that as distinct clusters they partition the pattern space (Alexe et al., 2006). Computer scientists use distance metrics, statistics, rules, decision trees, or other techniques to identify the clusters, depending on their preferred pattern-recognition strategy (Kotsiantis et al., 2006). Partition quality ultimately depends on the compactness of cluster features and the separation between the cluster groups (Cardoso and de Carvalho, 2009).

We should not partition patterns using all of the information features at our disposal however. Ignoring the computational consequences of the curse of dimensionality or using overly-

specific pattern cases that are difficult to find may mean that no patterns are ever detected in the instances. In the end only a judicious selection of the small set of features that efficiently divide up the pattern space will make efficient and accurate pattern-detection possible (Sima and Dougherty, 2008).

There are a number of ways to identify these core differentiating features. Gunal and Edizkan (2008) see benefit in using separability measures to find the features with the highest pattern-discriminating power. Features that have high information gain and appear often (Richards et al., 2006) or that often appear together (Aggarwal and Yu, 2001) are also usually significant. Identifying these key features is also easier if we use filtering to remove redundant or irrelevant features first (Last et al., 2001).

How the classifier measures the match of pattern class and data instance differs according to the pattern-recognition strategy used (Kotsiantis et al., 2006). For the most part if the calculated match crosses a pre-determined threshold then the classifier detects the pattern, otherwise it does not (Jain et al., 2000). If the classifier detects a pattern that it should not, a false positive occurs. If it does not detect a pattern that it should, then a false negative occurs.

Not detecting a pattern in an instance known to contain it indicates that the features used to detect the pattern are at fault (Thabtah, 2007). Further tests of different feature combinations should result in a set that better partitions the pattern space and that permits a more accurate detection of distinct patterns in data instances (Jain et al., 2000). Fürst et al. (2008) describe in some detail a technique that uses pattern-detection error rates to guide the discovery of the optimal pattern-differentiating feature set.

In software engineering, people use existing design patterns—often with some difficulty—to detect structural regularities in the problem domain or a model of it. To remove design-pattern overlap especially we must identify the pattern features that will more sharply partition the design-pattern space. Design patterns described by such features should in turn support a more accurate

detection by all designers of distinct and specific patterns in instances of things.

2.6.2 Implications for Design Patterns

The findings of the computer sciences have two important implications for the continued use of design patterns in software engineering.

First of all, in describing design patterns we must use pattern properties that have high discriminatory power. Software engineers must know the specific features that make design patterns unique if they are to use them to detect specific patterns in the problem domain. The pattern properties picked should also have high analogical power if they are to facilitate the transfer of the patterns detected there into the technical realm.

Second, we must choose the right concepts to use for design-pattern differentiation. Machine based pattern recognition uses pattern-detection error rates on test instances to identify the best features to use to partition the pattern space. In this thesis we use a similar approach to evaluate the analogical and discriminatory power of the pattern-case and pattern-ontology properties used by designers to detect patterns in things. This method is described in some detail in chapter 5, Experimental Method.

2.7 Summary

In this chapter, we examined several domain-centric, quality-centric, and pattern-centric approaches to software engineering. Only the last approach seemed able to preserve both model accuracy and quality, and so to meet both the functional and non-functional software requirements of end-users over the long-term. We then investigated why designers often find existing design patterns hard to use. The reason: the critical high-level design concepts that can address the current proliferation, overlap, and isolation of design patterns have yet to be found.

We then examined the findings of the cognition and computer sciences: how the human mind uses analogy and patterns to transfer knowledge across different domains; and how machines partition the pattern space so that it is easier for them to identify specific patterns in things. From these findings came two conclusions: first, that the design concepts used to define design patterns should have both analogical and discriminatory power; second, that a method is needed to identify the design concepts that do in fact have these powers, powers that are so very necessary to the accurate identification of specific patterns in things.

In the next chapter, Theoretical Model, we use the research findings above to predict how accurately a designer is likely to detect pattern properties in domain knowledge using the traditional isolated pattern-case approach, and the one that this thesis advocates: pattern ontology. This sets the stage for the formulation of the hypotheses of the thesis, as well as for the studies undertaken to test their validity.

Chapter 3

Theoretical Model

The purpose of a theoretical model is to guide scientific research in a field of human inquiry. Theoretical models are not only a source of hypotheses about reality; they are also a guide to the construction of experiments that can validate these hypotheses. Software-engineering research is to some extent based on theoretical models but their use in the validation of hypotheses has so far been limited (Hannay et al., 2007).

In this chapter we present a theoretical model of the cognition processes that underpin software design. We examine the role of design patterns in the transfer of pattern knowledge from the application domain to the technical realm. We use our model to predict the pattern-detection accuracy to be expected from the pattern-case and pattern-ontology approaches to software design.

We base our model on the findings of two sciences in particular. Cognition science shows that the accuracy of knowledge transfer between two domains increases if the concepts used for the transfer are common to both of them. Computer science offers proof that pattern-detection accuracy improves if concepts with high discriminatory power are used to differentiate the pattern space.

Our hypothesis is that designers will detect the properties of design patterns in the problem domain more accurately using a pattern ontology than using isolated pattern cases. We expect this

to be true because pattern ontology uses concepts with high analogical and discriminatory power to describe design patterns while pattern cases do not. We show how the findings of the cognition and computer sciences support this thesis in the sections that follow.

3.1 Human Cognition in Software Design

As mentioned previously, people are at the center of software design. We expect then that the mental mechanisms that people use to interact with the world on a daily basis will also be used to design software. The rules of cognition govern how we think. To design software one must think. Hence we examine the effect of cognition rules on software-design processes next.

3.1.1 Cognition in Design Activities

Studies in cognition science have shown that humans acquire the mechanisms of cognition essential to survival early in life, and that they use these mechanisms throughout their lives to better understand and to organize their world. Human minds detect patterns in things, produce conceptual models of these patterns, and then revise these models of reality to take into account new or changing patterns in the outside world. We see these same mechanisms hard at work in software-design activities as well.

In domain analysis for example software engineers construct domain models that mirror the patterns that they detect in the application domain. In the technical realm they construct technical models using design patterns that allow computers to run software more efficiently. In both fields software engineers use refactoring to correct or refine their models of domain or technical realities. These design activities are illustrated for the two knowledge areas in question in Figure 3.1.

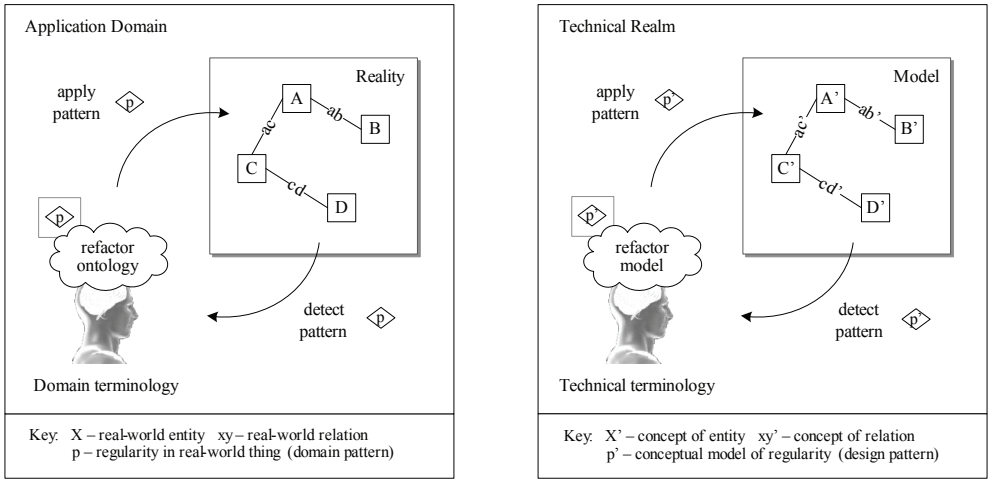


Figure 3.1: Essential design activities in two knowledge areas

The software-design cycles illustrated in Figure 3.1 highlight the fact that the mechanisms that we use to model reality are generally always the same, regardless of the knowledge area. The patterns that people use to model things may vary across individuals, as may the specific knowledge that is the object of their thoughts. However few would dispute the fact that all human minds use more or less the same basic mechanisms of cognition to analyze and to model knowledge.

3.1.2 The Cognitive Gap

In the examples given in Figure 3.1, software engineers work in very specialized fields: that of a domain or of the technical realm. Specialized fields need special terminology, models, and patterns it is true. Even so specialization also has the negative effect of isolating knowledge areas from each other and of making information transfer between them difficult. This is true because knowledge specificity reduces the number of concepts that the knowledge areas have in common.

This cognitive gap exists between the application domains and the technical realm also. Machines are logical and require correct information for computational purposes. Real-world things behave as nature intended and in a much more chaotic way. The concepts used to describe things in domains and in technology are different. This makes knowledge transfer between them difficult. As

seen in the literature review, these differences even influence software-engineering practice: there are domain-centric and technology-centric approaches to software design; a designer will use one or the other depending on whether model accuracy or technical quality is their overriding goal.

As we saw earlier, designing software in a domain- or technology-centric way also has negative consequences. Making changes to the model after the fact displaces important domain or design patterns already built into the model. This may result in software that does not support both the functional and non-functional needs of end-users in the long-term. A reactive approach to software design simply does not bridge the gap between domain and technical knowledge effectively.

Using design patterns to transfer domain patterns into the technical realm at design time can circumvent this problem. Design patterns can act as a bridge between domain realities and the technical realm. The specifics of how design patterns actually help the mind bridge the cognitive gap between the two knowledge areas follow.

3.1.3 Design Patterns as Bridges for Cognition

As mentioned briefly in the literature review on human cognition, a concept is defined by the semantic terms (i.e., the ideas) attached to it. For example, the concept `human` is defined as `rational animal` because human beings have both a mind and a body. Each of these terms is also a concept in its own right however and also has a set of ideas that define it. For example, the concept `rational` includes the terms `logical` and `sequential` while that of `animal` those of `living` and `warm-blooded`.

What we notice in these definitions is that the more specific the concept, the greater the number of terms that it contains, and the more abstract the concept, the fewer. This is the nature of specification and of abstraction. Adding information specifies what we know about a thing. Removing information makes what we know about a thing more general or abstract. For example, specifying the concept `human` with the terms `German`, `academic`, and `inventor of theory of`

`relativity` might lead us to name this new set of terms: `Albert Einstein`. Conversely, abstracting the concept `human` (i.e., a rational animal) by removing its `animal` term leads us to consider only the properties related to `rationality` in human beings.

In addition, because abstract ideas contain fewer terms, the mind can detect them in more specific concepts more easily than the other way around. For example, scientists can use the terms of the formula `velocity = distance / time` to easily detect the key real-world quantities that control the velocity of a moving object. The formulation of the original `velocity` formula took significantly more mental effort because of the need to extract from the specificity of many objects in motion a generalizable pattern.

As stated earlier, the mind transfers knowledge across differing domains by finding the conceptual model that they have in common. In fact the mind attempts to find, in the more specific concepts that describe the things that exist in a domain, the more abstract ideas present in its known patterns of reality. It does so by removing terms from domain concepts (abstraction) and adding terms to pattern concepts (specification) until a match between domain and pattern ideas and conceptual structures is found. This activity constitutes the bulk of the mind's work of bridging the cognitive gap between any two initially-unrelated knowledge domains.

In software engineering, designers should use the patterns proper to the technical realm to transfer the ontological regularities detected in a domain into the technical realm. Design patterns are defined using abstract concepts, for example, `Proxy` is defined as a "surrogate or placeholder for another object to control access to it" (Gamma et al., 1995). Conversely the concepts that describe domain realities (e.g., `Albert Einstein`) are usually quite specific. This means that the mind needs no cognition mechanisms other than those already at its disposal to detect the optimal design patterns to use to represent the ontological regularities detected in things.

Through abstraction and specification the mind can attach the right design ideas to the patterns detected in the domain. As a result design patterns can act as cognitive bridges that facilitate

the transfer of domain patterns into the technical realm. Furthermore only design patterns can do so in a way that preserves both the domain accuracy and the technical quality of the software model.

3.1.4 Barriers to Design Pattern Use

Two barriers to the use of design patterns in software engineering exist however. As mentioned in the literature review, the design ideas currently used to describe design patterns lack analogical and discriminatory power. As a result software engineers find them hard to use in software-design activities that require the use of both analogy and discrimination in modeling the patterns detected in the problem domain.

In the two sections that follow we examine the analogical and discriminatory power of the design concepts used to describe design-pattern cases and design-pattern ontology. This examination will form the basis of our hypothesis about the accuracy in pattern detection that designers can expect when using pattern cases or pattern ontology in software-design activities.

3.2 Design Pattern Cases

As seen in the literature review, pattern authors presently describe each design pattern as an isolated pattern case for use in addressing a specific design defect in a solution model. This approach may be suitable to refactoring software models for quality. It is not suitable however if the designer's goal is to find useful patterns for model design in the knowledge of the problem domain.

The reason why is simple: the technical terms now used to describe design patterns lack analogical power; once the mind abstracts these terms away as part of the analogical knowledge-transfer process the terms lose their power to differentiate the design patterns as well.

3.2.1 Technical Terms Lack Analogical Power

Pattern authors for the most part describe design patterns using terms that are highly technical in nature. The description that the Gang of Four gives for the `Proxy` pattern, for example, uses terms like `initialize objects` and `implement interfaces` to describe the `Proxy` pattern's purpose and behavior (Gamma et al., 1995).

The technical terms used are also highly abstract (e.g., `implement interfaces`); one might think that this makes them suitable for detecting patterns in things. Unfortunately not all abstract terms have analogical power. Technical terms, although abstract, are quite specific to the technical realm and have little meaning outside of it; for example a designer is unlikely to detect a concept like `implement interfaces` in real-world things because these are described using terms like `rational` or `warm-blooded` or `living`.

Technical terms are not natural to the language and realities of the application domain. This creates a significant cognitive gap between domain and technical concepts that the technical terms—tightly bound as they are to the technical realm themselves—cannot truly bridge. This lack of analogical power makes technical terms an unsuitable vehicle for the transfer of domain patterns into the technical realm.

3.2.2 Technical Terms Lose Discriminatory Power

The mind also uses analogical processes to find the common conceptual structures that permit knowledge transfer across domains. As part of the concept-matching mechanism the mind uses abstraction to remove more specific terms from the concepts in both knowledge areas until it finds those that are a match between the two.

The same will hold true when using design patterns. When comparing the technical terms that describe the patterns against the ontological terms that describe a domain the mind will detect the cognitive gap that exists between the two. In response the mind will use abstraction to remove

terms overly specific to each knowledge area, the goal being to find the more abstract terms that are common to both and that will permit a knowledge transfer between them to occur.

This abstraction will result in the removal of technical terms from design-pattern concepts. The justification: these terms are too specific to the technical realm; the desired common concepts are unlikely to be discovered there. However this action also removes the terms currently used to differentiate the design patterns. As a result many patterns at a higher level of abstraction are nearly identical. This is why designers cannot really use design patterns as currently described to identify specific patterns in things.

The effect of abstraction on design patterns is illustrated in Table 3.1 and in Table 3.2.

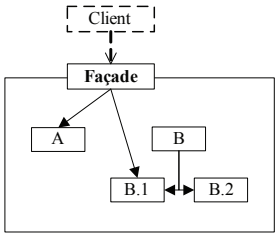
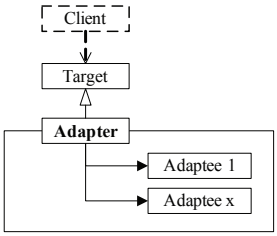
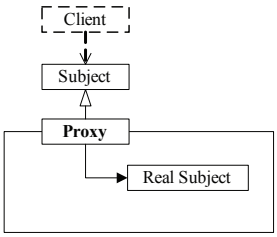
Pattern	Intent	Structure
Façade	A unified interface to a set of interfaces in a subsystem. A higher-level interface that makes the subsystem easier to use.	
Adapter	The conversion of the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	
Proxy	Provide a surrogate or placeholder for another object to control access to it.	

Table 3.1: Three Gang of Four design patterns

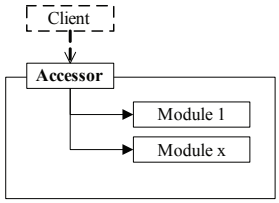
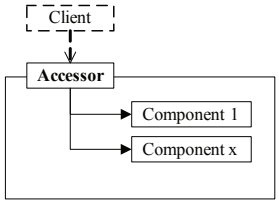
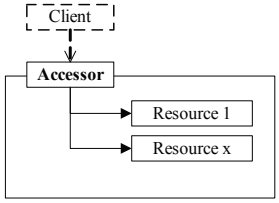
Pattern	Intent	Structure
Façade	Single access to different module(s).	
Adapter	Single access to different component(s).	
Proxy	Single access to different resource(s).	

Table 3.2: Three Gang of Four design patterns after abstraction

As seen in Table 3.1 and in Table 3.2, the effect of abstraction is to reduce the intent and structure of the `Façade`, `Adapter`, and `Proxy` patterns to essentially the same thing: an `Accessor` design idea that isolates a `Client` from a second entity which may be either a module, a component, or a resource. As currently defined, the mind cannot use the `Façade`, `Adapter`, and `Proxy` patterns to identify `isolation`, `adaptation`, or `in persona` patterns in domain knowledge. This is why design patterns described using technical terms are not useful in identifying specific domain regularities (like isolation, adaptation, or acting-as-another) in things.

To summarize: technical terms lack analogical power because they are too tightly bound to the technical realm; they lose their discriminatory power because of abstraction and analogical processes; as a result design patterns as currently described are not useful to software engineers in

detecting specific design patterns in things during software design.

3.3 Design Pattern Ontology

An alternate approach to defining design patterns is to use ontological terms that can bridge the cognitive gap between domain knowledge and the technical realm. Certain ontological terms span application domains and may also be found in the technical realm: this guarantees their analogical power. If concepts with high separability are used for the ontological terms, they will also have discriminatory power. Choosing terms that are abstract will allow designers to use them to detect patterns in more specific things. These criteria are all met in design-pattern ontology, as we shall see next.

3.3.1 Ontological Terms Have Analogical Power

As we saw above, abstraction removes the technical terms from the pattern definition and leaves only the essential design ideas that define each pattern's unique and characteristic intent and structure. Since it is these concepts that the mind uses for the analogical transfer of patterns into the technical realm it is these concepts that we should use to define design patterns in the first place.

The ontological terms used to define the patterns should also be immune to abstraction: they should be easy to detect both in domain patterns and in a specific design pattern; this will ensure that the mind finds the required matching concepts without the need for additional abstraction.

For example, we find the term `in persona`, that is, acting-as-another, embedded in the concepts of many domains: `acting` in theater, `identity theft` in law-enforcement, `social roles` in sociology, even `man-in-the-middle attacks` in computer science. We also find the `in persona` concept in the `Proxy` design pattern, and not in other patterns like `Façade` or `Adapter`. Hence using the ontological term `in persona` to detect a `Proxy` pattern in domains that speak of

`actors` or `identity theft` is likely to result in a solution model that better represents the things in those domains.

Ontological terms that cross application-domain boundaries have analogical power. We can discover the ones of use in software design through a two-step process: first applying abstraction to the technical design patterns to identify the terms that have analogical power; then selecting the ones most useful in mapping the design pattern to the targeted application domains. These are the design ideas that will allow the detection of specific patterns of design in things; these are the ones which for the given design pattern have the highest analogical power.

3.3.2 Ontological Terms Retain Discriminatory Power

Analogical power is not enough however. The ontological terms used for pattern definition must also effectively separate the design-pattern space. A designer must know the specific design ideas that make each design pattern unique. Only then can the designer use a design pattern to detect a specific pattern in the problem domain.

The concept of `sign` for instance can split all numbers into two distinct numeric sets: those that are `positive`, and those that are `negative`. Similarly, an ontological concept like `life` can also split things into two distinct classes: those that are `alive`, and those that are `dead`. Ideas with the power to separate concepts into distinct clusters are key to the partitioning of the design-pattern space. Only these key design ideas can remove the semantic overlap that currently plagues design-pattern definition.

Ideas that differentiate design concepts do exist in the pattern space also. For instance, there is a significant difference between a pattern that `isolates` entities (e.g., `Façade`) and one that enables `sharing` between them (e.g., `Mediator`). There is also a detectable difference between a pattern that simply `transfers` information (e.g., `Façade`) and one that `adapts` it to fit the requirements of a different interface (e.g., `Adapter`).

We can use these and other design ideas with like discriminatory power to efficiently partition the design-pattern space.

3.3.3 Understanding Pattern Ontology

In fact, we discern in the technical descriptions of existing design patterns only a small set of key design ideas that seem to have this differentiating power: `isolate` vs. `share`; `transfer` vs. `mediate`; `provide` vs. `collaborate`. Some of these key design ideas also seem to be present in multiple patterns, for example, the idea of `isolate` in the `Façade`, `Adapter`, and `Proxy` patterns. Grouping patterns according to the design idea that they have in common and separating them by design-idea differences gives us the pattern ontology shown in Table 3.3.

1 Isolate		
	1.1 Transfer	
	Facade	Isolate components internal to the system
	Wrapper	Isolate component external to the system
	Delegate	Represent a component internal to the system
	Proxy	Represent a component external to the system
	Strategy	Plug an exchangeable operation into the system
	Provider	Plug an exchangeable component into the system
	1.2 Mediate	
	Adapter	Map external component interface to system interface
	Bridge	Map internal component interface to system interface
	Mediator	Map interface for bi-directional interaction
	Broker	Bi-directional rule-based component and system interaction
	Interpreter	Translate component interactions with system interface
2 Share		
	2.1 Provide	
	Repository	Aggregate the data of multiple sources
	Database	Communicate data using a protocol
	Server (Client)	Provide data or data-transformation services to clients
	2.2 Collaborate	
	Publisher (Subscriber)	Registration controls component interactions
	Producer (Consumer)	Semaphores control component interactions
	Blackboard	Heuristic rules control component interactions
3 Activity		
	3.1 Detect	
	Polling	Monitor component change at periodic intervals
	Listener	Monitor state change in a component in real-time
	Observer	Monitor event change in a component in real-time

Table 3.3: Pattern Ontology – Using design ideas to group and to differentiate design patterns

Interestingly enough the pattern ontology above meets the criteria that we identified in section 2.4.3 for key pattern properties. The design ideas differentiate design patterns so that they are unique and easily recognizable. The design ideas define a hierarchical structure in which pattern variants inherit the more generic properties of their pattern parents. The design ideas cross application domains and so have analogical power. They also separate the design-pattern space and so have discriminatory power.

Pattern ontology seems well suited to bridging the cognitive gap between different domains and the technical realm. The pattern-mapping function that it can play in software-design activities is illustrated in Figure 3.2.

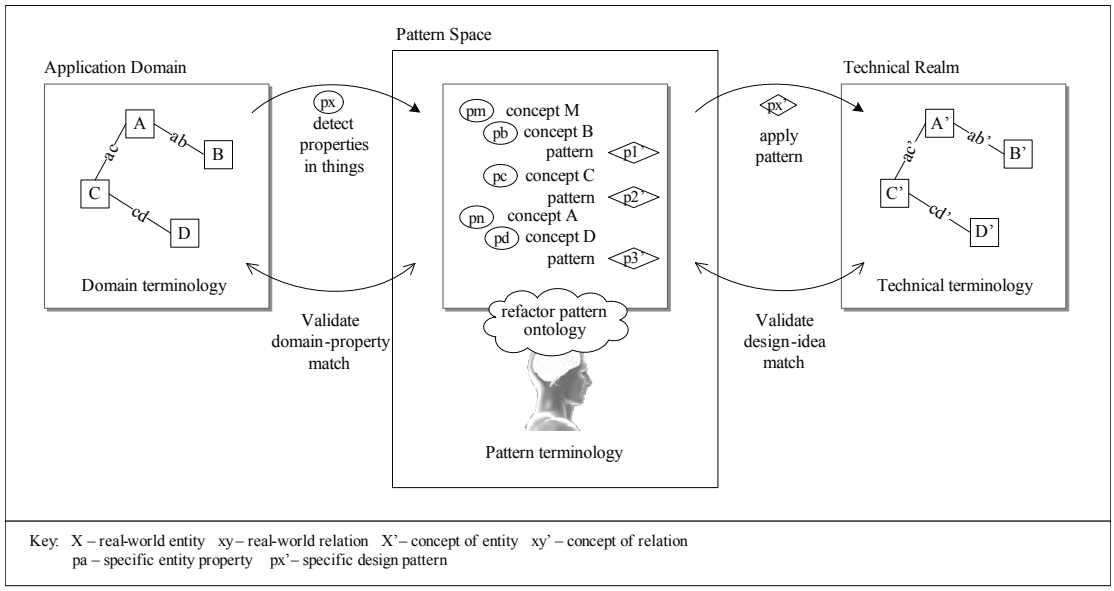


Figure 3.2: Pattern Ontology – Bridging the cognitive gap between domains and the technical realm

The identification of other key design ideas and pattern ontologies is possible, to be sure. However we expect that it is only by placing design patterns in ontologies that we will address the very real and still unresolved problems of pattern proliferation, overlap, and isolation.

Pattern ontology also seems closer to what Christopher Alexander would call a pattern language. In his own words:

> Suppose now, for a given act of building, you have a pattern language, and that the patterns in this language are arranged in proper sequence. To make the design, you take the patterns one by one, and use each one to differentiate the product of the previous patterns... Again, the patterns operate upon the whole: they are not parts, which can be added—but relationships, which get imposed upon the previous ones, in order to make more detail, more structure, and more substance." (Alexander, 1979)

To summarize: ontological terms have analogical power because they exist as concepts in multiple domains as well as in the technical realm; the ones extracted from technical design patterns also have discriminatory power that makes them immune to abstraction processes; pattern ontology is simply the hierarchical ordering of the design patterns according to the key design ideas that group and differentiate them; as a result design patterns described in a pattern ontology should be of great use to software engineers that have as a goal the detection of specific design patterns in solution models, end-user requirements, and things.

3.4 Pattern Detection Accuracy

We consider next the accuracy in pattern detection that pattern cases and pattern ontology permit. The analysis given here of the analogical and discriminatory power of the design ideas present in pattern cases and ontology are the basis of the hypotheses given in the next chapter.

3.4.1 Accuracy due to Analogical Power

As we saw above, the use of technical terms reduces the analogical power of pattern-case descriptions. The use of ontological terms, on the other hand, increases the analogical power of pattern-ontology descriptions. The effect of analogical power on pattern-detection accuracy is illustrated in Figure 3.3.

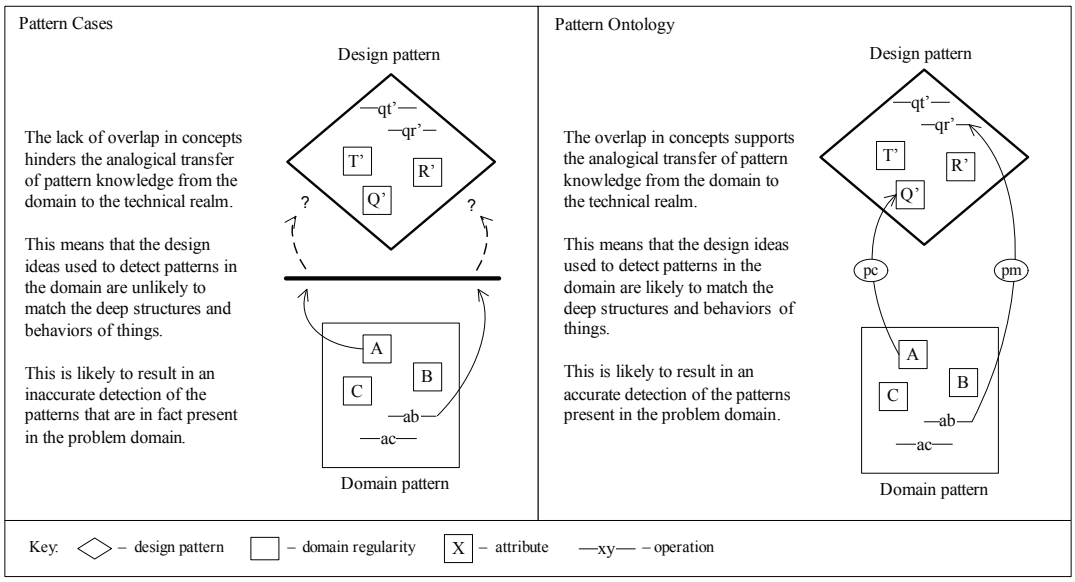


Figure 3.3: Analogical power and pattern-detection accuracy – Cases vs. ontology

The measure of accuracy in analogical processes is the degree of similarity between the concepts used for pattern transfer across domains. As illustrated in Figure 3.3, the technical concepts used to define pattern cases have little to no similarity to those used to describe the problem domain. Hence it seems safe to predict that a designer's detection of design ideas and patterns in the domain using pattern cases is likely to be inaccurate.

Conversely the design ideas used to define pattern-ontology also describe the things in the domain. Our prediction here then is that design idea and pattern-detection using pattern ontology is likely to be more accurate than when using pattern cases.

3.4.2 Accuracy due to Discriminatory Power

Using technical terms also reduces the discriminatory power of pattern cases, while using terms with the power to partition pattern space increases that of pattern ontology. The effect of concept discriminatory power on pattern-detection accuracy is illustrated in Figure 3.4.

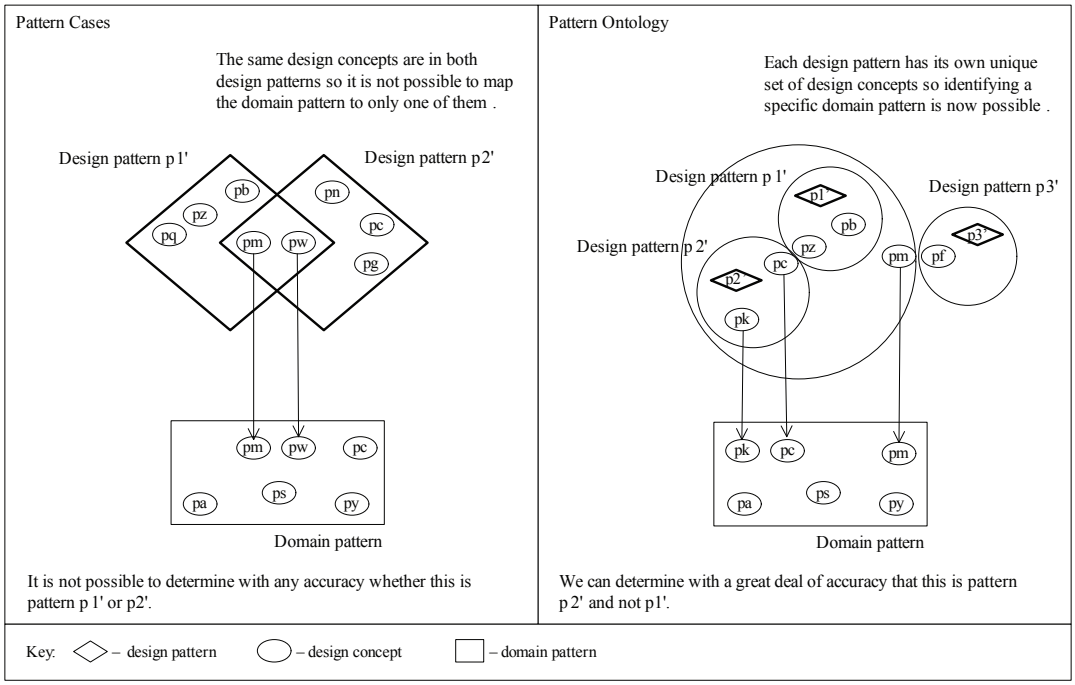


Figure 3.4: Discriminatory power and pattern-detection accuracy – Cases vs. ontology

The measure of accuracy in discriminatory processes is the degree of dissimilarity between pattern concepts that permits the identification of a unique pattern in a domain. As illustrated in Figure 3.4, abstraction causes pattern cases to contain near-identical design ideas, while those defined in pattern ontology never lose their distinctiveness.

Hence it seems safe to predict once again that designer accuracy in detecting design ideas and patterns in the domain will be higher when using pattern ontology than when using pattern cases.

3.4.3 Measuring the Accuracy of Pattern Detection

Finally we describe the method used in the thesis studies to measure the accuracy of a designer's detection of core design ideas and of patterns in domain knowledge. Note that a more detailed explanation of the method used is given in chapter 5, Experimental Method.

We first integrate a known pattern into a requirement description for software in a specific problem domain. Software designers new to design patterns are then taught either pattern cases or pattern ontology. They are asked to detect the design ideas and patterns present in the requirement description, within tests used to capture the number of their accurate and inaccurate detections. Test results tell us the degree of pattern-detection accuracy to be expected when new designers use pattern cases or pattern ontology to identify regularities in a problem domain as described in a software-requirement description.

In the case of pattern ontology, errors in pattern detection can also guide changes to the design ideas used to differentiate the design patterns. The purpose of this refactoring of the pattern ontology is to produce an improved version that has even greater pattern-detecting power.

3.5 Summary

In this chapter we presented a theoretical model of the cognition mechanisms underlying software design. We examined the use of cognition in software-design activities, and of design patterns as a cognitive bridge between the application domain and the technical realm.

We looked into why ontological terms have analogical and discriminatory power, and why technical terms do not. A description of pattern ontology followed. These considerations led to a prediction of the accuracy in pattern detection to be expected from a designer using the pattern-case or the pattern-ontology approach to software design.

In the next chapter we define the problem, proposed solution, and hypotheses of the thesis. A description of the experimental method used to validate the hypotheses follows next.

Chapter 4

Problem, Solution, and Hypotheses

We now identify the problem with using pattern cases to detect design ideas and patterns in the problem domain, our proposed solution to this problem, and our hypotheses about the detection accuracy to be expected when using pattern cases or pattern ontology in software-design activities.

4.1 Problem

The proliferation, overlap, and isolation of design patterns and the present need to refactor substandard software models are symptoms of a much deeper problem: the design ideas currently used to describe design patterns do not truly differentiate them nor do they support the accurate transfer of application-domain patterns into the design patterns of the technical realm.

Pattern authors currently use technical terms to describe design patterns. Technical terms lack analogical power and lose their discriminatory power during the analogical knowledge-transfer process. As a result both novice and experienced designers find it difficult to use existing design patterns to identify specific design ideas or patterns in things. This results in an inaccurate detection of patterns of design in the problem domain. This in turn leads to software models that in the end will lack domain accuracy, or technical quality, or both.

At the present time software engineers cannot use design patterns to model the problem domain in a way that from the outset meets both the functional and quality requirements of end-users. Given the cost of maintaining software over the long-term this is a significant problem indeed.

4.2 Proposed Solution

The solution to this problem is to define design patterns differently. We must first extract the key design ideas present in traditional design patterns and from these select those that have the highest analogical and discriminatory power. We must then use the ontological versions of these design ideas to group and to differentiate the design patterns in a pattern ontology.

Software engineers can use the pattern ontology to detect specific design ideas in the regularities of the problem domain and to link them to a specific design pattern. This will result in a more accurate transfer of specific domain patterns into the design patterns of the technical realm.

Pattern ontology should result in software models that from the outset support both the functional and non-functional requirements of end-users in the problem domain. In other words, pattern ontology should improve the quality of software design.

4.3 Hypotheses

Our thesis rests on the following three hypotheses for pattern cases and pattern ontology:

H_0 There is no significant difference in detection accuracy when using pattern cases or when using pattern ontology to detect design ideas and patterns in a problem domain.

H_1 Using pattern cases to detect design ideas and patterns in a problem domain results in low detection accuracy.

H_2 Using pattern ontology to detect design ideas and patterns in a problem domain results in higher detection accuracy than using pattern cases.

H_0 is the null hypothesis. Hypotheses H_1 and H_2 are based on the predictions about pattern cases and pattern ontology given in section 3.4, Pattern Detection Accuracy. We proceed now to a description of the experimental method used to validate the hypotheses above.

Chapter 5

Experimental Method

In this chapter we describe the experimental method and the studies used to validate the two hypotheses of the thesis.

We first identify the experimental factors and the subject group used in our two studies. We describe the treatments and tests used in each study. We also step through the procedure used to apply the pattern-case and pattern-ontology treatments to the subject group and to test for post-treatment effects.

In the next chapter we examine study test results in the light of our theoretical model to see whether our predictions about pattern cases and pattern ontology were correct.

5.1 Experimental Factors

The independent variables used in the studies are *Pattern Property*, *Requirement Set*, and *Design Idea Format*. The dependent variable is the *Detected Property*. The measure of the accuracy of property detection is *Detection Accuracy*.

Pattern Property refers to any one of the many design concepts used to define a design pattern. For the purposes of our two studies, a `design pattern` is defined as `a collection of`

`related design ideas`. For the most part study tests require that subjects use either a specific design idea or a named design-idea collection (i.e., a pattern) to identify the domain patterns in the requirement set.

Requirement Set refers to a description of what the end-users require of the software. In requirement sentences we use domain language to describe a domain pattern that mirrors the design ideas or pattern that a test targets. In each test subjects try to detect the design idea or pattern integrated into the requirement sentences. We expect the accuracy of their detection of the domain pattern to vary according to the design-idea format used by the participant in the test.

Design Idea Format is the approach used to describe a design pattern: either as an isolated pattern case or within a pattern ontology. Pattern cases use technical concepts and model diagrams to describe design-pattern properties. Pattern ontology uses ontological concepts instead, and in a hierarchy that distinguishes design patterns by their key design ideas.

Detected Property is the design idea or pattern that the subject actually detects in the requirement set during a test. If a subject detects the wrong design idea or pattern, it is an inaccurate detection. Otherwise, it is an accurate detection. In tests involving pattern cases only design-pattern names can be given to subjects to use in identifying design ideas in requirements, because design ideas only exist in pattern cases in an undifferentiated form and as part of the technical description of the pattern. In the case of pattern ontology however both the names of specific design ideas and of formal design patterns can be provided.

Detection Accuracy is measured by the number of subjects in the subject group that accurately detect a design idea or pattern in the requirement set during a test. For example if half of the subjects identify the right design idea in a test then the detection accuracy for that test is 50%; if only a third, then 33%. Detection accuracy is linked directly to the design-idea format that the subject uses to complete the test. Hence test results should reflect the pattern-detection power of using either pattern cases or pattern ontology to detect the patterns of design in things.

5.2 Subject Group

The composition, size, and motivation of the subject group participating in the two studies can affect the validity of study test results. We consider each of these factors in turn.

5.2.1 Composition

Knowing the composition of the subject group allows us to extrapolate test results to the population at large. In our case, participants must have enough pattern knowledge to learn pattern cases and pattern ontology, but not so much that study treatments have no effect on them. Participants must also be familiar enough with the application domain to be able to detect patterns of design in the requirement set.

Subject accuracy in detecting design ideas must also be due only to the design-idea format used in the test. If participants cannot learn design ideas and patterns or have a pre-existing mastery of them, or if they are unfamiliar with the domain knowledge used in the test, then the treatment might not be the sole cause of the observed detection-accuracy effect.

For the thesis studies we use a subject group that meets the composition criteria above to ensure that our test results retain both their external and internal validity.

5.2.2 Size

To produce a statistically valid result the size of the subject group must not be too small. Larger sample sizes ensure that an observed effect is not unique to a few select individuals; they also increase confidence that test results do in fact apply to the population at large.

Dybå et al. (2006) report that of the 5,453 published studies in software engineering that they examined, 92 of them followed experimental procedures in testing a hypothesis. The subject sample size used in these studies varied widely (from 26 to 119 for the mean and from 15 to 65 for

the median) but on average was 48 (σ = 51) with a median size of 30. We use subject groups of a similar size to ensure that our test results have statistical validity.

5.2.3 Motivation

Finally, participants must be motivated to put effort into the treatment and test activities. Once again, accuracy in design-idea and pattern detection must be due to the design-idea format used in the test, and not in this case to a lack of participant effort in acquiring pattern knowledge or in detecting patterns of design in the requirements during a test.

Since pattern ontology uses design ideas extracted from design patterns taken from the pattern catalogs, and since learning how to use design patterns effectively in software-design activities is essential to software engineering, the study of pattern ontology finds a natural place in a course in software engineering. Software-engineering students are motivated by bonus marks and tests both to learn and to apply their learning correctly. We use both to ensure that our test results are not negatively affected by poor subject motivation.

5.2.4 Study Participants

As a result of the considerations above we chose as study participants Computer Science students enrolled in a first- or second-year course in software engineering.

These students had all passed the prerequisites for the course in software engineering. As a result they had experience with the simple patterns used to develop software, but also limited to no experience using named design patterns to guide the work of software design. They also had knowledge of application domains related to the technical realm, for example, how to use a database or a network to support business function. This ensured that test results were due to the design-idea format learned and then used in the tests and not to other extraneous factors.

Two different classes participated. In the first group were forty-five students and in the

second forty-one. These group sizes are similar to those used in other software-engineering studies and so seemed sufficient to ensure the statistical validity of study results.

Finally, students were given bonus marks for participating. Since design patterns are an important part of the software-engineering curriculum they were also tested on their mastery of pattern knowledge in two course quizzes: one on pattern cases and one on pattern ontology. These marks seemed sufficient to encourage the students to put real effort into learning design ideas and patterns and in using them correctly in study tests.

A questionnaire given at the start of each course evaluated student fitness to participate in the study. Questionnaire results (given in Appendix A) confirm that the students participating in the two studies met our criteria for the subject group. As a result we expect study results to be applicable to the population at large, that is, to software engineers new to using design patterns to identify ontological regularities in the problem domain within software-design activities.

5.3 Treatments and Tests

We use two studies to test the hypotheses about pattern cases and pattern ontology. The treatments and tests used in the studies are described next.

5.3.1 Study 1 and Study 2 – Pattern Cases

In both the first and second study we test the ability of participants to detect pattern properties in software requirements using pattern cases.

We include a pattern-case test in both studies for three reasons. First of all, we need to determine how accurately each subject group is able to detect patterns of design in the requirements using pattern cases. These test results are needed to validate the first hypothesis. Second, we need a baseline to which we can compare the pattern-ontology test results. Both test results are needed

to validate the second hypothesis. Finally, duplicating the test addresses some of the concerns that some authors have raised about software-engineering experiment repeatability (Miller, 2005).

In the pattern-case part of each study we do the following: 1. We integrate the pattern case into requirement sentences using the language of the target application domain. 2. We apply the pattern-case treatment: in this case participants study the technical design patterns to be used in the pattern-detection test. 3. We test participant use of pattern cases in identifying patterns in the requirements. 4. We evaluate participant pattern-detection accuracy individually and as a group.

The materials used for the pattern-case part of both studies are described in more detail in Appendix B.

5.3.2 Study 1 – Pattern Ontology

In the first study we test the ability of participants to detect pattern properties in requirements using a pattern ontology.

In the pattern-ontology part of each study we do the following: 1. We integrate the design idea or pattern into requirement sentences using the language of the target application domain. 2. We apply the pattern-ontology treatment: in this case participants study the ontological design ideas and patterns to be used in the pattern-detection test. 3. We test participant use of pattern ontology in identifying patterns in the requirements. 4. We evaluate participant pattern-detection accuracy individually and as a group.

The materials used for the first pattern-ontology study are described in Appendix C.

5.3.3 Study 2 – Pattern Ontology

In the second study we again test the ability of participants to detect patterns in requirements using pattern ontology. The materials used for this study are described in Appendix D. Some of the results in the first study prompted changes to the ontology and to the requirement set used in

the second. The reasons for these changes are described next.

Refactoring the Pattern Ontology

As we shall see in section 6.2, the first study confirms our predictions about pattern ontology: pattern ontology does result in a more accurate detection of domain patterns in requirements than pattern cases. In some cases however the detection accuracy is still less than satisfactory. Pattern-detection failures are not a cause of alarm however. To the contrary they have great value in assisting researchers in refactoring a pattern ontology so that the design ideas used within it have increased analogical and discriminatory power.

In machine-based pattern recognition the classifier uses the error rate to guide its selection of the features used to partition the pattern space. In machine learning especially the combination of features that gives the lowest classification error on the test instances is usually the optimal set, all things considered. In our case the error rates guide us to low-performing design ideas. We can change these design ideas and even the structure of the ontology through refactoring. The revised ontology should result in improved pattern-detection accuracy in subsequent tests. If it does not we need only repeat the process until the level of detection accuracy achieved is satisfactory.

Our thesis is that using pattern ontology can improve the quality of software design in the long-term too. Refactoring activities allow us to increase the pattern-detection power of the design ideas used in a pattern ontology through objective tests and over time. Needless to say, this is simply not possible using pattern cases.

Addressing Requirement Ambiguity

There is another possible cause for reduced detection accuracy. The problem may not be due to the design ideas used for pattern detection; it may be due to the lack of domain-pattern specification in the requirement set.

In machine-based pattern recognition the classifier may have difficulty classifying a data instance if it is missing feature data essential to classification. In such cases we must add information to the data instance so that it is classified correctly. Another option is for the classifier to place the instance in the class that matches it best, if this is possible.

In our case requirements may be missing some of the design ideas necessary for exact pattern detection. If this occurs a designer is wise to acquire more details about the domain pattern so as to link it to as specific a design pattern as possible in the ontology. Another option is for the designer to identify the design ideas present in the requirements, and then to adapt one of the patterns that contains the design ideas to match the pattern in the domain. The option chosen will depend of course on how closely the domain pattern matches available design ideas.

In this thesis we do the following to test whether pattern ontology retains its pattern-discriminating power even in the face of requirement ambiguity:

1. We integrate design ideas into the requirements so that two patterns of design can solve the domain problem. The design idea that differentiates the two targeted patterns of design however is left out so that the requirements become ambiguous (i.e., they lack specification).

2. As before, we apply the pattern-ontology treatment and test participant use of pattern ontology in identifying patterns in the requirements.

3. We evaluate participant pattern-detection accuracy individually and as a group. The expectation is that participants will exclude all but the two targeted patterns of design from consideration. Participants will then select one or the other pattern as the solution to the problem, the choice depending on each participant's resolution of the ambiguity in the requirements.

If participants are able to exclude all patterns other than the two targeted ones using the one design idea, but if there is evidence that they cannot differentiate between these two because the design idea necessary to do so is not clearly specified in the requirements, then the test will

show that pattern ontology retains its ability to discriminate between patterns in the domain up to the point that the requirements lose the specificity required for more accurate pattern detection.

5.4 Procedure

We used the following procedure to regulate study activities.

At the beginning of the course students filled in the questionnaire used to evaluate their fitness to participate in the study. They were also informed at this time about the study on design patterns, the two quizzes, and the bonus marks attached to participation. Eighty-six students agreed to participate in the two studies.

In the first treatment, the instructor taught participants the targeted pattern cases in a one-hour class. A closed-book quiz lasting thirty minutes followed a week later. This quiz evaluated participant accuracy in detecting patterns of design in requirements using pattern cases.

In the second treatment, the instructor taught participants the pattern ontology specific to the first or second study, again in a one-hour class. A closed-book quiz lasting thirty minutes followed a week later. This quiz evaluated participant accuracy in detecting patterns of design in requirements using pattern ontology.

In both cases, participant results were marked for accuracy according to the evaluation grid specified for each test. The test results were then analyzed in the light of the theoretical model to see if they supported or contradicted the hypotheses.

5.5 Summary

In this chapter we described the experimental method used to validate the hypotheses.

We considered the experimental factors, the subject group, the treatments and tests used to test pattern-detection accuracy using pattern cases and pattern ontology, and the procedure used to

regulate the studies. We also examined the use of pattern-detection errors in guiding the refactoring of a pattern ontology, as well as the extent to which pattern ontology retains its pattern-detection power even in the face of requirement ambiguity.

We examine study results in the light of the theoretical model next.

Chapter 6

Findings

In this chapter we give the results of the pattern-case and pattern-ontology studies and examine them in the light of the theoretical model. The raw test results are given in Appendix E.

We first consider the results of the first study. A brief discussion of the reasons for the changes made to the pattern ontology and requirements used in the second study follow. We then consider the results of the second study. We close with an evaluation of the hypotheses in the light of the significance (p-values) of the different study findings.

6.1 Study 1 – Pattern Cases

In this section we examine the results of the first pattern-case study. The treatment, requirements, and test questions used in this study are given in Appendix B.

In the first question we asked participants to identify all the patterns that they detected in the requirements. As seen in Figure 6.1, over half of the participants detected patterns of design that were simply not there: a `Façade` (48.9%), a `Proxy` (53.3%), and a `Bridge` (60.0%). On the other hand all the participants accurately detected the `Shared Data` pattern (100%). The latter result is likely due to the fact that the `Share` design idea has high analogical and discriminatory power. It is

for this reason that we also include it in our pattern ontology.

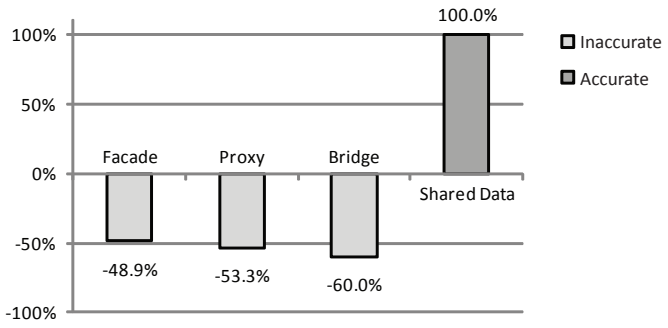


Figure 6.1: Study 1 Pattern Cases – Identifying patterns in requirements

In the second question we expected participants to match the requirement "application... easily switched to (another) component" to the `Wrapper` pattern. As seen in Figure 6.2, only about a quarter of participants (22.2%) were able to do so.

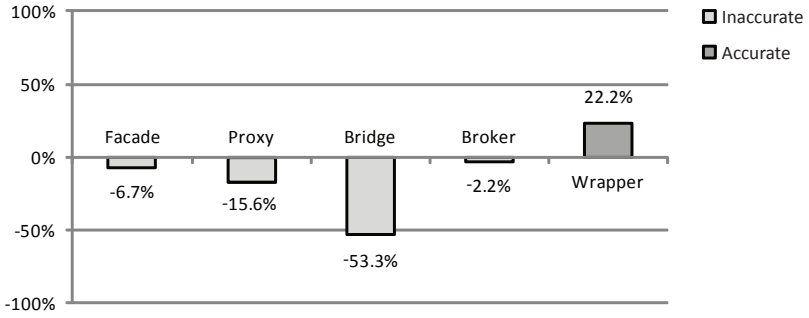


Figure 6.2: Study 1 Pattern Cases – Distinguishing the Wrapper pattern

As predicted by the theoretical model, patterns with overlapping semantics are difficult for people to detect precisely in domain knowledge. `Façade` and `Proxy` contain elements of isolation while `Bridge` contains an element of function switching. In this case however the ability to change a component points to the `Wrapper` pattern specifically. That less than one out of four participants could use pattern-case descriptions to distinguish between these patterns and to pick the one that actually matched the requirements is significant.

In the third question we expected participants to match the requirement "store is connected to the central... computer for all credit card transactions" to the `Client-Server` pattern. In this case participants varied widely in their choice of the pattern that best fit this requirement, as shown in Figure 6.3.

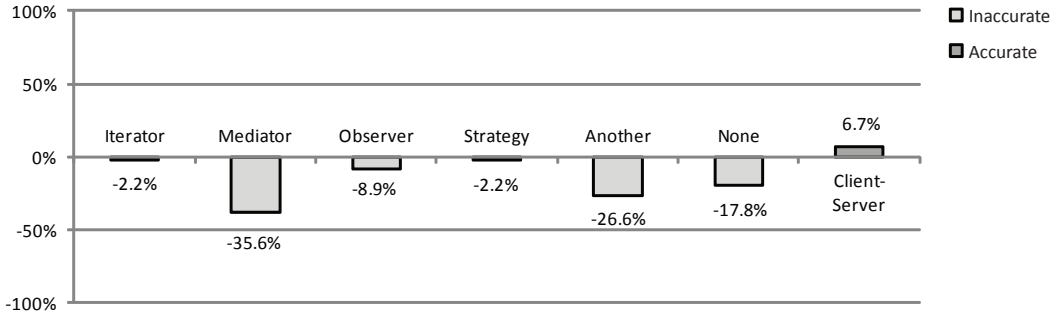


Figure 6.3: Study 1 Pattern Cases – Mapping requirements to the Client-Server pattern

`Mediator` was a popular choice (35.6%) as were other patterns (26.6%). Some even said that there was no pattern discernible here at all (17.8%). However only one central computer is mentioned and it is providing a credit-card service to each store: this is a `Client-Server` pattern first and foremost. Unfortunately only a handful of participants (6.7%) were able to use this pattern's technical description to detect the right domain pattern in this problem domain.

6.2 Study 1 – Pattern Ontology

In this section we examine the results of the first pattern-ontology study. The treatment, requirements, and test questions used in this study are described in Appendix C.

In the first question we asked the same subjects to identify patterns of design in the requirements but this time using design ideas with higher analogical and discriminatory power. As a result and as can be seen in Figure 6.4, subject pattern-detection error rates decreased significantly.

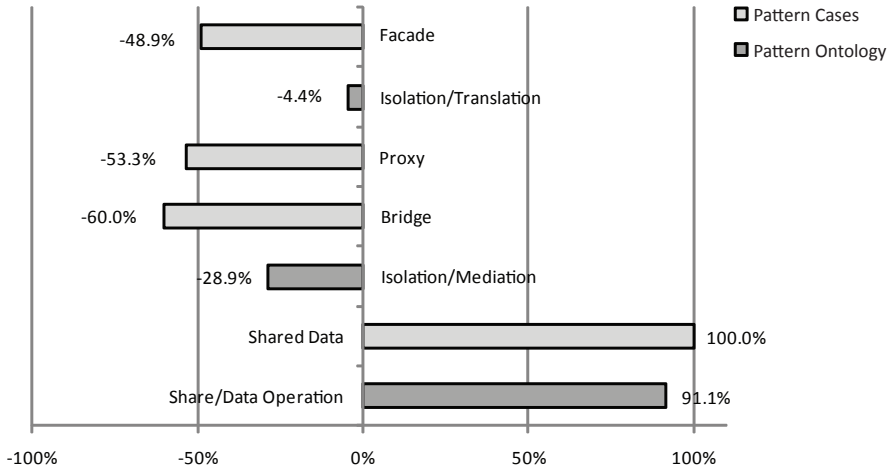


Figure 6.4: Study 1 Pattern Ontology – Identifying design ideas in requirements

In the pattern-case study about half the participants incorrectly detected the `Façade`, `Proxy`, and `Bridge` patterns in the requirements. In this study the results were significantly different: detection errors of only 4.4% for `Isolation/Translation` and 28.9% for `Isolation/Mediation`. As expected, using the `Share` design idea also gave good results (91.1% detection accuracy) within the `Share/Data Operation` pattern set.

In the second question we asked participants to detect in the requirements the patterns of design that underpin the `Wrapper` pattern. The results are shown in Figure 6.5.

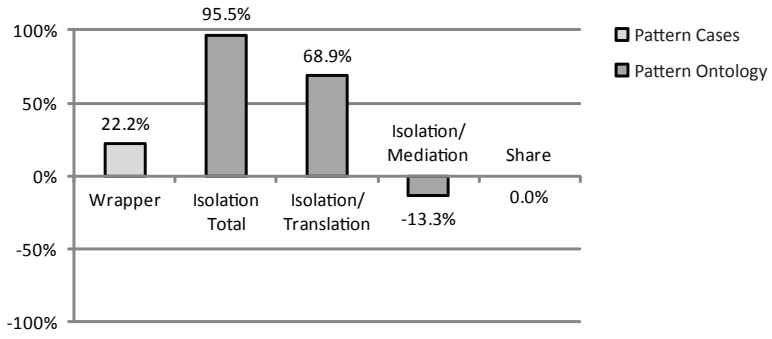


Figure 6.5: Study 1 Pattern Ontology – Distinguishing more precise patterns of design

Using pattern cases only 22.2% detected the set of design ideas that define the `Wrapper` pattern. Using pattern ontology 95.5% accurately detected the `Isolation` design element and 68.9% the `Isolation/Translation` pattern set.

Pattern cases use technical terms that make the detection of patterns in the problem domain difficult. Pattern ontology uses ontological terms that differentiate patterns according to their core design ideas. These findings in particular demonstrate that pattern ontology results in a more precise and accurate detection of pattern parts in domain knowledge than the pattern-case approach.

Finally in the third question participants were asked to detect the pattern present in the requirement "rural computers connect to the central MTS computer for Internet browsing". In this case the results using pattern ontology were inconclusive, as shown in Figure 6.6.

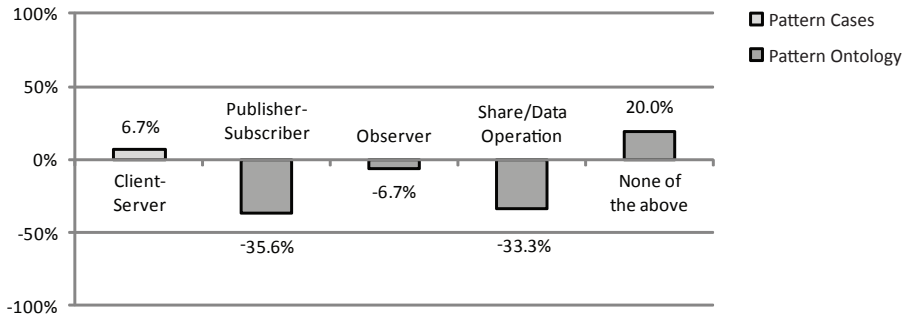


Figure 6.6: Study 1 Pattern Ontology – Detecting flaws in a pattern ontology

Although only 6.7% of subjects correctly identified a `Client-Server` pattern using pattern cases, significant errors in detection accuracy also occurred using the pattern ontology: 35.6% for `Publisher-Subscriber` and 33.3% for `Share-Data Operation`. The correct answer here is `Isolation/Mediation` or more precisely `Proxy`. That four-fifths of the participants (80.0%) did not detect this pattern in the requirements using this pattern ontology is significant.

The results for the first two questions were as expected. Test results for the third question however prompted changes to the pattern ontology and requirements used in the second study. The

changes made are described next.

6.3 Addressing Errors in Pattern Detection

To recognize a pattern in domain knowledge the mind must find a match between the domain and the design pattern concepts. This means that pattern-detection failures may be due as much to poorly-defined requirements (poorly-chosen domain concepts) as to faulty pattern descriptions (poorly-chosen design concepts). We consider the second case first.

6.3.1 Errors due to the Pattern Ontology

As mentioned in the previous chapter, we can use detection-error rates to identify deficiencies in a pattern-ontology and to guide the refactoring effort. The design ideas in the original pattern ontology that had higher-than-expected error rates are shown in Table 6.1.

Detection Errors	Likely Cause	Refactoring Required
Question 1 & 2 Translation vs. Mediation Adapter	The design idea Translate contains an *adapt* term that make it too similar to Mediate. This makes it hard for participants to distinguish between patterns that pass data and those that make changes to it.	Change the design idea from Translate to Transfer to better distinguish the two pattern groups. This in turn will result in the Adapter pattern moving to the Isolate/Mediate group.
Question 3 Proxy	Participants were not able to detect a Proxy as an Isolate/Mediate in the requirements. The *in persona* term, though an important feature of this pattern, does not seem to have sufficient discriminatory power by itself to distinguish this pattern from others.	Abstract out the *in persona* term so that the design ideas that better characterize the fundamental aspects of the Proxy pattern predominate. The Proxy pattern now moves to the Isolate/Transfer group.
Question 3 Data Operation Observer	The technical design idea Data Operation applies to any information transfer. Hence this term lacks pattern-differentiating power. The Observer pattern is also misplaced: as a pattern it is an activity-detecting mechanism and not one that supports the sharing of information.	Change the technical design idea Data Operation with one that has better analogical and discriminatory power, e.g., Provide vs. Collaborate. Move the Observer pattern to a new Activity/Detect pattern group.

Table 6.1: Using detection errors to guide the refactoring of pattern ontology

Refactoring led to the revised ontology shown in Figure 6.7.

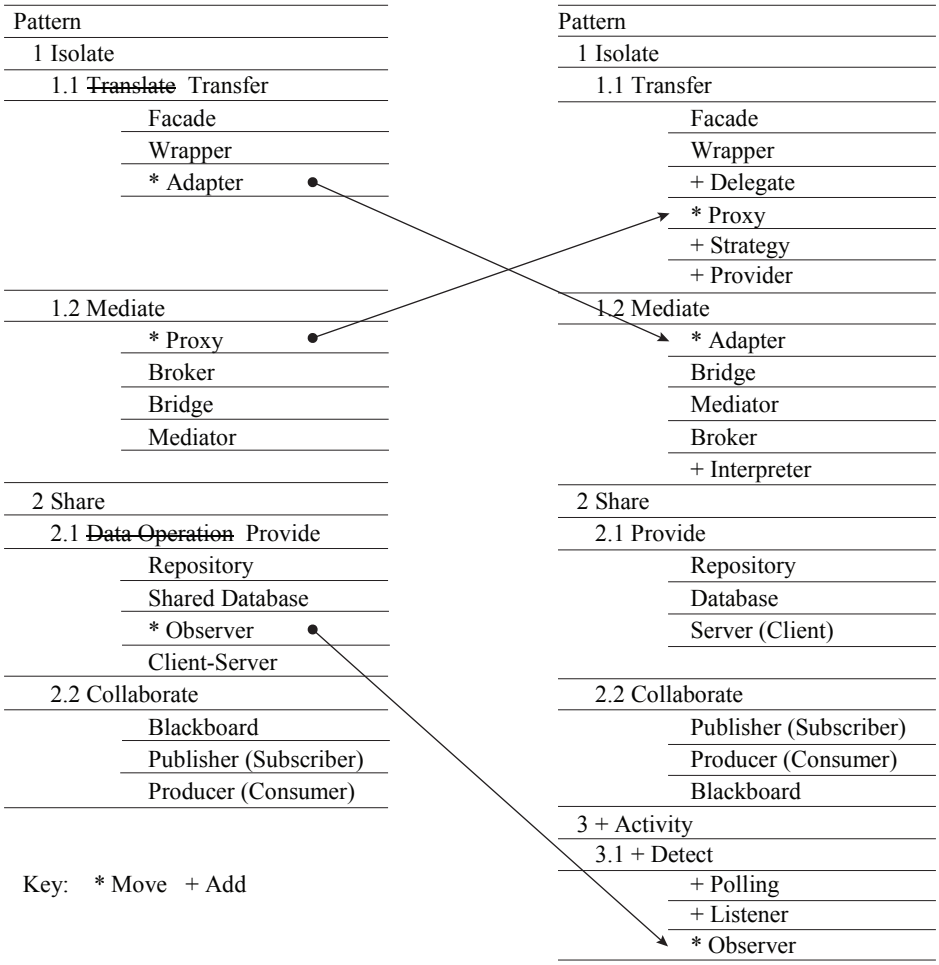


Figure 6.7: Post-study Analysis – Refactoring a pattern ontology

In the second study on pattern ontology we test whether the refactored ontology does in fact result in higher pattern-detection accuracy.

6.3.2 Errors due to Ambiguous Requirements

We must take care however not to attribute detection errors due to ambiguous requirements to the pattern ontology. Hence in the second study we test the extent to which pattern ontology

permits the accurate detection of patterns even in ambiguous requirements.

To test this we first integrate into the requirements two patterns that have a common design idea as well as one that differentiates them. Only one of these two patterns is a solution to the domain problem: the rest of the patterns are not. We expect participants to use pattern ontology to exclude all but the two patterns from consideration. The actual pattern picked should vary however because participants are likely to resolve the ambiguity present in the requirements differently.

Specifically, we place in the requirements of the second pattern-ontology study information that points to both the `Isolate/Transfer` and `Isolate/Mediate` pattern set. These two are different from the `Share` pattern set, which participants should exclude because of their detection of the `Isolate` design idea in the requirements. However participant choice of either the `Transfer` or `Mediate` design idea should vary because the requirement description permits the choice of either one to be correct.

We expect test results then to show that pattern ontology allows the accurate detection of a pattern in the requirements, but only up to the point that these become ambiguous. If so then: pattern ontology permits the accurate detection of design ideas even in ambiguous requirements; and refactoring pattern ontology should be done with caution because not all detection errors are necessarily the fault of the design ideas used to define the pattern ontology.

6.4 Study 2 – Pattern Cases

In this section we examine the results of the second pattern-case study. The treatment, requirements, and test questions used in this study are described in Appendix B.

In the first question participants were to identify all the patterns that they detected in the requirements. The results are shown in Figure 6.8.

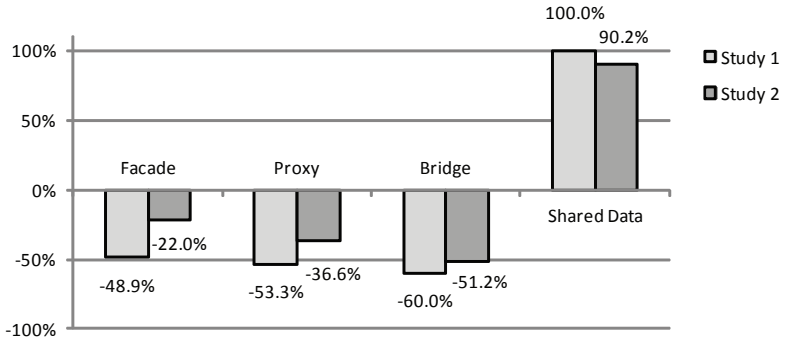


Figure 6.8: Study 2 Pattern Cases – Identifying patterns in requirements

Participants again detected patterns that were not present there: a `Façade` (22.0%), a `Proxy` (36.6%), and a `Bridge` (51.2%). Participants once again detected the `Shared Data` pattern well enough (90.2%) due to the ontologically-sound `Share` design idea.

In the second question participants were to match the requirement "application... easily switched to (another) component" to the `Wrapper` pattern. Figure 6.9 shows that participants again had difficulty differentiating between patterns when using pattern cases. Only a fifth identified the `Wrapper` pattern correctly (19.5%). The others picked `Façade` (7.3%), `Proxy` (22.0%), `Bridge` (41.4%), or `Broker` (9.8%).

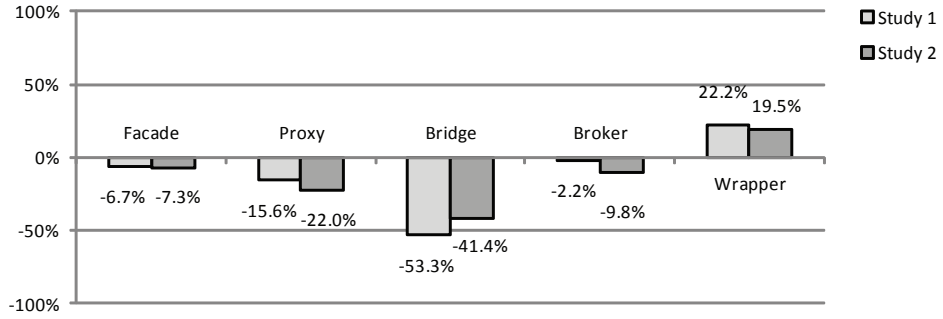


Figure 6.9: Study 2 Pattern Cases – Distinguishing the Wrapper pattern

That four-fifth of participants (80.5%) could not detect the right pattern in the require-

ments using pattern cases is significant.

In the third question we expected participants to detect the `Client-Server` pattern in the requirements. Figure 6.10 shows the test results for the two subject groups.

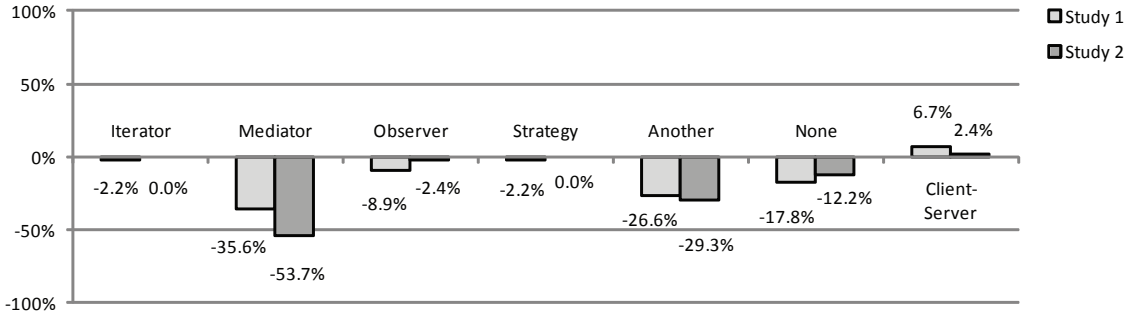


Figure 6.10: Study 2 Pattern Cases – Mapping requirements to the Client-Server pattern

Once again pattern-case descriptions did not help participants link the design ideas that define the `Client-Server` design pattern to the domain patterns present in the requirements. In this case almost all the participants (97.6%) did not detect the right pattern using the pattern-case descriptions.

6.5 Study 2 – Pattern Ontology

In this section we examine the results of the second pattern-ontology study. The treatment, requirements, and test questions used in this study are described in Appendix D.

In the first question we asked participants to identify patterns of design in the requirements using the refactored ontology. Figure 6.11 shows these results. The error rates were again much smaller when using ontology instead of cases for pattern detection: only one in ten participants had a detection error when using `Isolate/Transfer` (9.8%) and `Isolate/Mediate` (7.3%); the error rate was much higher (22.0%, 36.6%, and 51.2%) for the corresponding pattern cases.

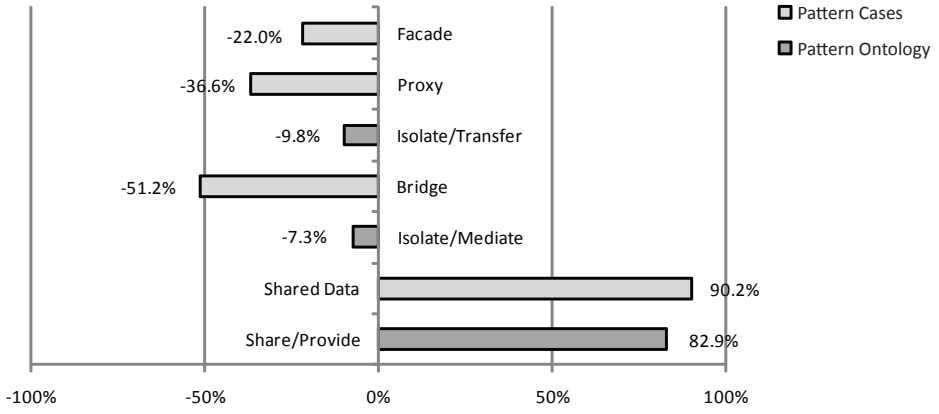


Figure 6.11: Study 2 Pattern Ontology – Identifying design ideas in requirements

In the second question we examined the effect of ambiguous requirements on pattern ontology. As shown in Figure 6.12, three out of four participants (78.0%) excluded all but the two patterns integrated into the requirements from consideration. Almost equal numbers selected the second design-pattern idea: `Transfer` (26.8%) vs. `Mediate` (36.6%).

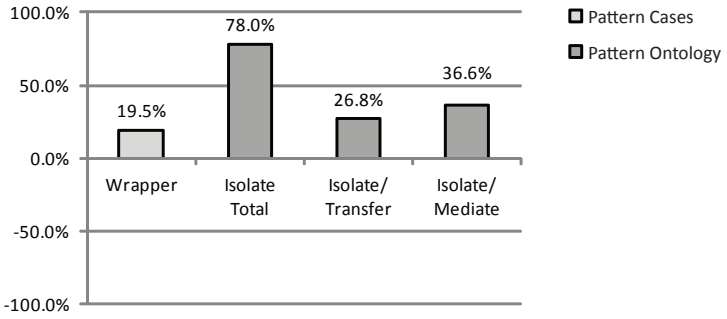


Figure 6.12: Study 2 Pattern Ontology – Accuracy in the face of requirement ambiguity

These results show that pattern ontology retains its pattern-discriminating power even in the face of requirement ambiguity: participants did detect the `Isolate` design idea in large numbers; however when it came to the second design idea (`Transfer` or `Mediate`) participant detection was mixed given the ambiguous description of the domain pattern integrated into the requirements.

In the first study the test results for the third question were inconclusive: participants were not able to use the pattern ontology to detect the `Proxy` or `Isolation/Mediation` pattern in the requirements. The pattern ontology was refactored to address these deficiencies. In particular, the `Proxy` pattern was moved to the `Isolate/Transfer` group to better differentiate it from patterns that support the adaptation of information.

In the second study participants were again asked to detect the pattern present in the requirement "rural computers make Internet requests through the central MTS computer". The test results are shown in Figure 6.13.

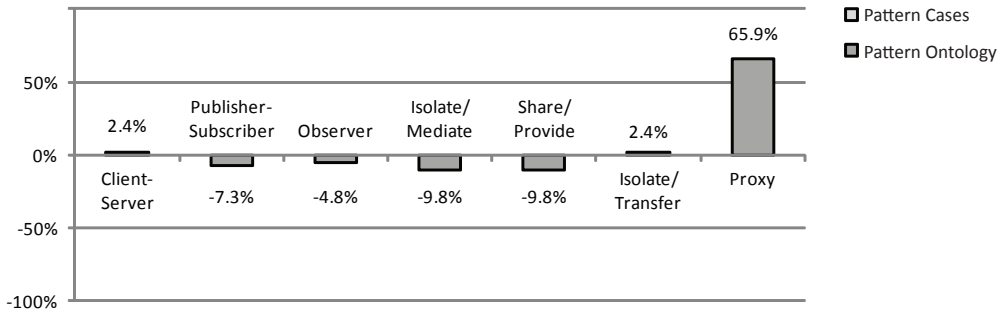


Figure 6.13: Study 2 Pattern Ontology – Distinguishing more precise patterns of design

Changing the design ideas that characterize the different design patterns seems to have increased their distinctiveness in the minds of the participants and made them easier to detect.

As a result we see low detection errors for patterns not present in the requirements: `Publisher-Subscriber` (7.3%), `Observer` (4.8%), `Isolate/Mediate` (9.8%). We also see a corresponding increase in detection accuracy for the pattern whose design ideas are in fact present there: `Proxy` (65.9%). These results are also much better than those obtained using pattern cases, i.e., the 2.4% accuracy rate obtained using the pattern-case descriptions.

As seen in Figure 6.14, the increase in pattern-detection accuracy (and corresponding decrease in pattern-detection inaccuracy) is likely due to the use in the revised pattern ontology of

design ideas with higher analogical and discriminatory power. Repeating the experiment is likely to result in a pattern ontology that supports ever-improving levels of pattern-detection accuracy for the design patterns described by this particular pattern ontology.

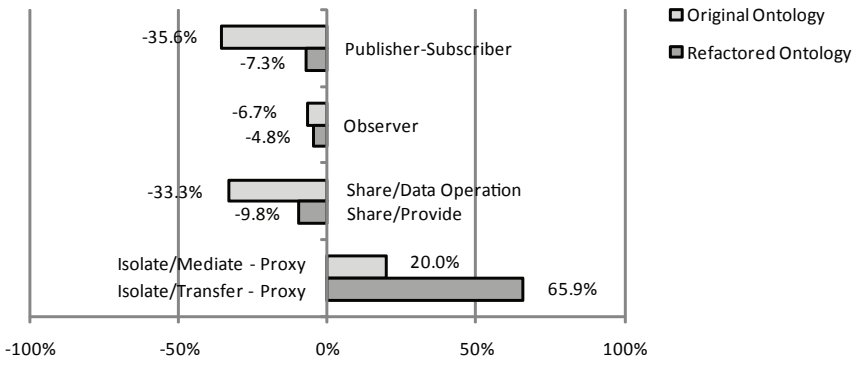


Figure 6.14: Study 2 Pattern Ontology – Increased detection accuracy due to refactoring

6.6 Summary: Study Results vis-à-vis the Hypotheses

A summary of study results in the light of the hypotheses and relevant t-tests is given in Appendix F. This summary leads us to the following conclusions about the hypotheses.

$\mathbf{H}_0$	There is no significant difference in detection accuracy when using pattern cases or when using pattern ontology to detect design ideas and patterns in a problem domain.	**Rejected**
$\mathbf{H}_1$	Using pattern cases to detect design ideas and patterns in a problem domain results in low detection accuracy.	**Supported**
$\mathbf{H}_2$	Using pattern ontology to detect design ideas and patterns in a problem domain results in higher detection accuracy than using pattern cases.	**Supported**

Our thesis then that pattern ontology is superior to pattern cases when it comes to detecting patterns of design accurately in the problem domain would appear to be substantially correct.

Chapter 7

Conclusion

In this chapter we review research findings, our theoretical model of software design, the problem with existing design-pattern descriptions and our proposed solution to it, and whether study results support our hypotheses about pattern cases and pattern ontology. We also consider threats to the validity of our findings, the contribution that this thesis makes to software-engineering research, and future directions for research in the area of pattern ontology.

7.1 Validation of the Thesis

In this thesis we examined three approaches to software engineering. The first results in a model that mirrors domain patterns but that may not have the qualities required to run efficiently on a computer. The second structures the model for run-time quality but using design patterns that may be difficult to maintain as the problem domain evolves over time. The third approach also uses design patterns, but in this case as a means to transfer domain patterns into the technical realm.

Unfortunately design patterns as currently described lead to pattern proliferation, overlap, and isolation. Both novice and experienced designers find them hard to use in transferring pattern knowledge into the technical realm. This is likely due to the current practice of describing patterns

using technical terms that are quite foreign to the concepts used to describe real-world things.

Since software engineers use natural mental mechanisms to design software, research into human cognition can provide insight into how to improve software-design activities. The findings of computer scientists working in machine pattern-recognition can also shed light on the fundamental logical rules that govern the accurate identification of specific patterns in things.

From human cognition we learn that pattern-knowledge transfer is only possible between two knowledge areas if common concepts exist between the two: the more similar the concepts the higher their analogical power. From machine pattern-recognition we learn that pattern identification is only possible if the concepts that describe a set of patterns clearly differentiate them within the pattern space: the more different the concepts the higher their discriminatory power.

In the thesis we also described a theoretical model of software design based on these findings of the cognition and computer sciences. We explained how the cognitive gap between technical terms and real-world concepts reduces their analogical power. We described how abstractive mechanisms cause technical terms to lose their discriminatory power. Since this is not the case with ontological terms, we predicted that designer accuracy in detecting patterns of design in domain knowledge would be higher when using pattern ontology than when using pattern cases.

We identified the problem with traditional design patterns, namely, that they are currently described using technical terms that lack analogical and discriminatory power. We proposed as a solution a small set of key design ideas that could differentiate the design-pattern space and also map domain knowledge to the technical realm: this is pattern ontology. Our first hypothesis was that using pattern cases to detect patterns in the problem domain would lead to low pattern-detection accuracy. Our second was that pattern ontology would lead to higher pattern-detection accuracy than if pattern cases were used.

We described the experimental method used to test these two hypotheses. Factors that might affect the internal and external validity of our two software-engineering studies were consid-

ered as well.

Finally, we discussed study results in the light of the theoretical model. The null hypothesis was rejected. Study results confirmed reports of designer difficulty in using traditional design patterns in software-design activities. Study results supported both the first and the second hypothesis, confirming our prediction that pattern ontology would be superior to pattern cases in assisting software engineers in identifying patterns of design in the problem domain.

7.2 Threats to Validity

To reduce threats to the internal validity of the studies we did the following. We chose a subject group that had sufficient knowledge of patterns to participate in the studies but not so much that study treatments would have no effect. Participants were also familiar with the problem domain considered. We gave participants sufficient motivation to give their best efforts to their work. University instructors also delivered the treatments and tests within a formal university course to guarantee the objectivity of treatment and test delivery.

Two threats to the internal validity of the studies were possible however. In the first place there was a delay of one week between pattern treatment application and testing. It is not known whether this gave participants too much or too little time to master the pattern content. Second there were no controls placed on participant learning in this time period. Subjects would have had ample opportunity to acquire additional knowledge about design patterns if they so desired.

The effect of pattern-learning time on study results is still to be considered. The second threat is unlikely to have played a significant factor in the results however. The pattern-case treatment contained all essential information about the design patterns. Learning more about them is unlikely to have negatively affected the subject group's ability to identify them in domain knowledge; if anything, the opposite is likely to be true. As for the pattern-ontology knowledge, it was

not made available to participants until presented to them for the first time. Hence in both cases the threat to the internal validity of the studies due to the treatments is likely to have been minimal.

The size of the two subject groups was also comparable to that used in other software-engineering studies reported in the literature. The focus of the study was software designers not yet familiar with using formal design patterns in design activities: thesis findings do not apply to people unable to learn design patterns or to practitioners that have already mastered their use. Hence the threats to the external validity of study results are likely to be minimal as well.

7.3 Contributions and Future Directions

This thesis makes several important contributions to the field of software-engineering research: 1. A theoretical model that sheds light on the cognition mechanisms underpinning accurate pattern detection and pattern-knowledge transfer in software design. 2. The introduction of ontological analysis to design-pattern definition. 3. The identification of several key design ideas that can efficiently describe and differentiate the design patterns currently published in the pattern catalogs. 4. Experimental studies that show that using pattern ontology results in a more accurate detection of patterns of design in requirements than using traditional design-pattern descriptions.

Much more still remains to be done in this area however. The extraction of other core design ideas from the design patterns present in the catalogs remains. The analysis of the analogical and discriminatory power of these design ideas and where they fit best in a pattern ontology would contribute greatly to our understanding of essential pattern relationships. Integrating existing pattern variants into a community-shared pattern ontology would also help reduce the proliferation, overlap, and isolation of design patterns that currently make design patterns difficult to use.

Other avenues of research are also possible. For instance, semantic-web techniques used to measure the relatedness of ideas might prove useful in measuring and comparing the analogical

and discriminatory power of design ideas. Research into how pattern-detection accuracy is affected by other types of requirement failings, for example missing information or irrelevant information (i.e., noise), is also possible. The effect on pattern-detection accuracy of the abstraction level of the concepts used to match design and domain patterns might also be a fruitful research direction.

7.4 Closing Remarks

We close with the two main findings of the thesis.

First of all, using pattern cases to detect ontological regularities in domain knowledge is likely to result in low design-pattern detection accuracy. This means that software engineers that use existing technical design-pattern descriptions to design or to redesign software models are likely to produce software that will lose its domain accuracy and even its technical quality over time.

Second, designer accuracy in identifying patterns of design in the problem domain will be higher when using a pattern ontology than when using pattern cases. This means that software engineers that use pattern ontology to identify design patterns in the problem domain are likely to produce models that in the long-term retain both domain accuracy and technical quality.

As a method pattern ontology can also help researchers identify the best design ideas to use to characterize and to differentiate design patterns. In this way too and in the long-term pattern ontology can play a major role in improving the quality of software design.

Appendix A

Participant Questionnaire

A.1 Questionnaire

Before the start of the software-engineering course, students filled in the following questionnaire.

1. I have completed the following courses (please check all that apply).

___ COMP 1010 - Intro. Computer Science I
___ COMP 1020 - Intro. Computer Science II
___ COMP 2080 - Analysis of Algorithms
___ COMP 2130 - Discrete Math for CS
___ COMP 2140 - Data Structures and Algorithms
___ COMP 2150 - Object Orientation
___ COMP 2160 - Programming Practices
___ COMP 2190 - Intro. to Scientific Computing
___ COMP 2280 - Intro to Computer Systems
___ COMP 3010 - Distributed Computing
___ COMP 3020 - Human-Computer Interaction 1
___ COMP 3030 - Automata & Formal Languages
___ COMP 3040 - Technical Communication in CS
___ COMP 3170 - Algorithms & Data Structures
___ COMP 3190 - Intro. to Artificial Intelligence
___ COMP 3350 - Software Engineering
___ COMP 3370 - Computer Organization
___ COMP 3380 - Database Concepts
___ COMP 3430 - Operating Systems
___ COMP 3490 - Computer Graphics 1
___ COMP 3620 - Professional Practice
___ COMP 3720 - Computer Networks 1

2. I have the following level of experience in analyzing user requirements for software. This includes problem-analysis experience obtained in Computer Science or in other university courses.

___ None ___ 6 to 11 months ___ 1 to 3 years ___ more than 3 years

3. I have the following level of experience in designing software. This includes solution preparation experience obtained in Computer Science or other university courses.

___ None ___ 6 to 11 months ___ 1 to 3 years ___ more than 3 years

4. I have the following level of experience in using object-oriented principles such as the relationship between a super-class and its child classes.

___ None ___ 6 to 11 months ___ 1 to 3 years ___ more than 3 years

5. I can describe or give a definition of the following design patterns (please check all that apply).

___ Façade	___ Mediator	___ Repository	___ Shared Database
___ Proxy	___ Broker	___ Client-Server	___ Publisher-Subscriber
___ Bridge	___ Wrapper	___ Observer	___ Producer-Consumer
___ Adapter	___ Singleton	___ Blackboard	___ Pool

6. Analyze the following user requirements. Answer the question that follows.

> The Department of Agriculture is revamping its computer systems. They have asked you to design a new real-time software system that can link the province's temperature, humidity, and wind monitoring stations to the central office here in Winnipeg.
>
> The sensors send their data to very old and primitive station network which the new system must still somehow use because the sensor system is too expensive to replace at this time.
>
> Department staff will perform analysis and reporting on the data collected and stored on a central office server. The system must also alert staff if sensor readings are abnormal because this might signal a defective station sensor.
>
> The new system must be designed so that additional stations can be added in the future, using network addresses that can be changed dynamically to accommodate changes in the network.
>
> They want you to use off-the-shelf software components as much as possible, but in such a way that the department can replace the purchased components with different ones in the future to meet changing department needs.

Select only one of the two answers below.

___ I do not know enough yet about finding patterns in user requirements to detect the patterns.

___ I am able to detect patterns in the user requirements above given previous experience.

7. If you selected the second answer to the question above, namely, that you are able to detect the design patterns in the user requirements above, please fill in the following section as well.

I detected the following design patterns in the user requirements above (please check all that apply).

___ Façade	___ Mediator	___ Repository	___ Shared Database
___ Proxy	___ Broker	___ Client-Server	___ Publisher-Subscriber
___ Bridge	___ Wrapper	___ Observer	___ Producer-Consumer
___ Adapter	___ Singleton	___ Blackboard	___ Pool

A.2 Questionnaire Results

The questionnaire results for the 86 students participating in the two studies follow.

1.	Previous course work	Computer Science I & II, Analysis of Algorithms, Discrete Math, Data Structures & Algorithms, Object Orientation, Computer Systems (> 90%)
		Computer Organization, Database Concepts (80-90%)
2.	Experience analyzing requirements	None (24%), 6 to 11 months (48%), 1 to 3 years (21%), > 3 years (7%)
3.	Experience designing software	None (4%), 6 to 11 months (31%), 1 to 3 years (42%), > 3 years (25%)
4.	Experience in object-oriented analysis	None (1%), 6 to 11 months (18%), 1 to 3 years (55%), > 3 years (27%)
5.	Can describe or define design patterns	Wrapper and Client-Server (> 85%); Proxy (69%), Façade, Adapter, Singleton, Repository and Shared-Database (40-60%); All other patterns (< 20%)
6.	Claims of proficiency in detecting patterns in user requirements	53% of the 86 students
7.	Accuracy of pattern-detection of students claiming proficiency	Some detection: Adapter (60%), Shared-Database (65%), Client-Server (82%)
		Poor detection: Proxy, Bridge, Repository, Observer (< 25%)
		Detected pattern not actually in requirements: Wrapper (38%), Publisher-Subscriber (24%), Façade (20%)

Table A.1: Questionnaire results for potential study participants

A.3 Summary of Results

In the results above we see that the students in the course do have experience analyzing requirements and designing software. Their course work also gives them some experience with simple patterns and domains but little with design patterns per se.

About half of the group claims proficiency with design patterns. In a test however these students demonstrate a lack of proficiency in using most of them. Hence all things considered the study group is a good fit for the studies on pattern cases and pattern ontology.

A.4 Patterns Present in the Requirements

The design patterns present in the requirements used in the questionnaire pattern-detection test are shown below.

Design pattern	Requirement sentences
Client-Server	Need real-time software system that can link the province's temperature, humidity, and wind monitoring stations to the central office.
Bridge	The sensors send their data to a very old and primitive station network which the new system must somehow use.
Shared Database	Department staff will perform analysis and reporting on the data collected and stored on a central office server.
Observer	The system must also alert staff if sensor readings are abnormal because this might signal a defective station sensor.
Proxy	The system must be designed so that additional stations can be added in the future, using network addresses that can be changed dynamically.
Wrapper	Use off-the-shelf software components but in such a way that the department can replace the purchased components with different ones in the future.

Appendix B

Study 1 and Study 2 – Pattern Cases

The descriptions of the design patterns used in the pattern-case study are based on those of the original Gang of Four (Gamma et al., 1995). The pattern descriptions for the other design patterns used in the study mirror the example given below.

B.1 Treatment

The software-engineering course materials (Mišić, 2009) used to teach the Proxy pattern to the subject group follow.

(1) The Proxy Pattern (Gang of Four-style)

- Context: often, it is time-consuming and complicated to create instances of a class (heavyweight classes)
- There is a time delay and a complex mechanism involved in creating the object in memory
- Problem: how to reduce the need to create instances of a heavyweight class?
- Forces: we want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities
- It is also important for many objects to persist from run to run of the same program

(2) The Proxy Pattern (continued)

Stand-ins for object may be needed because the real object

- Is not locally available;
- Is expensive to create; or
- Needs protected access for security or safety.

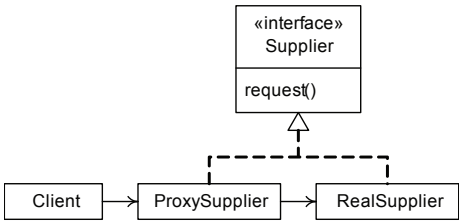
The stand-in must

- Have the same interface as the real object;
- Handle as many messages as it can;
- Delegate messages to the real object when necessary.

Analogy: a stand-in or proxy

(3) Proxy Pattern Structure

Diagram

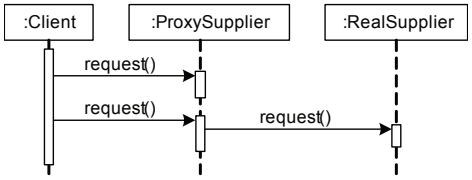


Elements and relations

- Entities: Client, ProxySupplier, RealSupplier, common interface of Supplier
- Client request to ProxySupplier
- ProxySupplier and RealSupplier share same Supplier interface

(4) Proxy Pattern Behavior

Diagram

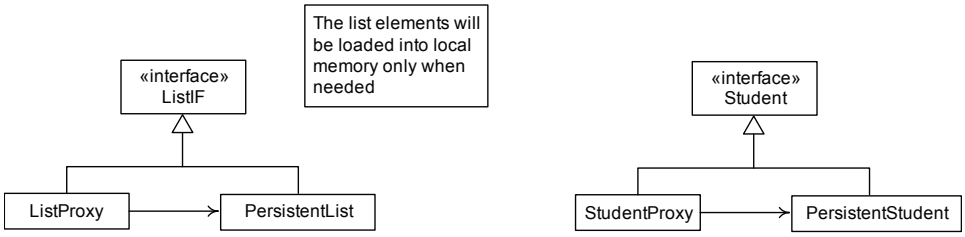


Behavior

- Client requests an object from the ProxySupplier
- The first request results in the creation of the RealSupplier object and the subsequent use of this object by the Client via the ProxySupplier. The common ProxySupplier and RealSupplier interface (Supplier) permits a transparent transfer of the Client request to the RealSupplier object.
- Subsequent requests result in the use by the Client of the existing object RealSupplier via the ProxySupplier.

(5) Example Proxies

Diagram



Description - ListProxy

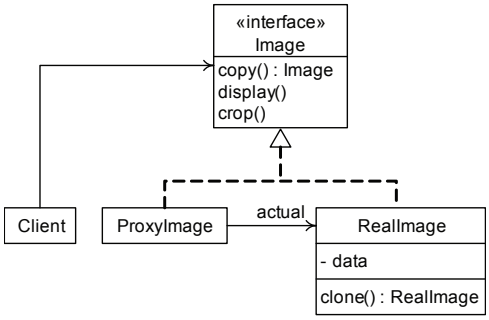
- ListProxy and PersistenList share the same ListIF interface.
- Client call to ListProxy loads the list elements the first time or as needed.
- ListProxy calls PersistentList on behalf of Client and completes request for list elements.

Description - StudentProxy

- StudentProxy and PersistentStudent share the same Student interface.
- Client request to StudentProxy (e.g. parent) results in involvement of actual student as needed.
- Client makes requests to actual PersistentStudent via StudentProxy proxy.

(6) Example Proxies (continued)

Diagram



Description

- Client uses common interface Image shared by ProxyImage and RealImage to access what it assumes is a RealImage.
- On Client request, ProxyImage makes actual request calls to RealImage whose results are then returned to the Client by the ProxyImage

(7) When to Use Proxies

Use the Proxy pattern whenever the services provided by a supplier need to be mediated or managed in some way without disturbing the supplier interface. Kinds of proxies:

- Virtual proxy – Delay the creation or loading of large or computationally expensive objects
- Remote proxy – Hide the fact that an object is not local
- Protection proxy – Ensure that only authorized clients access a supplier in legitimate ways

B.2 Test

There are three steps to preparing a test for a pattern-case study. We first choose the design patterns to test and describe them using domain examples and terminology. We then prepare a requirement set that inserts the domain patterns into the context of the problem domain. Finally we prepare test questions that elicit the detection of the targeted design patterns in the requirement set.

B.2.1 Integration of Patterns into Requirements

In this step we select the design patterns to test and describe them using domain examples.

Pattern properties	Requirement sentences
Entities request service from single entity. (Client-Server)	Each store is connected to the central Winnipeg office computer for all credit card transactions.
Central data accessed via protocols. (Shared Data)	Store managers will run reports on inventory in the computer system.
Transfer to external component. (Wrapper)	Interact with existing Exchange application using Office... must be easily switched to an open-source email component in the future.

B.2.2 Requirement Set Preparation

In this step we insert the domain examples into the context of a problem to solve in the domain.

> Safeway hires you to design a new computer system to run their grocery stores. In the new system: staff will increase stock counts whenever there is an inventory delivery to a store; cash registers will decrease stock counts when there is a sale to a customer; store managers will run reports on inventory counts in the computer system; and each store is connected to the central Winnipeg office computer for all credit card transactions.
>
> Safeway wants the new system to interact with an existing Microsoft Exchange application using Office protocols to cut costs. However, the company also wants you to design the system so that it can easily switch to using an open-source email component in the future.

B.2.3 Test Questions

In this step we ask participants to detect the targeted design patterns in the requirement set.

1. The following pattern(s) are present in the user requirements above [check all that you detect].

- □ Façade
- □ Shared Data (or Shared Database)
- □ Proxy
- □ Bridge
- □ None of the above

2. Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern.

- □ Façade
- □ Proxy
- □ Wrapper
- □ Bridge
- □ Broker

3. Having store computers send credit card requests to the central Winnipeg computer for processing is an example of the _____ pattern.

- □ Iterator
- □ Mediator
- □ Observer
- □ Strategy
- □ Another pattern: _____
- □ None of the patterns above is discernible in these requirements

B.3 Evaluation of Participant Results

We evaluate the accuracy of participant design-pattern detection by comparing subject answers to the correct answers. Using a multiple-choice format for the questions ensures that the participant compares each design pattern to the domain pattern present in the requirements before making a decision as to which is the best match.

1. The following pattern(s) are present in the user requirements above [check all that you detect].

□ Façade	No. There is no isolation here of internal components.
□ Shared Data (or Shared Database)	Yes. Stores connect to a central server that provides common data.
□ Proxy	No. There is no isolation or representation of another entity here.
□ Bridge	No. There is no cross-over of function here.
□ None of the above	No. There is a discernible pattern here.

2. Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern.

□ Façade	No. Façade is for a component internal to the system.
□ Proxy	No. The isolating entity does not represent the email component.
□ Wrapper	Yes. Switching the external email component requires a wrapper.
□ Bridge	No. There is no cross-over of function here.
□ Broker	No. There is no mediation of function here.

3. Having store computers send credit card requests to the central Winnipeg computer for processing is an example of the _____ pattern.

□ Iterator	No. There is no repetitive function here.
□ Mediator	No. No other entity is mentioned so this is a service only.
□ Observer	No. The detection of events is not needed here.
□ Strategy	No. There is no swapping of function here.
□ Another pattern: _____	Yes. Client-Server because of the service provided. Shared Data while plausible is not valid because the request is for transactional processing and not for access to common data.
□ None of the patterns above is discernible in these requirements	No. There is a pattern discernible in the store-server interaction.

Appendix C

Study 1 – Pattern Ontology

The design ideas used to define the pattern ontology come from the design-pattern descriptions of the original Gang of Four (Gamma et al., 1995). A description of the materials used in the first pattern-ontology study follows.

C.1 Treatment

The software-engineering course materials (Mišić, 2009) used to teach pattern-ontology design ideas to the subject group follow.

(1) What's wrong with patterns?

- Patterns are good but: there is a non-negligible risk that patterns will be misused (or misapplied) because they are misunderstood
- Now, how can a pattern be misunderstood? Does that happen because
  - We don't read the pattern carefully?
  - We deliberately ignore the description of the pattern?
  - We don't understand the problem well?
  - The description of the pattern does not lend itself to easy application to a real problem?
- Let us focus on this last... pattern... and see why the current pattern application wisdom may fail to deliver (the purported improvement)

(2) An Example

> Safeway hires you to design a new computer system to run their grocery stores. In the new system: staff will increase stock counts whenever there is an inventory delivery to a store; cash registers will decrease stock counts when there is a sale to a customer; store managers will run reports on inventory counts in the computer system; and each store is connected to the central Winnipeg office computer for all credit card transactions.
>
> Safeway wants the new system to interact with an existing Microsoft Exchange application using Office protocols to cut costs. However, the company also wants you to design the system so that it can easily switch to using an open-source email component in the future.

(3) And one of the questions

Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern.

- □ Façade
- □ Proxy
- □ Wrapper
- □ Bridge
- □ Broker

(4) The correct answer is...

- Adapter or Wrapper, which is described as the pattern that "converts the interface of a class into another interface clients expect"
- But one might have been tempted to answer otherwise, for example:
  - Bridge, because it "decouples an abstraction from its implementation so that the two can vary independently"
  - Façade, which "defines a higher-level interface that makes the subsystem easier to use"
  - Or perhaps even a Proxy, which "provides a surrogate or placeholder for another object to control access to it"
- All of which do seem similar enough
- So, what does that say about our problem?

(5) The problem is...

- The standard definition of design patterns makes them highly contextualized - the correct pattern to apply is very much dependent on the context of the problem
- While this is the essence of the pattern-based design, we must acknowledge that the context of the pattern is not easy to identify from the requirements specification which describes the problem from a viewpoint quite different from the designer's one
- As a result, several patterns may have quite similar descriptions of their respective contexts
- Which makes it quite easy (and, thus, likely) to MISINTERPRET the context of the real requirements
- And, by extension, lead the designer to apply a different and inappropriate pattern to the problem at hand

(6) In other words

- Current approaches to pattern-based design necessitates the use of highly contextualized pattern descriptions
- Which results in designers having difficulty in selecting the correct pattern to apply to their own set of requirements
- But this may also (and quite frequently) result in an inability to apply the correct pattern in a different domain (which is characterized by a different context)
- And may even result in the need to modify the pattern to suit the problem, but modify it in such a way that its power to implement the specifications is lost

(7) So, is there a better way?

- A different approach relies on the familiar paradigm of abstraction: it begins with a classification of patterns which is more abstract but, arguably, easier to apply
- Such patterns will be referred to as generic patterns or super-patterns
- Regular patterns are but special cases of their parent super-patterns, distinguished by the dependence on a specific context
- In other words, we talk about pattern inheritance, much in the spirit of the object-oriented paradigm

(8) Example: Wrapper and Façade

- Wrapper, as defined before, "converts the interface of a class into another interface clients expect" - in other words, it exposes methods which effectively map the interface of a component to a known API
- It applies to a component domain (can't wrap a data tier, can you?)
- Façade, as defined before, "defines a higher-level interface that makes the subsystem easier to use" - it exposes methods from various objects in a lower tier
- But this is applicable at the architecture level - it makes no sense to provide a façade to a component, right?

(9) Now, to find generic patterns

- We need to find the commonalities as well as differences between the two
- So, what would be the main FUNCTION of both Wrapper and Façade? What do they do, in most abstract terms?
- The common theme: they provide ISOLATION between different tiers/components
- But note that the concept of isolation is quite abstract and, thus, context-independent
- Now, the difference between the two is provided by the CONTEXT in which isolation is required
- In fact, there is more variation in that department...

(10) Basic classification

Pattern		Pattern Specialization
1 Isolate		
1.1 Translate		
	Facade	To sub-component
	Wrapper	To integrated external component
	Adapter	Using component communication strategy
1.2 Mediate		
	Proxy	To available provider
	Broker	Using provider communication protocol
	Bridge	Using provider communication strategy
	Mediator	Between multiple clients and/or providers
2 Share		
2.1 Data Operation		
	Repository	Aggregation from multiple providers
	Shared Database	Requests made using provider protocols
	Observer	Response based on data change detection
	Client-Server	Service response to multiple requestors
2.2 Collaborate		
	Blackboard	Response based on cooperative heuristics
	Publisher (Subscriber)	Explicit coordination of data broadcast
	Producer (Consumer)	Synchronization of providers and requestors

(11) Example

The Department of Agriculture is revamping its computer systems. They have asked you to design a new real-time software system that can link the province's temperature, humidity, and wind monitoring stations to the central office here in Winnipeg. The sensors send their data to very old and primitive station network which the new system must still somehow use because the sensor system is too expensive to replace at this time.

Department staff will perform analysis and reporting on the data collected and stored on central office servers. The system must also alert staff if sensor readings are abnormal because this might signal a defective station sensor. The new system must be designed so that additional stations can be added in the future, using network addresses that can be changed dynamically to accommodate changes in the network.

They want you to use off-the-shelf software components as much as possible, but in such a way that the department can replace the purchased components with different ones in the future to meet changing department needs.

(12) Let's detect Generic patterns

- Real-time software system links monitoring stations to the central office: collaboration
- The new system must still use the old and primitive station network: translation
- Data is collected and stored on servers: sharing
- Staff perform analysis and reporting: sharing
- System alerts staff of abnormal readings: communication
- Designed so additional stations can be added in the future: mediation
- Replace system components components: translation

C.2 Test

There are three steps to preparing a test for a pattern-ontology study. We first choose the patterns to test and describe them using domain examples and terminology. We then prepare a requirement set that inserts the pattern of design into the context of the problem domain. Finally we prepare test questions that elicit the detection of the targeted patterns in the requirement set.

C.2.1 Integration of Patterns into Requirements

In this step we select the patterns to test and describe them using domain examples.

Pattern properties	Requirement sentences
Share/Data operation (Client Server)	Rural computers will connect via satellite to a central MTS computer in Winnipeg… that communicates with the Internet.
Isolate/Mediate (Proxy)	Customers… communicate with the Internet using a wide variety of different protocols… the central MTS computer communicates with the Internet in one way only…
Share/Data operation (Shared-database)	store all customer transactions so that the company can bill customers
Isolate/Translate (Wrapper)	connection software… easily switched to a better technology

C.2.2 Requirement Set Preparation

In this step we insert the domain examples into the context of a problem to solve in the domain.

> MTS hires you to design a new computer system to service rural customers. Rural computers will connect via satellite to a central MTS computer in Winnipeg. Customers may be using a Mac, Windows or Linux operating system that communicates with the Internet using a wide variety of different protocols. However, the central MTS computer communicates with the Internet in one way only, so the new system must deal with this problem. The system must store all customer transactions so that the company can bill customers based on how much bandwidth they used while connected to the Internet.
>
> The company also plans to purchase an All Stream subsidiary that builds components for optimal satellite data transmissions. Consequently, they want you to design the MTS connection software installed on customer machines so that it can be easily switched to a better technology should it becomes available in the future.

C.2.3 Test Questions

In this step we ask participants to detect the targeted patterns of design in the requirement set.

1. The following generic pattern / sub-pattern is not present in the user requirements above.
   - □ Isolation / Mediation
   - □ Isolation / Translation
   - □ Share/ Data Operation
   - □ Share / Collaboration

2. Designing the system so that the MTS connection component can be switched easily in the future to the component of the All Stream subsidiary is an example of the _____ generic pattern (or generic pattern/ sub-pattern pair).
   - □ Isolation
   - □ Isolation / Mediation
   - □ Isolation / Translation
   - □ Share
   - □ None of the above

3. Having rural computers connect to the central MTS computer for Internet browsing is an example of the _____ pattern.
   - □ Blackboard
   - □ Observer
   - □ Publisher-Subscriber
   - □ Share / Data Operation
   - □ None of the above

C.3 Evaluation of Participant Results

We evaluate the accuracy of participant design-pattern detection by comparing subject answers to the correct answers. Using a multiple-choice format for the questions ensures that the participant compares each pattern to the domain pattern present in the requirements before making a decision as to which is the best match.

1. The following generic pattern / sub-pattern is not present in the user requirements above.

□ Isolation / Mediation	No. There is a conversion of multiple protocols to one protocol.
□ Isolation / Translation	No. There is a need to switch to new technology.
□ Share / Data Operation	No. Rural computers connect to the central one for Internet service.
□ Share / Collaboration	Yes. There is no entity actively mediating the share.

2. Designing the system so that the MTS connection component can be switched easily in the future to the component of the All Stream subsidiary is an example of the _____ generic pattern (or generic pattern/ sub-pattern pair).

□ Isolation	Yes. The entity isolates. However, Isolate/Translate is better.
□ Isolation / Mediation	No. No adaptation of information is needed in a switch.
□ Isolation / Translation	Yes. The entity both isolates and transfers data.
□ Share	No. There is no client sharing of data here.
□ None of the above	No. There is a detectible pattern in these requirements.

3. Having rural computers connect to the central MTS computer for Internet browsing is an example of the _____ pattern.

□ Blackboard	No. There is no mention of cooperative heuristics.
□ Observer	No. No entity is monitoring a data change.
□ Publisher-Subscriber	No. There is no mention of coordination of data.
□ Share / Data Operation	No. There is no data store provision here.
□ None of the above	Yes. There is no Isolate/Mediate pattern in the list above.

Appendix D

Study 2 – Pattern Ontology

The pattern ontology used in the second study is a refactored version of the one used in the first. A description of the materials used in the second pattern-ontology study follows.

D.1 Treatment

The software-engineering course materials (Mišić, 2009) used to teach pattern-ontology design ideas to the subject group follow.

(1) What's wrong with patterns?

- Patterns are good but: there is a non-negligible risk that patterns will be misused (or misapplied) because they are misunderstood
- Now, how can a pattern be misunderstood? Does that happen because
  - We don't read the pattern carefully?
  - We deliberately ignore the description of the pattern?
  - We don't understand the problem well?
  - The description of the pattern does not lend itself to easy application to a real problem?
- Let us focus on this last... pattern... and see why the current pattern application wisdom may fail to deliver (the purported improvement)

(2) An Example

> Safeway hires you to design a new computer system to run their grocery stores. In the new system: staff will increase stock counts whenever there is an inventory delivery to a store; cash registers will decrease stock counts when there is a sale to a customer; store managers will run reports on inventory counts in the computer system; and each store is connected to the central Winnipeg office computer for all credit card transactions.
>
> Safeway wants the new system to interact with an existing Microsoft Exchange application using Office protocols to cut costs. However, the company also wants you to design the system so that it can easily switch to using an open-source email component in the future.

(3) And one of the questions

Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern.

- □ Façade
- □ Proxy
- □ Wrapper
- □ Bridge
- □ Broker

(4) The correct answer is...

- Adapter or Wrapper, which is described as the pattern that "converts the interface of a class into another interface clients expect"
- But one might have been tempted to answer otherwise, for example:
  - Bridge, because it "decouples an abstraction from its implementation so that the two can vary independently"
  - Façade, which "defines a higher-level interface that makes the subsystem easier to use"
  - Or perhaps even a Proxy, which "provides a surrogate or placeholder for another object to control access to it"
- All of which do seem similar enough
- So, what does that say about our problem?

(5) The problem is...

- The standard definition of design patterns makes them highly contextualized - the correct pattern to apply is very much dependent on the context of the problem
- While this is the essence of the pattern-based design, we must acknowledge that the context of the pattern is not easy to identify from the requirements specification which describes the problem from a viewpoint quite different from the designer's one
- As a result, several patterns may have quite similar descriptions of their respective contexts
- Which makes it quite easy (and, thus, likely) to MISINTERPRET the context of the real requirements
- And, by extension, lead the designer to apply a different and inappropriate pattern to the problem at hand

(6) In other words

- Current approaches to pattern-based design necessitates the use of highly contextualized pattern descriptions
- Which results in designers having difficulty in selecting the correct pattern to apply to their own set of requirements
- But this may also (and quite frequently) result in an inability to apply the correct pattern in a different domain (which is characterized by a different context)
- And may even result in the need to modify the pattern to suit the problem, but modify it in such a way that its power to implement the specifications is lost

(7) So, is there a better way?

- A different approach relies on the familiar paradigm of abstraction: it begins with a classification of patterns which is more abstract but, arguably, easier to apply
- Such patterns will be referred to as generic patterns or super-patterns
- Regular patterns are but special cases of their parent super-patterns, distinguished by the dependence on a specific context
- In other words, we talk about pattern inheritance, much in the spirit of the object-oriented paradigm

(8) Example: Wrapper and Façade

- Wrapper, as defined before, "converts the interface of a class into another interface clients expect" - in other words, it exposes methods which effectively map the interface of a component to a known API
- It applies to a component domain (can't wrap a data tier, can you?)
- Façade, as defined before, "defines a higher-level interface that makes the subsystem easier to use" - it exposes methods from various objects in a lower tier
- But this is applicable at the architecture level - it makes no sense to provide a façade to a component, right?

(9) Now, to find generic patterns

- We need to find the commonalities as well as differences between the two
- So, what would be the main FUNCTION of both Wrapper and Façade? What do they do, in most abstract terms?
- The common theme: they provide ISOLATION between different tiers/components
- But note that the concept of isolation is quite abstract and, thus, context-independent
- Now, the difference between the two is provided by the CONTEXT in which isolation is required
- In fact, there is more variation in that department...

(10) Generics classification

- The table that follows presents some basic generic patterns
- Followed by a sub-classification based on some known variants (but which are still generic, i.e., context-independent, to a large degree)
- Until we reach the contextual patterns we are familiar with, patterns which specializes the generic set in a specific, concrete way
- Note that we provide the reason for including the contextual pattern in a given set the specialization is simply the contextual distinction that makes the pattern different from any other in the set (and which justifies our giving it a name, such as Wrapper or Façade)

(11) Generics classification (continued)

Pattern		Pattern Specialization
1 Isolate		
1.1 Transfer		
	Facade	Isolate components internal to the system
	Wrapper	Isolate component external to the system
	Delegate	Represent a component internal to the system
	Proxy	Represent a component external to the system
	Strategy	Plug an exchangeable operation into the system
	Provider	Plug an exchangeable component into the system
1.2 Mediate		
	Adapter	Map external component interface to system interface
	Bridge	Map internal component interface to system interface
	Mediator	Map interface for bi-directional interaction
	Broker	Bi-directional rule-based component and system interaction
	Interpreter	Translate component interactions with system interface
2 Share		
2.1 Provide		
	Repository	Aggregate the data of multiple sources
	Database	Communicate data using a protocol
	Server (Client)	Provide data or data-transformation services to clients
2.2 Collaborate		
	Publisher (Subscriber)	Registration controls component interactions
	Producer (Consumer)	Semaphores control component interactions
	Blackboard	Heuristic rules control component interactions
3 Activity		
3.1 Detect		
	Polling	Monitor component change at periodic intervals
	Listener	Monitor state change in a component in real-time
	Observer	Monitor event change in a component in real-time

(12) Example

The Department of Agriculture is revamping its computer systems. They have asked you to design a new real-time software system that can link the province's temperature, humidity, and wind monitoring stations to the central office here in Winnipeg. The sensors send their data to very old and primitive station network which the new system must still somehow use because the sensor system is too expensive to replace at this time.

Department staff will perform analysis and reporting on the data collected and stored on central office servers. The system must also alert staff if sensor readings are abnormal because this might signal a defective station sensor. The new system must be designed so that additional stations can be added in the future, using network addresses that can be changed dynamically to accommodate changes in the network.

They want you to use off-the-shelf software components as much as possible, but in such a way that the department can replace the purchased components with different ones in the future to meet changing department needs.

(13) Walkthrough: the Isolate-Transfer branch

- Isolate: a component that separates entities
  - Includes the simple passing of data (Transfer) and a more complex mapping of data between interfaces (Mediate)
- Isolate-Transfer: a component that isolates entities and passes data between them
  - Façade: internal components are isolated
  - Wrapper: external shrink-wrap component is isolated
  - Delegate: component represents another internal to system
  - Proxy: component represents another external to system
  - Strategy: plug in an exchangeable operation
  - Provider: plug in an exchangeable component

(14) Let's detect Generic patterns

- Real-time software system links monitoring stations to the central office: Share → Provide
- The new system must still use the old and primitive station network: Isolate → Mediate
- Data is collected and stored on servers: Share → Provide
- Staff perform analysis and reporting: Share → Provide
- System alerts staff of abnormal readings: Activity → Detect
- Designed so additional stations can be added in the future: Isolate → Transfer
- Replace system components components: Isolate → Transfer

D.2 Test

There are three steps to preparing a test for a pattern-ontology study. We first choose the patterns to test and describe them using domain examples and terminology. We then prepare a requirement set that inserts the pattern of design into the context of the problem domain. Finally we prepare test questions that elicit the detection of the targeted patterns in the requirement set.

D.2.1 Integration of Patterns into Requirements

In this step we select the patterns to test and describe them using domain examples.

Pattern properties	Requirement sentences
Share/Provide (Client-Server)	Rural computers will connect via satellite to a central MTS computer in Winnipeg.
Isolate/Transfer (Proxy)	…MTS computer system used to service rural customers. Rural computer connect via satellite to a central MTS computer in Winnipeg. Customers… access the Internet using a wide variety of protocols. … convert to a standardized MTS-defined 'usage protocol' to ensure that the company has a consistent view of bandwidth use by rural customers.
Share/Provide (Database)	store all customer transactions so that the company can bill customers
Isolate/Transfer (Wrapper) or Isolate/Mediate (Adapter)	design will accommodate the new All Stream components; … easy given similar technologies and business practices of the companies

D.2.2 Requirement Set Preparation

In this step we insert the domain examples into the context of a problem to solve in the domain.

> MTS hires you to prepare an update to the MTS computer system used to service rural customers. Rural computers connect via satellite to a central MTS computer in Winnipeg. Customers using the Mac, Windows or Linux operating system access the Internet using a wide variety of different protocols. The system must store all customer transactions so that the company can bill customers based on how much bandwidth they use while connected to the Internet. Bandwidth-use reports must convert each protocol transaction to a standardized and measurable MTS-defined 'usage protocol' to ensure that the company has a consistent and accurate view of bandwidth use by rural customers.
>
> The company has also recently purchased All Stream, a company that builds components that optimize satellite data transmissions. They want you to ensure that your design will accommodate the new All Stream components; they expect this should be easy given the similar technologies and business practices of the two companies.

D.2.3 Test Questions

In this step we ask participants to detect the targeted patterns of design in the requirement set.

1. The following generic pattern / sub-pattern is <u>not</u> present in the user requirements above.
   - □ Isolate / Mediate
   - □ Isolate / Transfer
   - □ Share/ Provide
   - □ Share / Collaborate

2. Designing the system so that the All Stream transmission component can be used by the updated system is an example of the _____ generic pattern (or generic pattern/ sub-pattern pair).

- □ Isolate
- □ Isolate / Mediate
- □ Isolate / Transfer
- □ Share
- □ None of the above

3. Having rural computers make Internet requests through the central MTS computer is an example of the _____ pattern.

- □ Publisher-Subscriber
- □ Observer
- □ Proxy
- □ Share / Provide
- □ Isolate / Transfer
- □ Isolate / Mediate

D.3 Evaluation of Participant Results

We evaluate the accuracy of participant design-pattern detection by comparing subject answers to the correct answers. Using a multiple-choice format for the questions ensures that the participant compares each pattern to the domain pattern present in the requirements before making a decision as to which is the best match.

1. The following generic pattern / sub-pattern is <u>not</u> present in the user requirements above.

□ Isolate / Mediate	Yes. Using the All Stream component may require adaptation.
□ Isolate / Transfer	Yes. There is the wrapping of the All Stream component.
□ Share/ Provide	Yes. There is reporting on stored data.
□ Share / Collaborate	No. There is no entity controlling data-sharing interactions.

2. Designing the system so that the All Stream transmission component can be used by the updated system is an example of the _____ generic pattern (or generic pattern/ sub-pattern pair).

□ Isolate	Yes. Isolation is present. However, Isolate/Mediate or Isolate/Transfer is a better choice.
□ Isolate / Mediate	Yes. If the "accommodation" is perceived as not being easy given the need to integrate two different companies.
□ Isolate / Transfer	Yes. If the expectation that the "accommodation" will be as easy as expected is perceived as being realizable.
□ Share	No. There is no sharing of centralized data here.
□ None of the above	No. There is a detectable pattern here.

3. Having rural computers make Internet requests through the central MTS computer is an example of the _____ pattern.

□ Publisher-Subscriber	No. There is no shared-data collaboration here.
□ Observer	No. There is no detection of an event-change activity here.
□ Proxy	Yes. The central MTS server isolates and transfers data, but also represents the rural computers attached through it to the Internet.
□ Share / Provide	No. In this function of the server, it routes but does not store data.
□ Isolate / Transfer	Yes. The routing is simple with no real modification of the data. However, Proxy is better given its representation property.
□ Isolate / Mediate	No. The server isolates but does not adapt the data. Switching protocols does not warrant the term mediation.

Appendix E

Study Test Results

The test results for the first study of pattern cases are given in Table E.1.

Question	Pattern Properties	Detection (n=45)	
		Accurate	Inaccurate
1. The following pattern(s) are present in the user requirements. Answer: Shared Data	Façade	51.1%	(48.9%)
	Shared Data	100.0%	0.0%
	Proxy	46.7%	(53.3%)
	Bridge	40.0%	(60.0%)
	None of the above	100.0%	0.0%
2. Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern. Answer: Wrapper	Facade		(6.7%)
	Proxy		(15.6%)
	Wrapper	22.2%	
	Bridge		(53.3%)
	Broker		(2.2%)
		22.2%	(77.8%)
3. Having store computers send credit card requests to the central Winnipeg computer for processing is an example of the _____ pattern. Answer: Client-Server	Iterator		(2.2%)
	Mediator		(35.6%)
	Observer		(8.9%)
	Strategy		(2.2%)
	Another	6.7%	(26.6%)
	None of the above		(17.8%)
		6.7%	(93.3%)

Table E.1: Study 1 Pattern Cases – Test results

The test results for the first study of pattern ontology are given in Table E.2.

Question	Pattern Properties	Detection (n=45)	
		Accurate	Inaccurate
1. The following generic pattern/ sub-pattern is not present in the user requirements. Answer: Share/Collaboration	Isolation/Mediation	71.1%	(28.9%)
	Isolation/Translation	95.6%	(4.4%)
	Share/Data Operation	91.1%	(8.9%)
	Share/Collaboration	57.8%	(42.2%)
2. Designing the system so that the MTS connection component can be switched easily in the future to the component of the All Stream subsidiary is an example of the _____ generic pattern. Answer: #1 Isolation/Translation #2 Isolation	Isolation	13.3%	
	Isolation/Mediation		(13.3%)
	Isolation/Translation	68.9%	
	Share		0.0%
	None of the above		(4.5%)
		82.2%	(17.8%)
3. Having rural computers connect to the central MTS computer for Internet browsing is an example of the _____ pattern. Answer: Isolate/Mediate	Blackboard		(4.4%)
	Observer		(6.7%)
	Publisher-Subscriber		(35.6%)
	Share/Data Operation		(33.3%)
	None of the above	20.0%	
		20.0%	(80.0%)

Table E.2: Study 1 Pattern Ontology – Test results

The test results for the second study of pattern cases are given in Table E.3.

Question	Pattern Properties	Detection (n=41)	
		Accurate	Inaccurate
1. The following pattern(s) are present in the user requirements. Answer: Shared Data	Façade	78.0%	(22.0%)
	Shared Data	90.2%	(9.8%)
	Proxy	63.4	(36.6%)
	Bridge	48.8	(51.2%)
	None of the above	97.6	(2.4%)
2. Designing the system so that the email component can be switched easily in the future to the component of a different vendor is an example of the _____ pattern. Answer: Wrapper	Façade		(7.3%)
	Proxy		(22.0%)
	Wrapper	19.5%	
	Bridge		(41.4%)
	Broker		(9.8%)
		19.5%	(80.5%)
3. Having store computers send credit card requests to the central Winnipeg computer for processing is an example of the _____ pattern. Answer: Client-Server	Iterator		0.0%
	Mediator		(53.7%)
	Observer		(2.4%)
	Strategy		0.0%
	Another	2.4%	(29.3%)
	None of the above		(12.2%)
		2.4%	(97.6%)

Table E.3: Study 2 Pattern Cases – Test results

The test results for the second study of pattern ontology are given in Table E.4.

Question	Pattern Properties	Detection (n=41)	
		Accurate	Inaccurate
1. The following generic pattern/ sub-pattern is not present in the user requirements. Answer: Share/Collaborate	Isolate/Mediate	92.7%	(7.3%)
	Isolate/Transfer	90.2%	(9.8%)
	Share/Provide	82.9%	(17.1%)
	Share/Collaborate	65.9%	(34.1%)
2. Designing the system so that the All Stream transmission component can be used by the updated system is an example of the _____ generic pattern. Answer: #1 Isolate/Transfer #2 Isolate/Mediate #3 Isolate	Isolate	14.6%	
	Isolate/Mediate	36.6%	
	Isolate/Transfer	26.8%	
	Share		(19.5%)
	None of the above		(2.5%)
		78.0%	22.0%
3. Having rural computers make Internet requests through the central MTS computer is an example of the _____ pattern. Answer: #1 Proxy #2 Isolate/Transfer	Publisher-Subscriber		(7.3%)
	Observer		(4.8%)
	Proxy	65.9%	
	Share/Provide		(9.8%)
	Isolate/Transfer	2.4%	
	Isolate/Mediate		(9.8%)
		68.3%	(31.7%)

Table E.4: Study 2 Pattern Ontology – Test results

Appendix F

Study Results vis-à-vis the Hypotheses

A summary of study results vis-à-vis the hypotheses is given in Table F.1 and in Table F.2.

In all cases we used a two-sample Student t-test for independent groups with two-tail P values to determine if the null hypothesis might be safely rejected.

For the first hypothesis about pattern cases, sample 1 was taken from the baseline questionnaire results, with an "I do not know enough yet about finding patterns in user requirements to detect the patterns" answer or an incorrect detection being treated as a failure to detect the pattern accurately in domain knowledge.

For the second hypothesis about pattern ontology, sample 1 was taken from the pattern-case results, since these results were the most representative of the pattern-detection ability of participants immediately before the pattern-ontology treatment and tests.

Test	Pattern-Detection Sample 1	Pattern-Detection Sample 2	P-value (two-tail)	H_0	H_1	H_2
Pattern Cases						
Question 1	*Façade*	*Façade*	*p ≈ .532*	Reject	Support	-
	Proxy	Proxy	p < .05			
	Bridge	Bridge	p < .001			
	Low detection accuracy: ≈ 50% accurate detection of Façade, Proxy, and Bridge					
Question 2	Bridge	Bridge	p < .001	Reject	Support	-
	Wrapper	*Wrapper*	*p ≈ .603*			
	Low detection accuracy: ≈ 50% accurate detection of Bridge ≈ 25% accurate detection of Wrapper					
Question 3	Client-Server	Client-Server	p < .001	Reject	Support	-
	Low detection accuracy: ≈ 10% accurate detection of Client-Server					
Pattern Ontology						
Question 1	Façade	Isolation/Translation	p < .001	Reject	-	Support
	Proxy	Isolation/Mediation	p < .05			
	Bridge	Isolation/Mediation	p < .01			
	Improved detection accuracy: ≈ 95% accurate detection of Isolation/Translation ≈ 70% accurate detection of Isolation/Mediation					
Question 2	Wrapper	Isolation Total	p < .001	Reject	-	Support
	Wrapper	Isolation/Translation	p < .001			
	Improved detection accuracy: ≈ 90% accurate detection of Isolation ≈ 70% accurate detection of Isolation/Translation					
Question 3	*Client-Server*	*Isolation/Mediation*	*p ≈ .064*	null	-	?
	No detectable improvement in detection accuracy					

Table F.1: Study 1 results vis-à-vis the hypotheses

Test	Pattern-Detection Sample 1	Pattern-Detection Sample 2	P-value (two-tail)	H_0	H_1	H_2
Pattern Cases						
Question 1	Façade	Façade	$p < .001$	Reject	Support	-
	Proxy	Proxy	$p < .001$			
	Bridge	Bridge	$p < .001$			
	Low detection accuracy: ≈ 50% accurate detection of Proxy and Bridge					
Question 2	Bridge	Bridge	$p < .001$	Reject	Support	-
	Wrapper	*Wrapper*	*$p \approx .788$*			
	Low detection accuracy: ≈ 50% accurate detection of Bridge ≈ 20% accurate detection of Wrapper					
Question 3	Client-Server	Client-Server	$p < .001$	Reject	Support	-
	Low detection accuracy: ≈ 5% accurate detection of Client-Server					
Pattern Ontology						
Question 1	*Façade*	*Isolate/Transfer*	*$p \approx .134$*	Reject	-	Support
	Proxy	Isolate/Transfer	$p < .05$			
	Bridge	Isolate/Mediate	$p < .001$			
	Improved detection accuracy: ≈ 90% accurate detection of Isolate/Transfer & Mediate					
Question 2	Wrapper	Isolate Total	$p < .001$	Reject	-	Support
	Wrapper	Isolate/Transfer	$p \approx .439$			
	Wrapper	Isolate/Mediate	$p \approx .087$			
	Improved detection accuracy: ≈ 80% accurate detection of Isolate, with requirement ambiguity causing split in detection of Transfer (≈ 30%) and of Mediate (≈ 40%)					
Question 3	Client-Server	Proxy	$p < .001$	Reject	-	Support
	Improved detection accuracy: ≈ 70% accurate detection of Proxy					

Table F.2: Study 2 results vis-à-vis the hypotheses

Bibliography

Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. *SIGPLAN Notices*, 33(10):134–143, October 1998. doi: 10.1145/286942.286952.

Charu C. Aggarwal and Philip S. Yu. Mining associations with the collective strength approach. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):863–873, November 2001. doi: 10.1109/69.971183.

Wafi Al-Karaghouli, Sarmad AlShawi, and Guy Fitzgerald. Negotiating and understanding information systems requirements: The use of set diagrams. *Requirements Engineering*, 5(2):93–102, 2000. doi: 10.1007/PL00010348.

Yukiko Sasaki Alam. Lexical-semantic representation of the lexicon for word sense disambiguation and text understanding. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing*, ICSC '09, pages 83–88. IEEE, September 2009. doi: 10.1109/ICSC.2009.95.

Michael J. Albers. Information salience and interpreting information. In *Proceedings of the 25th Annual ACM International Conference on Design of Communication*, SIGDOC '07, pages 80–86. ACM, October 2007. doi: 10.1145/1297144.1297163.

Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

Gabriela Alexe, Sorin Alexe, and Peter L. Hammer. Pattern-based clustering and attribute analysis. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*, 10(5):442–452, 2006. doi: 10.1007/s00500-005-0505-9.

Javier Andrade, Juan Ares, Rafael García, Juan Pazos, Santiago Rodríguez, and Andrés Silva. A methodological framework for generic conceptualisation: problem-sensitivity in software engineering. *Information and Software Technology*, 46(10):635–649, August 2004. doi: 10.1016/j.infsof.2003.11.003.

Javier Andrade, Juan Ares, Rafael García, Juan Pazos, Santiago Rodríguez, and Andrés Silva. Definition of a problem-sensitive conceptual modelling language: foundations and application to software engineering. *Information and Software Technology*, 48(7):517–531, July 2006. doi: 10.1016/j.infsof.2005.05.009.

Giuliano Antoniol, Gerardo Casazza, Massimiliano di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, November 2001. doi: 10.1016/S0164-1212(01)00061-9.

Solomon Antony, Dinesh Batra, and Radhika Santhanam. The use of a knowledge-based system in conceptual data modeling. *Decision Support Systems*, 41(1):176–188, November 2005. doi: 10.1016/j.dss.2004.05.011.

Jonathan Arnowitz, Michael Arent, and Nevin Berger. *Effective Prototyping for Software Makers*. Elsevier, 2007.

Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina del Grosso, and Massimiliano di Penta. An empirical study on the evolution of design patterns. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE '07, pages 385–394. ACM, September 2007. doi: 10.1145/1287624.1287680.

Felix Bachmann, Len Bass, and Mark H. Klein. Illuminating the fundamental contributors to software architecture quality. Technical Report CMU/SEI-2002-TR-025, Defense Technical Information Center OAI-PMH Repository (United States), 2002.

Franck Barbier and Brian Henderson-Sellers. Object modelling languages: An evaluation and some key expectations for the future. *Annals of Software Engineering*, 10(1–4):67–101, January 2000. doi: 10.1023/A:1018935632373.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

Dinesh Batra and Nicole A. Wishart. Comparing a rule-based approach with a pattern-based approach at different levels of complexity of conceptual data modelling tasks. *International Journal of Human-Computer Studies*, 61(4):397–419, October 2004. doi: 10.1016/j.ijhcs.2003.12.019.

Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

Cinzia Bernardeschi, Nicoletta de Francesco, and Gigliola Vaglini. An approach to system design based on P/T net simulation. *Information and Software Technology*, 43(10):591–605, August 2001. doi: 10.1016/S0950-5849(01)00167-7.

Dines Bjørner. Pinnacles of software engineering: 25 years of formal methods. *Annals of Software Engineering*, 10(1):11–66, 2000. doi: 10.1023/A:1018983515535.

Marc Boyer and Vojislav B. Mišić. Generic patterns: Bridging the contextual divide. In *Proceedings of the 3rd International Workshop on Software Patterns and Quality*, SPAQu '09, pages 20–25, October 2009.

Andrew Burton-Jones and Peter N. Meso. Conceptualizing systems for understanding: An empirical test of decomposition principles in object-oriented analysis. *Information Systems Research*, 17 (1):38–60, March 2006. doi: 10.1287/isre.1050.0079.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

Margarida G. M. S. Cardoso and André Ponce de Leon F. de Carvalho. Quality indices for (practical) clustering evaluation. *Intelligent Data Analysis*, 13(5):725–740, October 2009. doi: 10.3233/IDA-2009-0390.

Sergio E. Chaigneau, Lawrence W. Barsalou, and Mojdeh Zamani. Situational information contributes to object categorization and inference. *Acta Psychologica*, 130(1):81–94, January 2009. doi: 10.1016/j.actpsy.2008.10.004.

Alexander Chatzigeorgiou, Nikolaos Tsantalis, and Ignatios Deligiannis. An empirical study on students' ability to comprehend design patterns. *Computers & Education*, 51(3):1007–1016, November 2008. doi: 10.1016/j.compedu.2007.10.003.

Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software – Practice and Experience*, 35(6):583–599, May 2005. doi: 10.1002/spe.v35:6.

Dan L. Chiappe and John M. Kennedy. Literal bases for metaphor and simile. *Metaphor and Symbol*, 16(3–4):249–276, October 2001. doi: 10.1207/S15327868MS1603&4_7.

Gillian Cohen. Hierarchical models in cognition: Do they have psychological reality? *Journal of Cognitive Psychology*, 12(1):1–36, March 2000. doi: 10.1080/095414400382181.

John D. Coley, Brett Hayes, Christopher Lawson, and Michelle Moloney. Knowledge, expectations, and inductive reasoning within conceptual hierarchies. *Cognition*, 90(3):217–253, January 2004. doi: 10.1016/S0010-0277(03)00159-8.

Maria Cutumisu, Curtis Onuczko, Duane Szafron, Jonathan Schaeffer, Matthew McNaughton, Troy Roy, Jeff Siegel, and Mike Carbonaro. Evaluating pattern catalogs: The computer games experience. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 132–141. ACM, May 2006. doi: 10.1145/1134285.1134305.

Jesse Daniels and Terry Bahill. The hybrid process that combines traditional requirements and use cases. *Systems Engineering*, 7(4):303–319, December 2004. doi: 10.1002/sys.v7:4.

Pauline Dibbets, J. H. Roald Maes, P. van den Berg, Sanne de Wit, and J. M. H. Vossen. Feature positive discriminations in adults and children. *Cognitive Development*, 17(2):1235–1248, April 2002. doi: 10.1016/S0885-2014(02)00115-6.

Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, November 2008. doi: 10.1016/j.infsof.2008.02.006.

Ravit Golan Duncan. The role of domain-specific knowledge in generative reasoning about complicated multileveled phenomena. *Cognition and Instruction*, 25(4):271–336, December 2007. doi: 10.1080/07370000701632355.

Tore Dybå, Vigdis By Kampenes, and Dag I.K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, August 2006. doi: 10.1016/j.infsof.2005.08.009.

Hernan R. Eguiluz and Mario R. Barbacci. Interactions among techniques addressing quality attributes. Technical Report CMU/SEI-2003-TR-003, Defense Technical Information Center OAI-PMH Repository (United States), 2003.

Magnus Eriksson, Kjell Borg, and Jürgen Börstler. Use cases for systems engineering—an approach and empirical evaluation. *Systems Engineering*, 11(1):39–60, February 2008. doi: 10.1002/sys.v11:1.

Eric Evans. *Domain-Driven Design*. Addison Wesley, 2003.

Joerg Evermann and Yair Wand. Toward formalizing domain modeling semantics in language syntax. *IEEE Transactions on Software Engineering*, 31(1):21–37, January 2005. doi: 10.1109/TSE.2005.15.

Frederico Fonseca and James Martin. Learning the differences between ontologies and conceptual schemas through ontology-driven information systems. *Journal of the Association for Information Systems*, 8(2):129–142, February 2007.

Nigel Ford. Modeling cognitive processes in information seeking: From Popper to Pask. *Journal of the American Society for Information Science and Technology*, 55(9):769–782, July 2004. doi: 10.1002/asi.20021.

Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Paul J. M. Frederiks and Theo P. van der Weide. Information modeling: the process and the required competencies of its participants. *Data & Knowledge Engineering*, 58(1):4–20, July 2006. doi: 10.1016/j.datak.2005.05.007.

Luka Fürst, Sanja Fidler, and Aleš Leonardis. Selecting features for object detection using an AdaBoost-compatible evaluation function. *Pattern Recognition Letters*, 29(11):1603–1612, August 2008. doi: 10.1016/j.patrec.2008.03.020.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

Jorge García-Duque, Martín López-Nores, José J. Pazos-Arias, Ana Fernández-Vilas, Rebeca P. Díaz-Redondo, Alberto Gil Solla, Manuel Ramos-Cabrer, and Yolanda Blanco-Fernández. Guidelines for the incremental identification of aspects in requirements specifications. *Requirements Engineering*, 11(4):239–263, August 2006. doi: 10.1007/s00766-006-0028-7.

Dedre Gentner and Brian F. Bowdle. Convention, form, and figurative language processing. *Metaphor and Symbol*, 16(3–4):223–247, October 2001. doi: 10.1207/S15327868MS1603&4_6.

Dedre Gentner and Kenneth J. Kurtz. Relations, objects, and the composition of analogies. *Cognitive Science*, 30(4):609–642, July 2006. doi: 10.1207/s15516709cog0000_60.

Marinos G. Georgiades, Andreas S. Andreou, and Constantions S. Pattichis. A requirements engineering methodology based on natural language syntax and semantics. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 473–474. IEEE, August 2005. doi: 10.1109/RE.2005.7.

Sam Glucksberg, Mary R. Newsome, and Yevgeniya Goldvarg. Inhibition of the literal: Filtering metaphor-irrelevant information during metaphor comprehension. *Metaphor and Symbol*, 16(3–4):277–298, October 2001. doi: 10.1207/S15327868MS1603&4_8.

Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001. doi: 10.1007/s007660170013.

Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, and Houari Sahraoui. Improving design-pattern identification: a new approach and an exploratory study. *Software Quality Journal*, 18(1):145–174, March 2010. doi: 10.1007/s11219-009-9082-y.

Serkan Gunal and Rifat Edizkan. Subspace based feature selection for pattern recognition. *Information Sciences*, 178(19):3716–3726, October 2008. doi: 10.1016/j.ins.2008.06.001.

Rubi Hammer, Gil Diesendruck, Daphna Weinshall, and Shaul Hochstein. The development of category learning strategies: What makes the difference? *Cognition*, 112(1):105–119, July 2009. doi: 10.1016/j.cognition.2009.03.012.

Jo E. Hannay, Dag I.K. Sjøberg, and Tore Dybå. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering*, 33(2):87–107, February 2007. doi: 10.1109/TSE.2007.12.

Scott Henninger and Victor Corrêa. Software pattern communities: current practices and challenges. In *Proceedings of the 14th Conference on Pattern Languages of Programs*, PLoP '07, pages 14:1–14:19. ACM, September 2007. doi: 10.1145/1772070.1772087.

Thomas T. Hills, Mounir Maouene, Josita Maouene, Adam Sheya, and Linda Smith. Categorical structure among shared features in networks of early-learned nouns. *Cognition*, 112(3):381–396, September 2009. doi: 10.1016/j.cognition.2009.06.002.

Nien-Lin Hsueh and Wen-Hsiang Shen. Handling nonfunctional and conflicting requirements with design patterns. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 608–615. IEEE, November 2004. doi: 10.1109/APSEC.2004.57.

Nien-Lin Hsueh, Jong-Yih Kuo, and Ching-Chiuan Lin. Object-oriented design: A goal-driven and pattern-based approach. *Software & Systems Modeling*, 8(1):67–84, February 2009. doi: 10.1007/s10270-007-0063-y.

Clemente Izurieta and James M. Bieman. How software designs decay: A pilot study of pattern evolution. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 449–451. IEEE, September 2007. doi: 10.1109/ESEM.2007.55.

Michael Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(14):903–912, November 2005. doi: 10.1016/j.infsof.2005.08.004.

Anil K. Jain, Robert P.W. Duin, and Jianchang Mao. Statistical pattern recognition: a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, January 2000. doi: 10.1109/34.824819.

Masita Abdul Jalil and Shahrul Azman Mohd Noah. The difficulties of using design patterns among novices: An exploratory study. In *Proceedings of the 5th International Conference on Computational Science and Applications*, ICCSA '07, pages 97–103. IEEE, October 2007. doi: 10.1109/ICCSA.2007.75.

Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

Dae-Kyoo Kim and Wuwei Shen. Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 16(3):329–359, September 2008. doi: 10.1007/s11219-008-9048-5.

Walter Kintsch and Anita R. Bowles. Metaphor comprehension: What makes a metaphor difficult to understand? *Metaphor and Symbol*, 17(4):249–262, October 2002. doi: 10.1207/S15327868MS1704_1.

Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '04, pages 111–126, June 2004.

Christian Kohls and Katharina Scheiter. The relation between design patterns and schema theory. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, pages 15:1–15:16. ACM, October 2008. doi: 10.1145/1753196.1753214.

Gwendolyn Kolfschoten, Stephan Lukosch, Alexander Verbraeck, Edwin Valentin, and Gert-Jan de Vreede. Cognitive learning efficiency through the use of design patterns in teaching. *Computers and Education*, 54(3):652–660, April 2010. doi: 10.1016/j.compedu.2009.09.028.

Sotiris B. Kotsiantis, Ioannis D. Zaharakis, and Panagiotis E. Pintelas. Machine learning: a review of classification and combining techniques. *Artifical Intelligence Review*, 26(3):159–190, November 2006. doi: 10.1007/s10462-007-9052-3.

Mark Last, Abraham Kandel, and Oded Maimon. Information-theoretic algorithm for feature selection. *Pattern Recognition Letters*, 22(6-7):799–811, May 2001. doi: 10.1016/S0167-8655(01)00019-8.

Tracy L. Lewis, Mary Beth Rosson, and Manuel A. Pérez-Quiñones. What do the experts say? Teaching introductory design from an expert's perspective. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE 04, pages 296–300. ACM, March 2004. doi: 10.1145/1028174.971405.

Zhi Li. Progressing problems from requirements to specifications in problem frames. In *Proceedings of the 3rd International Workshop on Applications and Advances of Problem Frames*, IWAAPF '08, pages 53–59. ACM, May 2008. doi: 10.1145/1370811.1370823.

Shu-Hsien Liao. Expert system methodologies and applications—a decade review from 1995 to 2004. *Expert Systems with Applications*, 28(1):93–103, January 2005. doi: 10.1016/j.eswa.2004.08.003.

Lassi A. Liikkanen and Matti Perttula. Exploring problem decomposition in conceptual design among novice designers. *Design Studies*, 30(1):38–59, January 2009. doi: 10.1016/j.destud.2008.07.003.

Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. A systematic review of domain analysis tools. *Information and Software Technology*, 52(1):1–13, January 2010. doi: 10.1016/j.infsof.2009.05.001.

Jean M. Mandler. Perceptual and conceptual processes in infancy. *Journal of Cognition and Development*, 1(1):3–36, February 2000. doi: 10.1207/S15327647JCD0101N_2.

Jean M. Mandler and Laraine McDonough. Advancing downward to the basic level. *Journal of Cognition and Development*, 1(4):379–403, November 2000. doi: 10.1207/S15327647JCD0104_02.

Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benefits. In *Proceedings of the 25th Annual Computer Software and Applications Conference*, COMPSAC '01, pages 574–579. IEEE, October 2001. doi: 10.1109/CMPSAC.2001.960670.

James Miller. Replicating software engineering experiments: a poisoned chalice or the Holy Grail. *Information and Software Technology*, 47(4):233–244, March 2005. doi: 10.1016/j.infsof.2004.08.005.

Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls. *International Journal on Software Tools for Technology Transfer*, 8(4):303–319, August 2006. doi: 10.1007/s10009-004-0173-6.

Vojislav B. Mišić. Coherence equals cohesion—or does it? In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, APSEC '00, pages 465–469. IEEE, December 2000. doi: 10.1109/APSEC.2000.896676.

Vojislav B. Mišić. *Software-Engineering Notes*. University of Manitoba, 2009.

Sandro Morasca. Refining the axiomatic definition of internal software attributes. In *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 188–197. ACM, October 2008. doi: 10.1145/1414004.1414035.

Gary P. Moynihan, Abhijit Suki, and Daniel J. Fonseca. An expert system for the selection of software design patterns. *Expert Systems*, 23(1):39–52, February 2006. doi: 10.1111/j.1468-0394.2006.00323.x.

Orna Muller. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the 1st International Workshop on Computing Education Research*, ICER '05, pages 57–67. ACM, October 2005. doi: 10.1145/1089786.1089792.

Orna Muller, David Ginat, and Bruria Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 151–155. ACM, June 2007. doi: 10.1145/1269900.1268830.

H. James Nelson and David E. Monarchi. Ensuring the quality of conceptual representations. *Software Quality Journal*, 15(2):213–233, June 2007. doi: 10.1007/s11219-006-9011-2.

James Noble. Classifying relationships between object-oriented design patterns. In *Proceedings of the Australian Software Engineering Conference*, ASWEC '98, pages 98–107. IEEE, November 1998. doi: 10.1109/ASWEC.1998.730917.

James Noble, Robert Biddle, and Ewan Tempero. Metaphor and metonymy in object-oriented design patterns. *Australian Computer Science Communications*, 24(1):187–195, January 2002. doi: 10.1145/563857.563823.

Ariadi Nugroho and Michel R.V. Chaudron. A survey into the rigor of UML use and its perceived impact on quality and productivity. In *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 90–99. ACM, October 2008. doi: 10.1145/1414004.1414020.

Mahamed G. H. Omran, Andries P. Engelbrecht, and Ayed Salman. An overview of clustering methods. *Intelligent Data Analysis*, 11(6):583–605, December 2007.

Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software & Systems Modeling*, 4(1):55–70, February 2005. doi: 10.1007/s10270-004-0059-9.

Russell S. Pierce and Dan L. Chiappe. The roles of aptness, conventionality, and working memory in the production of metaphors and similes. *Metaphor and Symbol*, 24(1):1–19, January 2009. doi: 10.1080/10926480802568422.

Helena Sofia Pinto and João P. Martins. Ontologies: How can they be built? *Knowledge and Information Systems*, 6(4):441–464, July 2004. doi: 10.1007/s10115-003-0138-1.

Sébastien Poitrenaud, Jean-François Richard, and Charles Tijus. Properties, categories, and categorisation. *Thinking & Reasoning*, 11(2):151–208, May 2005. doi: 10.1080/13546780442000169.

Vesna Popovic. Expertise development in product design—strategic and domain-specific knowledge connections. *Design Studies*, 25(5):527–545, September 2004. doi: 10.1016/j.destud.2004.05.006.

D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S. V. G. K. Raju, and A. Ananda Rao. An approach for pattern oriented software development based on a design handbook. *Annals of Software Engineering*, 10(1–4):329–358, January 2000. doi: 10.1023/A:1018904220078.

Rebeca P. Díaz Redondo, José J. Pazos Arias, Ana Fernández Vilas, Jorge García Duque, and Alberto Gil Solla. ARIFS methodology reusing incomplete models at the requirements specification stage. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):607–646, August 2005. doi: 10.1142/S021819400500249X.

Bob Rehder and Reid Hastie. Category coherence and category-based property induction. *Cognition*, 91(2):113–153, March 2004. doi: 10.1016/S0010-0277(03)00167-7.

Graeme Richards, Karl Brazier, and Wenjia Wang. Feature salience definition and estimation and its use in feature subset selection. *Intelligent Data Analysis*, 10(1):3–21, January 2006.

Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, November 1996. doi: 10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-#.

Johann Rost. Is "Factory Method" really a pattern? *ACM SIGSOFT Software Engineering Notes*, 29(5):1–1, September 2004. doi: 10.1145/1022494.1022519.

Ondrej Rypáček, Roland Backhouse, and Henrik Nilsson. Type-theoretic design patterns. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, ICFP '06, pages 13–22. ACM, September 2006. doi: 10.1145/1159861.1159864.

Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw Hill, 5th edition, 2002.

Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, May 2007. doi: 10.1007/s00766-007-0048-y.

Alexandr Shvets. *Design Patterns Simply*. Printia, 2008.

Chao Sima and Edward R. Dougherty. The peaking phenomenon in the presence of feature-selection. *Pattern Recognition Letters*, 29(11):1667–1674, August 2008. doi: 10.1016/j.patrec.2008.04.010.

Jason McC. Smith and David Stotts. Elemental design patterns: A formal semantics for composition of OO software architecture. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, SEW-27 '02, pages 183–190. IEEE, December 2002. doi: 10.1109/SEW.2002.1199472.

Pnina Soffer and Irit Hadar. Applying ontology-based rules to conceptual modeling: a reflection on modeling decision making. *European Journal of Information Systems*, 16(5):599–611, October 2007. doi: 10.1057/palgrave.ejis.3000683.

Ji Y. Son, Linda B. Smith, and Robert L. Goldstone. Simplicity and generalization: Short-cutting abstraction in children's object categorizations. *Cognition*, 108(3):626–638, September 2008. doi: 10.1016/j.cognition.2008.05.002.

John Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

Alistair Sutcliffe. Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International Requirements Engineering Conference*, RE '03, pages 320–329. IEEE, September 2003. doi: 10.1109/ICRE.2003.1232776.

Ladan Tahvildari and Kostas Kontogiannis. On the role of design patterns in quality-driven re-engineering. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, CSMR '02, pages 230–240. IEEE, March 2002. doi: 10.1109/CSMR.2002.995810.

Fadi Thabtah. A review of associative classification mining. *The Knowledge Engineering Review*, 22(1):37–65, March 2007. doi: 10.1017/S0269888907001026.

Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11): 896–909, November 2006. doi: 10.1109/TSE.2006.112.

Wouter van Diggelen and Maarten Overdijk. Grounded design: Design patterns as the link between theory and practice. *Computers in Human Behavior*, 25(5):1056–1066, September 2009. doi: 10.1016/j.chb.2009.01.005.

Veerle Vanoverberghe and Gert Storms. Feature importance in feature generation and typicality rating. *European Journal of Cognitive Psychology*, 15(1):1–18, January 2003. doi: 10.1080/09541440303600.

Marek Vokáč, Walter Tichy, Dag I.K. Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9(3):149–195, September 2004. doi: 10.1023/B:EMSE.0000027778.69251.1f.

Iris von Rooij and Todd Wareham. Parameterized complexity in cognitive modeling: Foundations, applications and opportunities. *The Computer Journal*, 51(3):385–404, 2008. doi: 10.1093/comjnl/bxm034.

Stefan Wagner and Florian Deissenboeck. Abstractness, specificity, and complexity in software design. In *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering*, ROA '08, pages 35–42. ACM, May 2008. doi: 10.1145/1370164.1370173.

Gursimran Singh Walia and Jeffrey C. Carver. A systematic literature review to identify and classify software requirement errors. *Information and Software Technology*, 51(7):1087–1109, July 2009. doi: 10.1016/j.infsof.2009.01.004.

Christine E. Wania and Michael E. Atwood. Pattern languages in the wild: exploring pattern languages in the laboratory and in the real world. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, DESRIST '09, pages 12:1–12:15. ACM, May 2009. doi: 10.1145/1555619.1555635.

Lynn Westbrook. Mental models: a theoretical overview and preliminary study. *Journal of Information Science*, 32(6):563–579, December 2006. doi: 10.1177/0165551506068134.

Katja Wiemer-Hastings and Xu Xu. Content differences for abstract and concrete concepts. *Cognitive Science*, 29(5):719–736, September 2005. doi: 10.1207/s15516709cog0000_33.

Guizhen Yang and Michael Kifer. Inheritance in rule-based frame systems: Semantics and inference. *Journal on Data Semantics*, 7:79–135, 2006. doi: 10.1007/11890591_4.

Uwe Zdun. Systematic pattern selection using pattern language grammars and design space analysis. *Software – Practice and Experience*, 37(9):983–1016, July 2007. doi: 10.1002/spe.v37:9.

Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9–10):1003–1034, August 2008. doi: 10.1016/j.infsof.2007.09.003.