A Formal Model of Design Patterns

Mike Mester

3 May 2010

Abstract

We present a novel formal model of design patterns in object-oriented programming and apply it to prove relationships between different design patterns. Our model captures the structure of a design pattern, allowing definitions of an equality relationship, a part-whole relationship, and a specialization-generalization relationship between design patterns. We present an example proof for each of these relationships, showing that Zimmer’s Objectifier pattern is equal to the classic Strategy pattern, that the Chain of Responsibility pattern is a part of the Decorator pattern, and that the Factory pattern is a specialization of the Abstract Factory pattern. For this last proof, we also provide a simple implementation of the patterns in C++ to justify the formal definition of the specialization relationship. In constructing our model, we argue for the treatment of design patterns as a higher level of abstraction over programs rather than as mere collections of accumulated best programming practices. We also examine the benefits of formalizing design patterns and then evaluate a few proposals in the literature, largely centered around the project of formalizing the Unified Modeling Language. Concluding that these efforts are insufficient to satisfy the goals we have for our model, we construct our formal model for design patterns. This model consists of a representation of classes and a definition of two relations on classes, inheritance and “containment” (i.e. aggregation, association, and composition treated as a single relation), thus capturing the static structure that design patterns represent.
1 Introduction

Given the incredible popularity of the object-oriented approach to programming and the centrality of design patterns to that approach, the fact that design patterns are still studied informally might be puzzling. After all, we would not expect such an important abstraction to escape formalization in another branch of computer science, particularly when informality bred disagreement, confusion, and ambiguity at the level that design patterns have. Nevertheless, design patterns have historically resisted and continue to resist formalization. From their roots in architectural literature to the canonical “Gang of Four” book, design patterns have always been presented as a concept that did not admit formal, rigorous reasoning.

In this report, we hope to make inroads against that entrenched informality of design patterns by developing a simple model for formally reasoning about patterns and their relationships with each other. Our model does not aspire to be a full formal semantics of object-oriented programming. Instead, we strive for simplicity, capturing only that which is essential to design patterns, representing only the defining features of the object-oriented approach that distinguish it from traditional programming. We take that essential feature to be the inheritance relation on classes and the static structure that inheritance creates in a program. Thus, we develop a model that formally represents the structural content of design patterns. This focus on structure is meant to distill design patterns down to their fundamental aspects, removing the useful-but-extraneous ancillary material that many patterns have accumulated over the years as a result of their informal, heuristic presentation. Our belief that inheritance is the sine qua non of the object-oriented approach heavily informs this distillation and the construction of our model more generally.

This report begins in Section 2 by presenting a particular view of design patterns that underlies our project. We explain why design patterns are really a higher level abstraction over code and why this view makes their formalization all the more important. In Section 3, we look at the UML and explain why it is not the formalization of design patterns that we seek here. In short, the UML is neither formal enough nor focused enough on design patterns specifically to capture what we hope our model can capture. We then begin the task of assembling our model, first by looking to formal semantics for a foundation, in Section 4. Formal semantics give us the building blocks of our model: a representation for classes and a semantics for and definition of the inheritance relation, which we argue is the crucial relation of object-oriented programming. Armed with these ideas from formal semantics, we present our full model in Section 5. That model consists of formal representations of classes and their members, definitions of the two kinds of relations on classes (i.e. inheritance and “containment”), a formal definition of a design pattern, and definitions of three natural relationships between design patterns. After defining and presenting our model, we apply it in three sample
proofs in Section 6. These samples serve a few purposes, the first of which is a proof of concept for the model. That our model supports the kind of mathematical reasoning that the proofs require suggests that we are at least on the right track. The second purpose of the proofs is to begin exploring the structure of the space of design patterns, which is one of the major motivations behind constructing a formal model of design patterns in the first place. Finally, in Section 7 and Section 8, we conclude by critically examining what the model has done to aid our understanding of design patterns and what might be done to extend it.

2 Design Patterns

Design patterns are often viewed as nuggets of collected expertise about how to properly structure object-oriented programs, little pieces of programmers' folk wisdom passed down from generation to generation. In this section, we would like to argue for a different view of design patterns, one that accords them a more fundamental role in the object-oriented philosophy. We present our perspective on design patterns as a higher level of abstraction in 2.1, showing that patterns do much more than express programming best practices. Instead, they link together broad classes of programs with a common structure. Then, in 2.2, we discuss the history of design patterns with an eye towards explaining why their presentation seems so stubbornly heuristic, informal, and imprecise even today, decades after the introduction of the concept. In 2.3, we examine some of the benefits of formalizing design patterns, and finally, in 2.4, we explain the goal of this report: developing a suitable formalization of design patterns.

2.1 A higher level of abstraction

Design patterns are useful representations of expert experience about how to solve a problem, but they are more than just collected wisdom. Instead, design patterns are a higher level of abstraction above programs, with each pattern capturing an entire class of programs. A design pattern, then, is more than a vehicle for transferring knowledge about good object-oriented design and more than a reference or educational tool. To be sure, patterns serve a critical role in actual software design and teaching, but they also provide insight into a taxonomy of program structures. Two programs that implement or use the same design pattern share a deep and important connection that might otherwise go unnoticed, obscured by differences at the code level. Perhaps one of the most important contributions of design patterns is their ability to make explicit these implicit relationships between superficially different programs.

The correct way to understand design patterns is therefore as an abstraction over the sets of programs
that instantiate them. This perspective was missing from the canonical “Gang of Four” book on design
patterns [16], which looked at patterns more for their usefulness in software engineering and implementation.
Later researches (e.g. [13]) recognized the more fundamental position of design patterns. Informally, a design
pattern represents a “template” that programs can instantiate, a skeletal structure onto which programmers
graft a program’s actual functionality. Although grouping programs together based on what they do (e.g. data
compression, encryption, etc.) might seem natural, design patterns reveal similarities in structure that are
often more enlightening. A design pattern unites into a coherent group all of the possible programs that use
that pattern, exhibiting a shared structure among them that differences in functionality or behavior might
otherwise obscure. Properly viewed, then, design patterns speak more to the organization and form of a
program than to its function. Put another way, design patterns are primarily structural and only secondarily
behavioral. This bias towards structure has important implications for what a a formal representation of a
design pattern should contain.

As a higher, structural abstraction, design patterns must intentionally omit many details in order to
achieve generality. A design pattern therefore underspecifies the programs that instantiate it. When a
design pattern specifies a class, for instance, it typically only specifies a critical function or two, the presence
of which defines the structure of the program. This is not to say that the class cannot have more public
functions. It almost always will, in fact, and this is no indication that the pattern is incomplete. Hoping to
simply read a program off a design pattern thus misses the purpose of the pattern. This underspecification
is where the design pattern as implementation tool and the design pattern as abstraction come apart:
implementers generally need more detail, since implementations cannot be underspecified. Abstractions, on
the other hand, by their very nature filter out details. When viewing design patterns as abstractions, as we
do in this report, the reader should remain mindful of this difference. Many of the features that the Gang
of Four point out in their pattern descriptions are not part of the patterns qua abstractions, although they
are no doubt invaluable for implementing them.

Similarly, design patterns may sometimes express structure that instantiating programs simplify, whether
due to syntactical limitations in the programming language or simplicity considerations. For example, a
design pattern \( \mathcal{P} \) might specify two abstract classes, \( A \) and \( B \), each serving a discrete purpose. A program
that uses \( \mathcal{P} \) might well implement both \( A \) and \( B \) with a single class \( C \), however, thereby conflating the
structurally different parts of the pattern. The program still instantiates the pattern, but it hides some of
the structure. We present this example to show that making inferences about a design pattern from sample
code implementing it can be dangerous, a point to which we return later in our discussion of the advantages
of formalization (see 2.3). Just as trying to reading off code from a design pattern should give one pause, so too should attempts to mindlessly read off a design pattern from code. Although they are obviously related, design patterns and code are separate abstractions. One is not shorthand for the other. Attempts like [1] to construct a formalization of design patterns from which tools can automatically generate C++ miss this important point.

In short, design patterns are an abstraction that is separate from and higher than code. Sample programs implementing a pattern can instruct and elucidate, but the pattern itself is a separate entity, over and above all implementations. Remembering that status of design patterns as higher abstractions will allow us to distill the often complicated patterns to their essential elements. Making a step towards that distillation is the fundamental goal of this report.

2.2 Informality

The history of design patterns has heretofore been dominated by an informal, implementation-driven perspective. The Gang of Four explicitly credit the inspiration for design patterns as Christopher Alexander’s architecture treatise [2], which presented a quasi-mystical version of patterns for use in designing buildings (see [16, §1.1]). Indeed, the paper that arguably introduced the concept of design patterns to computer science [3] is actually structured as an adaptation of Alexander’s work to object-oriented programming. Other prominent authors have advocated a “holistic,” even “humanistic” view of software design patterns consistent with Alexander’s philosophy (see e.g. [12]). Alexander has influenced the design pattern literature through [2] and his related works to such an extent that A Pattern Language might arguably constitute the founding text of the field. The overwhelming result has been an embrace of informality and ambiguity and an explicit rejection of mathematical rigor.

This informal approach may serve the goals of architects well, but it hampers the potential of design patterns for studying software. Although Alexander’s work likely provided inspiration for the articulation of design patterns as a higher level of abstraction, the study of patterns in software should avail itself of the powerful formal techniques that have pushed progress in the rest of the field. Informality in design patterns comes at a heavy cost.

One problem of informal design patterns, even for their use as implementation designs, is the ambiguity inherent to informal descriptions. Informality engenders confusion and disagreement, both as to which design pattern is which and as to what (if any) pattern a particular piece of code implements. Ambiguities like these can be fatal to the implementation of a large system. Formally described design patterns would go far
towards resolving these disagreements, enhancing the utility of patterns for implementation and for studying
program structure.

Another problem of informality is the lack of a rigorous basis for establishing relationships between design
patterns. When patterns are informal, grounding intuitive conceptions of how they relate is impossible. The
Gang of Four [16, §1.5] note the need to organize and document inter-pattern relationships, but their informal
approach forces them to fall back on qualitative assessments. Although Zimmer [24] went further in describing
inter-pattern relationships, he still could not transcend the informality of the underlying pattern descriptions.

There is, therefore, a taxonomic benefit to making a rigorous classification of design patterns, since such a
classification would add formal weight to decisions about how to catalog the growing population of patterns.

Redundant patterns could be identified and excised, while part-whole and specialization relations could give
rise to a hierarchy of design patterns. Moreover, viewing patterns as a level of abstraction in their own right
suggests that there is also much to be learned from a more formal approach to the structure of the design
pattern space itself. If patterns represent classes of programs (as we claim above), relationships between
design patterns reveal relationships between broad sets of programs. A rigorous look at the relationships
between design patterns, however, requires first that the patterns themselves be formally represented.

2.3 Benefits of formalization

The problems of informality discussed above in 2.2 suggest a few obvious benefits to a formalization of design
patterns. A formal pattern specification language like LePUS [13] resolves the ambiguities that the Gang of
Four’s natural language description style inevitably introduces. Implementers can thereby rest assured that
they are implementing the pattern the designer intended and designers can be confident that their intent has
been rendered faithfully. Formal specifications also put pattern descriptions at the right level of abstraction.

The Gang of Four’s extensive use of sample code to illustrate design patterns is didactically helpful but
obscures the status of patterns as an abstraction above code itself. When sample code is an integral part
of a pattern’s definition, as it must be in an informal description like the Gang of Four’s, design patterns
become polluted by the lower-level implementation details that running code requires. Formal specifications
more properly treat the design pattern as a separate abstraction, with its own description language.

For our purposes here, however, the most important benefit of formalization is likely the insight it gives
into the relationships between and structure of design patterns. Though the organizational benefits of those
relationships are important, perhaps more important is the view the relationships give into the structure of
the design space of programs. Formalization can show that a certain pattern is composed of several smaller
patterns, or that one pattern is actually a special case of another. Formalization also facilitates “looking inside” design patterns, rather than treating them as individual monoliths. Since we claim here that design patterns are abstractions over classes of programs, conclusions about patterns imply relationships between those classes of programs, relationships that might not be apparent from comparison of code alone.

Formalization also forces refinement of the concept of a design pattern to its essential elements. As we will argue in our discussion of UML (see 3.2), the complexity of current design pattern specifications belies any reasonable formal model. While the proliferation of extraneous descriptive apparatus for design patterns certainly serves some purpose, it distracts from the patterns’ essence as descriptions of program structure. Formalization imposes discipline on this proliferation, requiring the separation of that which is absolutely necessary from that which is merely convenient. Thus, simplicity is our explicit goal in constructing a formal model of design patterns. With the full knowledge that a simple model may ignore some important subtleties, we believe that its power in illuminating the fundamental structure of design patterns outweighs this loss of granularity.

2.4 Our purpose

The goal of this report is very limited, hoping only to develop a formalization that supports the sort of inferences about inter-pattern relationships described above in 2.3. We do not attempt to create a formal language for specifying design patterns; such a project would be beyond the scope of this report, and adequate systems like the UML and LePUS already exist. There are of course important, difficult problems that arise in crafting a formal pattern language, but they are largely orthogonal to this report. Instead, we will try to develop a system in which we can justify some of the relationships among patterns informally conjectured in [16] and [24]. Although most of these relationships are well known and seldom doubted, examining them in a rigorous but simple framework may give novel insights. We believe that reducing design patterns to their bare essentials and then representing them in a simple but formal structure can elucidate the structure of design patterns and, therefore, the structure of object-oriented programs.

3 The Unified Modeling Language

At first blush, the formalization of design patterns that we seek might seem to be a problem solved by the UML. After all, the UML is now the world standard for describing design patterns, and its syntax is exhaustively described in the several hundred pages of standards documentation that comprise UML
2.0 [17, 18]. UML even provides a “metamodel” for extending the UML concepts to applications as yet untouched by the standard.

In this section, we will explain why UML cannot accomplish the goals we seek to achieve with this report. First, in 3.1, we explain UML’s dual normative and descriptive roles and how they are inconsistent with the kind of formalization we want here. Next, in 3.2, we argue that the UML specification is not only too broad to reveal the essential elements of design patterns but also too narrowly focused on syntax to get at the underlying structure of patterns. Finally, in 3.3, we describe attempts to formalize the UML and what we can learn about this report’s project from them. We ultimately conclude that the UML provides some direction towards our goal but cannot get us all the way there.

### 3.1 UML’s uncertain status

At a philosophical level, the UML tries to maintain both a descriptive and a normative role, leading to confusion at best and contradiction at worst. Put another way, the UML standards committee refuses to decide whether the UML is intended to merely describe the way software is actually designed (and therefore take a descriptive role), or whether it specifies the correct way to design software (and therefore takes a normative role). The group that publishes the standard even maintains two separate sets of standards documents, a normative one and a “non-normative” one. Whether these two standard can coexist without contradiction is unclear, but they certainly make any effort to use the UML for formal purposes much more difficult. The actual use of the UML in practical software engineering will seldom follow any formally consistent syntax or semantics, so descriptive versions of the UML are poor candidates for building a formal understanding of design patterns. At the same time, the normative version overspecifies because it is defined with an eye towards Executable UML, which requires far too much detail for expressing design patterns. This tension between the descriptive and normative uses of the UML has practical consequences for the language as well.

More concretely, the UML cannot seem to decide whether it is a design pattern language, a medium through which designers and implementers communicate, or even a full fledged programming language in its own right. UML qua pattern description language, as we primarily view it in this report, actually represents only a very small part of the whole UML standard. If the UML were to be the formal system for reasoning about patterns that we seek here, it would therefore need to be seriously reduced. It remains unclear, however, whether a UML that consisted of, say, just class diagrams would be consistent or appropriate. Such a restricted version of the UML would also jibe poorly with the actual use of the UML in practice,
where the language finds its greatest utility in crafting software blueprints. In that role, the UML’s richness, complexity, and flexibility are all blessings, but design patterns operate at a level with substantially less granularity. Finally and most radically, some proponents of UML aspire to make the language a fully executable programming language, the so-called Executable UML [20]. Obviously, the precision of specification that Executable UML requires would make the language unsuitable for representing the higher level abstraction of design patterns. Indeed, Executable UML more or less turns UML into code, ignoring completely the distinction between patterns and code explained above in 2.1.

All of this confusion over the UML’s status makes the language a poor candidate for the sort of formalization we seek here. Because the UML tries to serve its many different masters, it cannot provide the simple, clean, formal model that will serve the purposes of this report. The UML is deliberately many things to many people [15, §1.1]. The model developed in this report, however, has a very limited purpose. Encumbering our model with the complexity of the UML would mask the simplicity of design patterns, defeating a major purpose of the formalization.

### 3.2 Simultaneously too broad and too narrow

Because the UML is designed to allow near-complete blueprinting of a program, it tends to radically overspecify design patterns. UML diagrams can convey almost anything about a program, but that detail is inappropriate for a higher level abstraction like a design pattern. Even if we restrict ourselves to UML class diagrams (a very small part of the entire language), UML diagrams encourage thinking at too low of a level for properly reasoning about patterns. Adding in the dynamic or behavioral patterns that the UML supports (e.g. activity diagrams or use case diagrams) further distracts from the more fundamental structural content that of a design pattern. These extra diagrams are certainly useful when designing and implementing a real software system, but that is precisely why they do not work well for describing design patterns. Design patterns deserve a representation that respects the level of abstraction at which they operate. Because the UML provides too much detail for that level of abstraction, it is too broad to serve as a formal representation of design patterns.

At the same time, the UML is too narrow for our purposes because it provides only a language for pattern description, rather than a mathematical representation of the structures that constitute the patterns. Put another way, the UML does not readily support the kind of formal mathematical reasoning that this report hopes to apply to design patterns. Attempts have been made to add formal semantics to the UML to bridge this gap (see below, in 3.3), but as we will see, they generally fall short. To ask the UML to provide the
representation of design patterns that we seek here therefore seems to ask too much of what is at heart a language rather than a representational system. The UML excels at providing software designers and implementers with a visual vocabulary for the concepts they frequently use in complex software systems. We should not expect it to comfortably support a mathematical representation of those concepts suitable for formal reasoning. To do so would burden the UML with yet another purpose, which it emphatically does not need.

3.3 Formalization efforts

Proposal have been offered for formalizing the UML by founding its syntax on semantics defined in a formal language like Z. The UML’s overbreadth, as discussed above in 3.2, means that even a properly formalized UML would not suffice for our purposes here. Moreover, the particular formalization attempts in the literature tend to suffer from a few common defects that make them inappropriate for reasoning about relationships among design patterns. Even if they cannot do the whole job, however, the UML formalizations give valuable guidance for the development of our model.

One tendency of several UML formalizations is to model classes as atomic entities with no internal structure [5], which the UML alternative LePUS mimics [13]. Relations among classes (e.g. inheritance) are then represented as relations on the atomic classes. This approach ignores the fact that classes are by definition complex objects composed of simpler elements. Since relations like inheritance are in fact defined by the internal structure of the classes themselves, allowing the relations to be defined separately from the classes gives too much freedom. Inheritance is after all not a relation that we arbitrarily impose upon a set of existing classes. The inheritance relation is instead defined by the classes themselves. The story is similar for other inter-class relationships like aggregation or association. Given this intimate, definitional connection between the structure of classes and the relations between them, modeling classes as atomic entities seems an odd choice. A more appropriate formalization would faithfully represent the internal structure of classes, since this internal structure is the defining feature of the object-oriented concept of the class. The benefits of the atomic approach are less clear.

The UML formalizations in the literature also tend to focus on enabling proofs about a single pattern, rather than trying to facilitate inter-pattern inferences [6, 7, 9, 14]. The goal of these efforts is more in the realm of proving that a UML diagram or model is self-consistent and realizable, rather than comparing different design patterns. There is also a strong trend toward making the formalization susceptible to automated reasoning, so that CASE tools can automatically check designs as designers create them. The
results of [4] showing that such reasoning is EXPTIME-complete, even for very restricted UML diagrams, casts doubt on the viability of the automated reasoning efforts. In any case, verification of UML diagrams is far from what we intend to accomplish in this report and requires a formalization of UML syntax the development of which is beyond our scope. Once again, our goals are far less ambitious, and the considerable complexity of UML formalizations distract from them.

UML formalizations therefore cannot remedy the problems with the UML that prevent it from accomplishing the goals of this report. Formalization might make precise any latent ambiguities in the incredibly detailed UML standard (if the formalization itself can be adequately understood), but the UML's problems with respect to design patterns run deeper than simple ambiguity. Even a perfect formalization of the UML would therefore fail to capture the essential structure of design patterns, which is what we are after here. The formalizations provide insight into one possible model of classes, namely the atomic model, and they show why atomic classes disregard one of the defining features of classes. While we can learn from the UML formalizations, their use of the atomic model fails give insight into the structure of design patterns. We must instead seek a model that is simpler and less cumbersome than the UML.

4 Formal Semantics

Unlike the UML, formal semantics do not appear likely to provide a ready solution to the problem of formally reasoning about design patterns. Indeed, formal semantics works at the level of programs, which we argued in 2.1 represent a lower level of abstraction than design patterns. Moreover, formal semantics tends to concern itself with what programs do, forcing a formal semantics description of design patterns into overspecification of behavior and underspecification of structure. Since design patterns are essentially structural, not behavioral (at least as we view them in this report), a proper characterization would do the exact opposite. We therefore cannot expect to find in formal semantics the solution to this report’s project. Nevertheless, formal semantics deals with many of the structural details that concern design patterns, structural details that even the formalizations of the UML ignored. Of particular interest are the representation of classes, which the UML treats atomically in spite of their inherent structure. Since inheritance is arguably the most fundamental object-oriented relationship (see 4.2 for more discussion on this point), the formal semantics of inheritance also bear heavily on the formalization of design patterns. The UML literature is largely silent on this question, so insights from formal semantics are especially valuable.

In this section, we present and discuss some results from object-oriented formal semantics as they relate
to our study of design patterns. First, in 4.1, we examine the literature on the mathematical structures used to represent classes, particularly the competing sets of objects and sets of attributes views. Then, in 4.2, we look at three formal approaches to the inheritance relation in object-oriented programming. These contributions from formal semantics heavily inform the model of design patterns we present in Section 5.

4.1 Representing classes mathematically

There are two basic approaches in the literature towards representing classes. They are not inconsistent, both latching onto reasonable definitions of a class and both therefore emphasizing different parts of the class structure. Before discussing the two types of representation, it is instructive to informally express the role of the class versus the object in object-oriented programming. Objects are by definition instances, specifically instances of classes. An object is therefore a run-time property of a software system: it is created after the program has begun executing and does not survive after execution halts. Many objects come into and out of being during the lifetime of a program execution, so the object diagram of a program (a lesser-used UML diagram type) must be defined with respect to a particular point in the execution. Classes, on the other hand, represent a part of the fixed, compile-time structure of a program. Classes are brought into being by the compiler; they can neither be created nor destroyed thereafter. This persistence is exactly why we can draw a class diagram for a program without contextualizing it on any particular time in the program’s execution. Similarly, the compile-time nature of classes makes them fundamentally static, structural elements of a program, whereas objects are more properly considered dynamic elements of a program. With this distinction between classes and objects refreshed, we can examine the first representation strategy for classes.

The first method for representing classes treats a class as a set of objects that instantiate it [6, 7], which seems most popular with formal semantics approaches based on the UML. We call this strategy the object-set approach. A class as a set of its instantiating objects carries intuitive appeal, capturing the sense in which classes are user-defined types. Indeed, classes are quite literally classes of objects. Because it defines classes in terms of objects, the object-set approach also prioritizes the behavioral elements of a program (i.e. the objects) over its structure. Given our view of design patterns as representative of higher level static structure about programs, this emphasis on behavioral structure makes the object-set approach less promising. Nevertheless, the intuitive appeal and elegance of the object-set approach make it a reasonable candidate for class representation.

Despite its advantages, the object-set approach carries consequences that are undesirable for our project.
Questions arise as to the ontological priority\(^1\) that the object-set approach imposes on object and classes. For classes to be defined in terms of objects, the objects must proceed the class in existence; they must be defined independently and more simply than the class is defined.\(^2\) This ordering contradicts our typical notions of the ontological relationship between class and object, however. After all, when we write an object-oriented program, we must define the class before we attempt to instantiate any objects of its type, suggesting that the object-set approach has its ontological priority backwards. There is also ambiguity over which objects are to be included in the definition of a class. Two natural possibilities exist. First, the class could comprise the set of objects of that class type that are in fact instantiated by the program in question. Second, the class could comprise all of the objects of that class type that ever could be instantiated by any program. The first definition has the advantage of simplicity, but it means that a single class used in several programs is unlikely to be represented by the same set in each program. Since our project here is about looking at classes across programs rather than examining single programs, that is a fatal limitation. The second definition, on the other hand, adds substantial complexity, since the number of potentially instantiating objects of a class is infinite (assuming unbounded memory). All told, despite its intuitive appeal, the object-set approach appears too cumbersome to serve our purposes.

A more promising approach to representing a class is as a set of its members [8, 10, 11, 23]. Note that in this representation, a class still corresponds to a set of instantiating objects, but its representation is more compact. Just as we define a class in an object-oriented language by defining its members (i.e. attributes and functions), the member representation defines a class by a set of members. Thus, the member representation reflects actual programming practice, unlike the object-set approach. The member representation also places classes in their appropriate ontological position relative to objects. Classes are also naturally defined across programs as well as within a single program: if a class in program \(A\) contains the same members as a class in program \(B\), they are one and the same class. Comparisons across programs and more importantly across design patterns are thereby facilitated.

Some problems remain with the member-based representation. One obvious question is how to represent the members of the class themselves. Attributes admit a relatively simple solution. A variable is just a name or label and a domain, which can be thought of as the variable’s type [8]. The name is merely a string and the type can be a class itself for complex variables or a set of possible values for an atomic

---

\(^1\)By “ontological priority” we mean roughly the order in which elements are defined. Some element \(\alpha\) is ontologically prior to another element \(\beta\) if and only if the definition of \(\beta\) in some way depends on \(\alpha\). Thus, in the object-set approach, objects are ontologically prior to classes because classes are defined as sets of objects (whose definition we leave unspecified here).

\(^2\)For the purpose of simplicity, we ignore any possibility of a recursive or self-referential definition, in which the object definitions reference the class to which they belong and vice versa.
type (e.g. $Z$ or some subset thereof for a typical integer type). Functions, however, pose a more troubling problem. Perhaps the simplest tactic is to represent functions by their signatures, since signatures consist only of labels and types, whose representation we have already presented. Representation by signature has parallels in actual object-oriented languages, too. Purely virtual methods in C++ and abstract methods and interfaces in Java define functions by their signature. This is not to say that the signature is intended to capture the entire specification of the virtual function. In almost every case, there will be other, informally expressed constraints on the behavior of implementing functions, but programming languages do not require those constraints to be expressed syntactically because doing so would be burdensome on the programmer and compiler. Thus emerges a limitation of signatures as representation: signatures say nothing about the behavior of a function and therefore almost necessarily underspecify the function.

Wegner and Zdonik [23] propose two other possibilities for representing functions in addition to the syntactic signature representation: a semantic approach based on formal specification and a pragmatic approach based on actual implementation. Formal semantics has a natural affinity for the semantic approach, but specifying constraints on functions in a formal language can be unnatural and unfamiliar. Worse, it remains unclear whether the sorts of constraints that design patterns put on function behavior are even susceptible to formal methods. Consider, for example, a design pattern that calls for a function that “notifies” an object that an event has happened, as in the Observer pattern. How to capture the wide range of behavior the “notify” function might take is far from obvious. At least for design patterns, then, the semantic approach appears to be a non-starter. The pragmatic, implementation-based approach suffers the problem of overspecification, since code imposes more constraints than the design pattern mandates. Expressing functions in a design pattern through code also breaks the separation between pattern and code that we argued for in 2.1.

Although the question of function representation remains of independent interest, signatures will suffice for the purposes of this report. Since we are expressly taking a structural view of design patterns, specifying the behavior of functions formally is largely orthogonal to our project. We therefore adopt the member-based representation of classes, with functions represented by signatures, knowing that that choice necessarily leaves out some important features of design patterns. The gains in simplicity, however, are worth the loss in descriptive power.
4.2 The inheritance relation

The second major contribution from formal semantics is a deeper understanding of the inheritance relation. Inheritance relates subclasses to superclasses and represents the fundamental structure of object-oriented programs. Indeed, inheritance is arguably the sine qua non of object-oriented programming [8], that which differentiates the object-oriented approach from other programming styles. Understanding and representing the inheritance relation appropriately is therefore critical to any study of design patterns. Fortunately, formal semantics provides two interesting ways of viewing inheritance.

Before examining methods for defining inheritance, it is instructive to note a few of the basic characteristics of the relation. As a notational convention, we will write $A\vdash B$ to mean that $B$ inherits from $A$ (i.e. $B$ is a subclass of $A$). Consider the directed graph defined by the inheritance relation, where classes are nodes and $uv$ is an edge if and only if $v\vdash u$. The inherently directed nature of the inheritance relation ensures that this graph will always be directed. Moreover, the inheritance graph will always be acyclic, since we ignore recursive types for simplicity. Apart from these two conditions, however, the properties of the inheritance relation depend on the specifics of the programming language in question. C++, for instance, allows both classes with no superclass and classes with multiple superclasses (multiple inheritance), making a C++ inheritance graph a DAG with potentially many separate connected components. Java, on the other hand, restricts the inheritance relation much more severely: all classes have a superclass (the universal base class Object, if none is specified) and only single inheritance is allowed. Thus, the Java inheritance graph is a single tree rooted at the Object class. As a side note, Java allows more flexibility in its interface relation, mimicking C++’s structure of potentially many unconnected DAGs.

The formal semantics literature provides two ways of viewing the inheritance relation in object-oriented programming. The first is “incremental modification,” whereby subclasses incrementally modify and extend the functionality of their superclasses [10, 22, 23]. This approach has the advantage of matching the intuitive purpose of inheritance in actual object-oriented programming practice. On the other hand, it is difficult to find a natural way to represent a modification of a superclass by a subclass in the class representation scheme we use here (see 4.1). To see this problem, recall that functions are represented by signatures. When a subclass modifies a function of its superclass, it must do so by changing the behavior only and leaving the signature unchanged (or else the subclass merely defines a new function). Put another way, polymorphism

\footnote{Ignoring recursive types is also justified by the practice of two major object-oriented languages, C++ and Java, neither of which supports circular inheritance. Indeed, given the standard C++ implementation of inheritance, objects of a class on an inheritance cycle would consume infinite memory, since infinitely many copies of the objects in the cycle would be included as superclasses.}
requires that function signatures remain constant as we travel down a path on the inheritance graph. If all
we use to represent a function is it signature, then, we ignore all of the modifications that the subclasses
make to it. Intuitively appealing though it may be, the incremental modification view of inheritance seems
difficult to fit with the class representation we have adopted.

The second view of inheritance in the literature is subtyping [19], whereby \( A \vdash B \) implies that \( B \) is a
“subtype” of \( A \). Subtyping captures the essence of polymorphism, an important consequence of inheritance
and one of the defining features of object-oriented languages. The status of \( B \) as a subtype of \( A \) has several
consequences for the semantics of \( B \) in a program. Two related axioms of the subtype relation have been
presented in the literature, the Principle of Substitutability [23] and the Axiom of Upwards Compatibility [22].
The Axiom of Upwards Compatibility is in fact a generalization of the Principle of Substitutability, which
we show below:

**Definition** (Principle of Substitutability). If \( B \) is a subtype of \( A \), then an object of type \( B \) must be able to
substitute anywhere an object of type \( A \) is used.

**Definition** (Axiom of Upwards Compatibility). If \( B \) is a subtype of \( A \), then everything that is true of \( A \)
must also be true of \( B \) (i.e. any proofs about a class must be monotonic over the inheritance relation).

**Claim.** The Axiom of Upwards Compatibility implies the Principle of Substitutability.

**Proof.** Assume by way of contradiction that the Axiom held but that for some particular pair of classes \( A \)
and \( B \) such that \( A \vdash B \), there was some situation where an \( A \)-object \( \alpha \) was used but a \( B \)-object \( \beta \) could
not be used in its place (i.e. the Principle does not hold). If replacing \( \alpha \) with \( \beta \) violates some constraint,
then there must be a property \( P(\alpha) \) such that \( \neg P(\beta) \), but this contradicts the assumption that the Axiom
holds (because \( P \) is true of \( A \) but not of \( B \)). Thus, we have proved that the Axiom implies the Principle, as
desired.

For the inheritance relation to induce proper subtyping on classes, it suffices to show that it satisfies
the Axiom of Upwards Compatibility, thus guaranteeing that intuitive notions of substitutability and poly-
morphism hold. The subtyping view of inheritance reduces to this single axiom that must be satisfied. If a
representation of classes and inheritance indeed satisfies that axiom, it correctly captures the semantics of
subtyping and therefore of inheritance.

Armed with the subtyping view of inheritance, we can define the inheritance relation \( \vdash \) in terms of our
member-set representation of objects presented in 4.1. Intuitively, when a class \( B \) inherits from \( A \), \( B \) gets all
of the members (e.g. attributes, functions, etc.) that \( A \) contains, plus the ability to add its own. Crucially,
\( B \) cannot remove any members it inherits from \( A \): the fact that \( A \vdash B \) implies that every member in \( A \) is
also in \( B \). When we represent classes as sets of members, then, \( B \) is simply a subset of \( A \). We can therefore
define inheritance in terms of the class representations themselves, instead of grafting a separate relation onto the model as the UML formalizations did (see 3.3):

**Definition** (Inheritance relation). For any two classes $A$ and $B$ represented by their member sets, $A \vdash B$ if and only if $A \subseteq B$.

In other words, $B$ contains every member that $A$ contains, possibly with extra members. Defining the inheritance relation is a crucial step in building a formal model of design patterns. This subset-based definition is supported by the formal semantics literature [8, 10], where inheritance is also defined by subset over classes. As an aside, note that this definition would reverse the subset order if we had used the object-set representation of sets discussed in 4.1, since $A \vdash B$ implies that all $B$-objects are $A$-objects but not necessarily vice versa. Thus, $B$’s object set is a subset of $A$’s object set, the reverse of what we define above. To justify our definition with respect to the view of inheritance as subtyping, it only remains to prove that the definition satisfies the Axiom of Upwards Compatibility:

**Claim.** The inheritance relation defined above satisfies the Axiom of Upwards Compatibility.

**Proof.** Recall that in our model, classes are represented by member sets (see 4.1). Thus, all the possible predicates that can be applied to a class must be defined in terms of the class’s members and its members alone. Indeed, when we model classes as member sets, the only acceptable predicates that can be applied to classes are existence of a particular member.\(^4\) Showing that our inheritance relation satisfies the Axiom therefore only requires showing that for any two classes $A$ and $B$ such that $A \vdash B$, if there exists some member $\delta$ that is a member of $A$ (i.e. $\delta \in A$), then $\delta$ is a member of $B$ as well (i.e. $\delta \in B$). By the definition of inheritance above, $A \vdash B$ implies that $A \subseteq B$, which shows that $\delta \in A \Rightarrow \delta \in B$, as desired.

The proof above shows that our inheritance relation is sound with respect to subtyping as a semantics for inheritance. Now that we have established a representation for classes (in 4.1) and a definition of the inheritance relation (in Definition 4.2), we are ready to present a formal model of design patterns.

## 5 A Formal Model of Design Patterns

In this section we discuss and develop a novel formal model for design patterns, based on the material presented in 4.1 and 4.2 on representing classes and inheritance, respectively. Before we begin detailing the

\(^4\)We thus exclude predicates like “contains exactly 4 members,” which if true of $A$ might not be true of $B$ even though $A \vdash B$ because $B$ adds members (as subclasses are allowed to do). Such pathological predicates are rightfully excluded, however, because the only sorts of questions that object-oriented languages will let us ask about an object are whether it has a certain member. In other words, the only predicates that ever get checked are these existence predicates, which the compiler checks whenever it encounters a piece of code that accesses a member of an object (e.g. Object $a$; $a$.foo();). If this exclusion seems ad hoc, the proof can just as well be interpreted as a definition of meaningful predicates operating on a class. Nothing is lost if that interpretation is adopted, but the exclusion is transformed into a matter of definition.
model, it is important to recall the very limited purpose of the model proposed here, which we first discussed in 2.4.

Our goal is to develop a simple model that captures, through design patterns, that which is essential to object-oriented programming specifically as opposed to programming more generally. To see the essence of the object-oriented approach, it helps to first identify that which is merely ancillary: programming language. What separates object-oriented programming from traditional procedural programming is not a choice of language. Using an object-oriented language like C++ or Java is neither necessary nor sufficient to produce truly object-oriented programs. Object-oriented techniques can be applied using plain ANSI C, and Java programmers are free to violate every object-oriented principle as often as they wish. Instead, the fundamental feature of object-oriented programs is the class structure, particularly with respect to inheritance. Even the name “object oriented” suggests that the division of a program into self-contained units (the classes) forms the core of the object-oriented philosophy. It should therefore come as no surprise that design patterns, which primarily capture the structure of programs, only became apparent after object-oriented programming had entered common practice. Many of the program structures that design patterns capture had been in use for years (see e.g. the history of the MVC pattern [21]), but it took an object-oriented lens to bring that common structure fully into focus.

Concretely, then, what the model will capture is the compile-time class structure of a design pattern, since it is the presence of class structure that defines object-oriented programming. To be sure, there are many dynamic and behavioral components to design patterns. This is neither surprising nor out of place. We should not forget that what we are ultimately after are programs, and programs ultimately must do something to be useful. Nevertheless, behavior cannot be what separates traditional programming from the object-oriented variety, because programs written in every style have always done things. Formally expressing the behavior of an object-oriented program is not in principle different from formally expressing the behavior of a traditional program. Structure is a whole different story, however. Since we intend this model as a study of object-oriented programming specifically, focusing our model on static structure is therefore justified.

We begin in 5.1 by recapitulating and formalizing the member-set class representation that we introduced in 4.1. In 5.2, we define the two primary relations that exist between classes, the more important of which is the inheritance relation introduced in 4.2. Then, in 5.3, we combine the concepts of classes and their relations to formally define a design pattern. Finally, in 5.4, we use this definition to explore the different formal relationships that can exist among design patterns.
5.1 Structure of classes

As we discussed in 4.1, we will represent classes by sets of their members. Formally, then, a class is a union of two disjoint sets:

**Definition 1 (Class).** A class $C$ is a set, $C = A \cup F$, where $A$ is a set of attributes (i.e. variables) and $F$ is a set of functions. An attribute $a \in A$ is an ordered pair, $a = \langle \text{name}, \text{type} \rangle$, where name is a unique string and type is either a class or a primitive type (i.e. a domain of possible values for the variable). A function $f \in F$ is a signature, represented by an ordered n-tuple consisting of a name, a return type, and a type for each argument.

It is important to note that the only members that are included in a class’s representation are those attributes and functions that are important to the pattern we are representing. Thus, we do not intend to include every public member of a class in its member-set, let alone every member of every scope. To do so would be to overspecify the pattern. A pattern will typically only define a few crucial attributes or functions for each class in it. Only those members are present in the classes’ respective representative sets.

We should also note before proceeding that while we have provided a workable representation for the attributes and functions in the set, the actual representation one chooses is not important for the proofs we present here. All the model requires is that the members support an equality operation that works “naturally,” so that we can determine if two members are equal in roughly the same way a reasonable compiler would.

5.2 Relations between classes

There are two important relations between classes in design patterns: the first is containment and the second is inheritance. We collapse composition, aggregation and association into a single relation that we call “containment” primarily for simplicity reasons. Although there are certainly important differences between the three, those differences are often subtle and open to debate. Moreover, all three implement versions of a more general “has a” relationship between classes, parallel to the “is a” relationship that inheritance models. Since inheritance is the critical object-oriented relation anyway, conflating the three containment relations loses little and allows the model to shed substantial complexity.

We proceed to define containment formally in terms of the definition of classes given in Definition 1:

**Definition 2 (Containment relation).** A class $A$ contains a class $B$ if and only if there exists an attribute $b \in A$ such that $b$.type $=$ $B$. If $A$ contains $B$, we write $B \in A$. 
Intuitively, we would like to simply say that when $A$ contains $B$, $B$ is a member of the set representing $A$. Unfortunately, because the attributes of a set are defined by both a name and a type, the traditional set-theoretic meaning of $B \in A$ does not quite work. Nevertheless, after overloading the $\in$ relation to work with our class structure, we can treat containment as simple set membership.

Our definition of the inheritance relation is the same as the one we presented in 4.2. We repeat it below for completeness:

**Definition 3** (Inheritance relation). For any two classes $A$ and $B$, represented by their member sets, $A \vdash B$ if and only if $A \subseteq B$.

These two relations, containment and inheritance, form the relational basis from which we build design patterns. Note, however, that both relations are defined in terms of the classes, rather than being separately specified mathematical objects. In other words, a set of classes determines both the containment and inheritance relations on that set, unlike in the formalizations of UML (see 3.3) where containment and inheritance were independently specified relations on atomic classes. This definition of inter-class relations in terms of the classes themselves fits with actual object-oriented programming, where containment and inheritance are determined by class definitions instead of being independently defined.

### 5.3 Defining a design pattern

With a formal representation of classes and the relations between them, we can now define a design pattern. Recall that our treatment of design patterns looks only to static structure; we do not seek to capture any of the behavioral or dynamic features of a pattern, some of which can be very important.

With that in mind, we define a design pattern as a three-tuple.

**Definition 4** (Design pattern). A design pattern is a three-tuple, $\mathcal{P} = (C, H, I)$, where $C$ is a set of classes (defined in Definition 1), $H \subseteq C \times C$ is the containment relation on $C$ (defined in Definition 2), and $I \subseteq C \times C$ is the inheritance relation on $C$ (defined in Definition 3).

Note that, as discussed in 5.2, both $H$ and $I$ are fully defined by a set $C$. Thus, the only free variable in the definition of a pattern is the set of classes, $C$. We include $H$ and $I$ in the definition, however, because it can be convenient to refer to them separately and because the intuitive notions and depictions of design patterns include them as separate elements.
5.4 Relationships among patterns

With the model built up through 5.1, 5.2, and 5.3, we can finally formally ground relationships among design patterns. Three obvious relationships suggest themselves: an equality relationship, a part-whole relationship, and a specialization-generalization relationship, the latter two introduced informally by Zimmer in [24]. Before we define these particular relationships formally, however, we must decide upon the appropriate way to represent inter-pattern relationships generally.

Morphisms are the natural representation for relationships between patterns, but the two relations (i.e. containment and inheritance) present in a pattern present us with an important choice. For instance, if we say that two patterns are homomorphic, it is unclear if we mean that they are homomorphic over the containment relation, the inheritance relation, or both.

Deciding among these alternatives is a matter of definition, but a few points argue in favor of defining the morphisms over the inheritance relation. First, inheritance is the fundamental relation of object-oriented programming, while containment is amply present in traditional programming (e.g., in C, a struct containing a pointer to another struct). Moreover, collapsing all of the classes connected by a containment relation into a single class makes no major structural differences to the code. Containment therefore appears to be a primarily organizational tool. Inheritance, on the other hand, gives rise to polymorphism, which has structural implications that are lost if inherited classes are collapsed into a single base class. For the purpose of this report, then, we will say that two patterns \( P \) and \( Q \) are morphic if and only if there exists a morphism \( f : C_P \to C_Q \) that preserves the inheritance relations \(^5\) in \( P \) and \( Q \) and that homomorphically preserves the containment relations in \( P \) and \( Q \). This convention corresponds intuitively to the notion that containments can be removed by simply combining classes but that inheritance structure is more fundamental.\(^6\)

With a morphism convention established, we formally define the three pattern relationships:

**Definition 5** (Equality). Two patterns \( P \) and \( Q \) are equal if and only if there exists a function \( f : C_P \to C_Q \) that is an isomorphism between \( P \) and \( Q \) over their respective inheritance relations and a homomorphism from \( P \) to \( Q \) or vice versa over their respective containment relations.\(^7\)

\(^5\)Formally, for all \( a, b \in C_P \), \( a \mapsto_P b \Rightarrow f(a) \mapsto_Q f(b) \) for a homomorphism or \( a \mapsto_P b \Leftrightarrow f(a) \mapsto_Q f(b) \) for an isomorphism. Preservation of the containment relation is similar.

\(^6\)Note that with one minor exception for the proof in 6.2, no proof we present in this report will depend on the convention above. The reader is therefore relatively free to substitute any other reasonable convention, including a two dimensional relationship structure whereby two patterns could be, say, inheritance homomorphic but containment isomorphic or any other permutation of morphisms and relations.

\(^7\)The “vice versa” addition in this definition is required to preserve the symmetry of the equality relation. Without the clause, we would only require a homomorphism over the containment relation from \( P \) to \( Q \), so it would be possible that there would be no homomorphism going in the other direction. Thus, we could not guarantee that if \( P \) were equal to \( Q \), then \( Q \) would also equal to \( P \), violating symmetry. An alternative solution to the “vice versa” clause is to change the morphism convention to require the same type of morphism for both inheritance and containment, in which case the containment morphism for equality...
Definition 6 (Part). A pattern $P$ is a part of a pattern $Q$ if and only if there exists some subset of $Q$’s classes $C'_Q \subseteq C_Q$ such that $P$ is equal to the pattern defined by the subset $C'_Q$ (recalling that the inheritance and containment relations in a pattern are defined by the classes).

Definition 7 (Specialization). A pattern $P$ is a specialization of a pattern $Q$ if and only if there exists a function $h : C_Q \rightarrow C_P$ that is a homomorphism from $Q$ to $P$ over their respective inheritance relations and a homomorphism from $Q$ to $P$ over their respective containment relations.

Informally, Definition 5 captures the idea that equality means two design patterns are structurally (at least in terms of inheritance) identical. The patterns might have different names for different components, but the underlying inheritance structure is the same. Similarly, Definition 6 says that for a pattern to be part of another pattern, it merely needs to be identical to some subset of the other pattern’s classes. Finally and perhaps most interestingly, Definition 7 captures the idea of a more specialized pattern as one with less structure than a more general pattern. If you homomorphically map one pattern onto another, as the specialization definition does, the mapped-to pattern must contain less structure than the mapping pattern (assuming the patterns are not also isomorphic). To further motivate these definitions, we provide a sample proof for each below in Section 6.

6 Sample Proofs

In this section, we present a sample proof on real design patterns for each of the three inter-pattern relationships defined in 5.4. Our first proof, in 6.1, shows that Zimmer’s pattern Objectifier [24, pp. 9–10] is actually equivalent to the traditional Strategy pattern [16, §5.9]. Thus, Objectifier is not a new pattern, contrary to Zimmer’s claims. Our second proof, in 6.2, illustrates the part relationship by proving that the Chain of Responsibility pattern [16, §5.1] is twice a part of the Decorator pattern [16, §4.4]. Finally, our third proof in 6.3 shows that the Factory pattern (also called Factory Method) [16, §3.3] is a specialization of the Abstract Factory pattern [16, §3.1]. For this last proof, we also provide and analyze sample code showing how the specialization relationship naturally reflects a one-way substitutability of the general pattern for the special pattern in code, but not vice versa. This short example program justifies the formal definition of the specialization relationship by showing how it is reflected in actual code.

The first proof is a novel contribution that arguably refutes Zimmer’s claim to a new design pattern. The latter two proofs, on the other hand, formally ground relationships that had long been conjectured (both by Zimmer [24] and by the Gang of Four [16], among others). Thus, these sample proofs show that the model would be an isomorphism and therefore reversible. This “vice versa” fix has the advantage of keeping our morphism convention, however. Note that this problem does not arise for Definition 7 because that relationship is inherently directional and therefore does not require symmetry.
can not only capture some of what is intuitively believed to be true about design patterns, but also provide new insights.

6.1 Objectifier is equal to Strategy

Zimmer claims to have identified a new, more general design pattern with Objectifier (UML diagram below, in Figure 1a). Looking at Objectifier next to the more traditional Strategy pattern (UML diagram below, in Figure 1b) suggests that Zimmer may have merely re-expressed the Strategy pattern, however. Indeed, the identical structure of the two patterns is obvious even on first inspection. The purpose of the two patterns is very similar as well. Zimmer says that the Objectifier is meant to turn behavior into an object so that it can be changed dynamically at run time and so that similar behaviors can be grouped together using inheritance, which is nearly identical to the Gang of Four’s description of the Strategy pattern’s purpose. We prove, using the model presented in Section 5 and the notion of equality presented in Definition 5 that Objectifier and Strategy are formally equal.

![Diagram](image-url)

(a) The Objectifier pattern  
(b) The Strategy pattern

Figure 1: The Objectifier pattern is equal to the Strategy pattern

Claim. The Objectifier and Strategy patterns are equal.

Proof. To prove equality under Definition 5, we must show that there exists an isomorphism $f$ from Objectifier’s classes to Strategy’s that preserves inheritance and that this isomorphism is also a containment homomorphism from Objectifier’s classes to Strategy’s classes. We will construct such a function: $f$ maps the Client class in Objectifier to the Context class in Strategy, maps the Objectifier class to the Strategy class, and maps the concrete Objectifier classes to the concrete Strategy classes. In other words, $f$ merely renames the classes in Objectifier to the names used in Strategy. Enumeration of the inheritance and containment relations shows that $f$ preserves all of them and $f$ is clearly invertible, so $f$ is an isomorphism as desired. We have thus shown that Objectifier is equivalent to Strategy.

23
The equality between Objectifier and Strategy seems naturally representable in code as well. All a translation from code implementing Objectifier to code implementing Strategy would require is a change of class and function names. Thus, the two patterns are equal, since names are not crucial parts of a program’s structure. Instead of introducing an entirely new pattern, then, Zimmer appears to have come up with a more general (or simply different) version of the Strategy pattern. It is unclear how Zimmer intended to differentiate his Objectifier pattern from Strategy and others, but the proof here shows that he cannot do so on the grounds of structure, which is the fundamental content of a design pattern.

6.2 Chain of Responsibility is a part of Decorator

In this section, we show that Decorator is composed of two instances of Chain of Responsibility by using the part relation defined in Definition 6. The Chain of Responsibility pattern is depicted below in Figure 2a, and the Decorator pattern is also depicted below in Figure 2b. Note that we have annotated the Decorator diagram to show the two Chains of Responsibility present in the pattern. Both Zimmer and the Gang of Four recognize this relationship, but neither grounds it formally. Perhaps as a result, neither expressly recognizes that the Decorator pattern is in fact two Chains of Responsibility, a subtler fact about the relationship that our formal model can capture. We now prove that Chain of Responsibility is twice a part of Decorator.

![Figure 2: The Decorator pattern is made up of two Chains of Responsibility](image)

**Claim.** The Chain of Responsibility pattern is twice a part of the Decorator pattern.

**Proof.** Recall that by Definition 6, to prove the Chain of Responsibility is a part of Decorator, we must show that Chain of Responsibility is equal to some sub-pattern of Decorator. We will do this twice, first showing
that Chain of Responsibility is equal to the sub-pattern of Decorator labelled “Chain of Responsibility A” in the diagram above. For convenience, call the Chain of Responsibility pattern \( C \) and the Decorator sub-pattern \( D_A \subset D \). Call the function that serves as the inheritance isomorphism and the containment homomorphism \( f_A \), which maps Handler to Component, ConcreteHandlerA to ConcreteComponent, and ConcreteHandlerB to Decorator. Observe that \( f_A \) is invertible and that it preserves both the inheritance and containment relations, so it is an inheritance and containment isomorphism as desired. Thus, \( C = D_A \) by the definition of equality given in Definition 5 and \( C \) is therefore a part of \( D \) by the definition of part given in Definition 6.

We now proceed with the second containment, showing that \( C \) is also equal to the Decorator sub-pattern \( D_B \subset D \). We call the map from the classes of \( C \) to the classes of \( D_B \) \( f_B \), which is defined as follows: \( f_B \) maps Handler to Decorator and maps the each concrete Handlers to one of the concrete Decorators (i.e. ConcreteHandlerA to ConcreteDecoratorA, etc.). The map clearly preserves inheritance and is also invertible, so it is an inheritance isomorphism. \( f_B \) is also a containment homomorphism, even though Decorator in \( D_B \) does not contain itself but Handler in \( C \) does.\(^8\) This is so because we only require that containment be homomorphically preserved from \( C \) to \( D_B \) or (as is the case here) from \( D_B \) to \( C \), but not necessarily both. Thus, \( C = D_B \) by the definition of equality given in Definition 5 and \( C \) is therefore a part of \( D \) a second time by the definition of part given in Definition 6.

This proof demonstrates the intuitive operation of the part relation. The purpose of the Decorator pattern is to chain responsibility for a component through potentially many decorators, so it is not surprising that the pattern uses the more basic Chain of Responsibility pattern. Our model captures this intuition with the proof above.

Note that this proof also gives the first hint of a possible formal hierarchy of design patterns. Since Chain of Responsibility is a part of Decorator but obviously not vice versa,\(^9\) we could argue that Chain of Responsibility is therefore a more fundamental or basic pattern than Decorator is. The part relation allows us to ground that ranking formally. There is potentially interesting work involved in trying to establish other part-whole relationships to see if more patterns fall into this natural ordering.

### 6.3 Factory is a specialization of Abstract Factory

For our final sample proof, we will show that the Factory pattern is a special case of the Abstract Factory pattern. The Factory pattern’s UML diagram is presented below in Figure 3a and the Abstract Factory pattern’s UML diagram is presented below in Figure 3b. Note that although we have labelled the containment relation with the tag “creates” to show that the factory objects instantiate new product objects, the relation is still normal containment and the pattern as presented here is still purely structural.

---

\(^8\)Readers might recall from note 3 and the accompanying text that we explicitly excluded recursive types in our definition of the inheritance relation. On first glance, the self-contains in the Chain of Responsibility and Decorator patterns might seem to violate that constraint, since a class \( C \) is a member of itself. One way around this difficulty is to define the type of the self-contained attribute as a reference or pointer type (which would be primitive) rather than as an actual class. We would thereby avoid the problem of infinite regress, much the way that actual programming languages do in similar circumstances.

\(^9\)The part relation requires an isomorphism, so since there are more classes in Decorator than in Chain of Responsibility, no isomorphism from Decorator to Chain of Responsibility could exist.
Like the relationship in 6.2, the existence of this relationship was no mystery to Zimmer, the Gang of Four, or anyone else. Even the names of the patterns suggest a connection. Unlike the two proofs above, however, the relationship between Factory and Abstract Factory is not an isomorphism that simply renames classes. Instead, to map Abstract Factory onto Factory, we need to collapse structure, leading to the conclusion that Factory is a specialization rather than an equivalent pattern. We now proceed to prove that the Factory pattern is a specialization of the Abstract Factory pattern.

(a) The Factory pattern

(b) The Abstract Factory pattern

Figure 3: The Factory pattern is a special case of Abstract Factory

Claim. The Factory pattern is a special case of the Abstract Factory pattern.

Proof. According to the definition of specialization given in Definition 7, we must show that there exists a function \( h \) from the classes of the AbstractFactory pattern (abbreviated \( \mathcal{A} \)) to the classes of the Factory pattern (abbreviation \( \mathcal{F} \)) that is an inheritance and containment homomorphism. We will construct such a function: \( h \) maps AbstractFactory to Factory, maps both concrete factories (A and B) to ConcreteFactory, and maps all of the product classes (AbstractProduct, ProductA, and ProductB) to Product. The factory inheritances are clearly preserved by this mapping, since Factory \( \triangleright_{\mathcal{F}} \) ConcreteFactory. The product inheritance are likewise preserved because the inheritance relation is by definition reflexive (see Definition 3), meaning that Product \( \triangleright_{\mathcal{F}} \) Product by definition. Showing containment preservation is trivial, since the only members defined by the pattern are the create() methods, which both Factory and ConcreteFactory contain. Thus, \( h \) is an inheritance and containment homomorphism, and the Factory pattern is a specialization of AbstractFactory, as desired.

To help show that this notion of inheritance homomorphism is a natural representation of specialization of design patterns, we illustrated this relation between Factory and Abstract Factory in code. To do this, we wrote two very simple C++ programs,\(^{10}\) both of which model actual factories producing widgets. The factory program uses a Factory pattern to produce a single type of product, called Widget. As is evident

\(^{10}\)A description of the source files and how to build these programs is provided in the appendix.
from the UML diagram for the Factory pattern, this is a canonical application of the pattern. To show that the Factory pattern is merely a specialization of Abstract Factory, we also wrote a \textit{factory-converted} program, which has the same functionality but uses the Abstract Factory design pattern. If our claim about specialization is correct and applies to actual programs, the source code for \textit{factory-converted} should be only trivially different from the source code for \textit{factory}. Fortunately for our model, this prediction holds. The only change to the code required for the pattern translation was an additional layer of abstraction over the Widget class, by way of an AbstractWidget class from which it inherited. This extra abstraction is in this program unnecessary, but because Factory is a specialization of Abstract Factory, the switch in patterns is natural and graceful. Indeed, the major difference in code between \textit{factory} and \textit{factory-converted} is contained in only a single line (the macro ABSTRACT is defined during compilation of \textit{factory-converted}). The simplicity of the transition from Factory to Abstract Factory supports our formal notion of specialization.

```
//The only difference between the factory and abstract factory is the type 
//of the widgets, which will be abstract in the abstract factory pattern
#else ABSTRACT
AbstractWidget** widgets = new AbstractWidget*[WIDGET_NUM];
#else
Widget** widgets = new Widget*[WIDGET_NUM];
#endif
```

Figure 4: Comparison of code from \textit{factory} and \textit{factory-converted}

To further justify specialization as a one-way relationship, we conducted a similar experiment but in reverse, trying to go from an Abstract Factory pattern program to a Factory pattern program. The program \textit{abstractfactory} works much like \textit{factory} except that it produces two kinds of products, RedWidgets and GreenWidgets, both of which are AbstractWidget subclasses and have their own factory classes. Thus, \textit{abstractfactory} is a natural implementation of the Abstract Factory pattern. We then wrote \textit{abstractfactory-converted}, an attempt to convert \textit{abstractfactory} into the more restrictive Factory pattern. First note that Factory only allows a single type of factory class and a single type of product, so our project hit trouble from the very beginning. In wrangling the code from \textit{abstractfactory} into the Factory pattern, we were forced to dispense with many core object-oriented techniques, chief among them polymorphism. Polymorphic function calls became switch statements on explicit type tags and virtual pointers became void*’s. The sample of code from \textit{abstractfactory-converted} corresponding to the code in Figure 4 demonstrates how unnatural the transition to Factory from Abstract Factory was (as above, the FACTORY macro is defined when \textit{abstractfactory-converted} is compiled).
In the abstract pattern, we can use the widget base class `AbstractWidget`, which handles both red and green widgets. Since we lack that base class in the factory pattern, we must store `void*`s to the objects and also store their types, so that we can know how to recast them to their real type later. Note that we are essentially implementing a very crude version of `vpointers here, with the types array serving as type tags.

```c
#ifdef FACTORY
    void** widgets = new void*[WIDGET_NUM * 2];
    WidgetFactory::WidgetType* types =
        new WidgetFactory::WidgetType[WIDGET_NUM * 2];
#else
    AbstractWidget** widgets = new AbstractWidget*[WIDGET_NUM * 2];
#endif
```

Figure 5: Code corresponding to Figure 4 for `abstractfactory` and `abstractfactory-converted`

The relative ease of converting Factory to Abstract Factory compared to the other direction argues in favor of our homomorphism-based definition of specialization between design patterns. If a pattern $P$ is a specialization of another pattern $Q$, we would intuitively expect that programs implementing $P$ could be converted to implement $Q$ without radical refactoring. Because specialization is one way, however, the reverse should not be true. Our simple example comported with those intuitions, suggesting that our definition of specialization is a natural one.

7 What The Formal Model Gives Us

Now that we have defined a formal model for design patterns and applied it in a few real proofs, it is appropriate to ask what the model has given us that we did not already have. The model seems to have four characteristics useful for a formal study of design patterns.

- A major advantage of our model over competing systems like formalizations of the UML is a simplicity that accurately reflects the level of abstraction at which design patterns operate. Although other systems may boast more explanatory power, particularly with respect to behavioral aspects of design patterns, they all seem too complex to capture the high level of abstraction that characterizes patterns. Because they represent very broad structure applicable across a wide range of programs, design patterns are naturally simple constructs. A formal model that faithfully represents them should respect that simplicity as much as possible. We believe our model follows that philosophy of simplicity, using only seven definitions to construct a representation of design patterns out of members of classes together with three natural relationships that can exist between patterns. The model certainly ignores much
about design patterns. Much of this ignorance is in fact intentional. We do not aim to replace the UML or LePUS as a way of specifying, recording, and communicating patterns. Instead, by stripping down our model to the bare essentials of design patterns, we hoped to elucidate some of the fundamental structure of design patterns that richer models might obscure. At least for our goals, then, simplicity is a virtue rather than a vice.

• Another contribution of our model is that it allows rigorous mathematical reasoning on concepts that were previously understood wholly or largely in informal terms. Our model is particularly well suited for proofs about the sort of inter-pattern relationships that Zimmer and the Gang of Four have already catalogued qualitatively. Even where a particular relationship was not in much doubt (e.g. that Factory specializes Abstract Factory; see 6.3), grounding common sense notions with formal reasoning has important benefits, giving a deeper view into a relationship than an informal approach can provide. The model can also prove novel relations, as in the proof that Objectifier and Strategy are equal (see 6.1). Given the self-avowedly anti-formal roots of design patterns [2], injecting even a small amount of mathematical rigor to their study could provide important insight into their underlying structure.

• One particularly important fruit of our formalization is the possibility of rigorously defining and justifying the complicated network of relationships between patterns that people have so far discussed informally. Formalizing these relations may give a better view of how different patterns relate to each other and might help solve the debates and ambiguity that necessarily arise out of informal specifications. The structure of the space of design patterns might also allow a hierarchy on or abstraction above individual design patterns, as we hinted at in our proof that Chain of Responsibility was a part of Decorator (see 6.2). The part and specialization relations might induce a natural partition of the design pattern space into different classes of patterns, for instance, as the Gang of Four tried to do in organizing their catalog of patterns. Alternatively, some patterns might turn out to be parts or generalizations of many other patterns, leading to a sense of a “more fundamental” or “more basic” pattern. All of these investigations into the structure of the space of design patterns are only really possible given a formalization of design patterns. Finally, the equality relation can help prevent a proliferation of structurally identical patterns camouflaged to look different, which is important as the number of design patterns seems to be rising quickly towards unmanageability. A formal equality relation ensures that we are really studying different patterns rather than different presentations or applications of the
same pattern.

- Finally and most philosophically, a formal understanding of design patterns, particularly a simple one, may shed light on what parts of design patterns in particular and object-oriented programming in general are essential and what parts are mere conveniences or ancillary additions. When we make decisions about which aspects of design patterns to reflect in our model and which to leave out, we make implicit judgments about how fundamental an aspect is. The most fundamental parts must be in the model at any cost; less fundamental features can be left out in service of simplicity or elegance. If the model we build in this way has some power to explain design patterns and their structure, then, we can reasonably assume that we made more or less correct judgments about what was essential. To the extent the model fails to explain something about design patterns, that failure may indicate a crucial omission in the model. By seeing how our model maps onto actual design patterns and object-oriented programming, then, we can begin to infer about what features define a design pattern and object-oriented programming. It has been a central if underlying thesis of this report (see e.g. 4.2) that static inheritance structure was chief among the central features of both design patterns and the object-oriented approach. If our model (which was based heavily on that claim) succeeds at all in explaining design patterns, that thesis might receive some minor validation.

These four contributions appear missing from other attempts at formalizing design patterns or object-oriented structure. Whether our model actually succeeds in providing them is unclear, but the model’s success in constructing the sample proofs presented here gives hope that it might at least be on the right track.

8 Conclusion

The main thrust of this report was the construction of a formal model for representing and reasoning on design patterns, described in Section 5, and the use of that model to prove some conjectured relationships between design patterns, in Section 6. To properly motivate and ground our model, however, we had to first evaluate the current state of design pattern research (in Section 2) and then look to other efforts in similar areas, particularly surrounding the UML and its formalization (in Section 3). After concluding that the UML formalizations could not accomplish our goals, we looked (in Section 4) to formal semantics for the basic tools out of which we could construct a simple model to capture the essential structural elements of design patterns. Our examination of formal semantics gave us the representation of classes and the inheritance relation that form the foundation of our model.
A few obvious next steps present themselves. We only exhibited three very simple proofs using our model in this report. It therefore remains to be seen whether the model is robust enough to facilitate more complicated proofs. One avenue for testing the model on this front would be to attempt proofs for the rest of Zimmer’s conjectured relationships between patterns. Zimmer’s catalog of relationships were the source of our sample proofs, but Zimmer presents many other, subtler relationships than the ones we proved here. Another interesting task is to see if the model can be extended to support some limited notion of behavioral or dynamic elements of design patterns. Although we were explicit in keeping our model focused on static structure in this report, the limitations of that approach emerged in the discussion of the factory patterns (see 6.3). We modelled the factories’ creation behavior as a containment relation, but there is something fundamentally behavioral about the factory patterns. Put another way, the factory patterns are patterns about doing in addition to structure. We still believe that static inheritance structure is the keystone of both design patterns and object-oriented programming. Nevertheless, adding to our model a way to capture behavior would allow us to appropriately reflect important behavioral nuances in the factory patterns and similar patterns. Finally, as discussed above in Section 7, it might be possible to use the model presented in this report to study the structure of the space of design patterns, perhaps establishing a hierarchy or categorization of patterns with more formal basis than the qualitative organization currently used by the Gang of Four and others.
Appendix A: Description of source code files and their compilation

There are two sets of source code, the source code for the Factory-to-Abstract Factory transitions and the source code for the reverse transition. Both code uses compiler macros to determine whether the original or transformed version should be compiled. Compilation of all the executables at once can be performed by simply executing make in the directory containing the make file and the source code. The behavior of the programs is very simple, consisting solely of creating a few objects and printing them to standard output.

A manifest of source files and their descriptions follows:

- **makefile**: The makefile for all of the executables. The target “all,” which is built be default, compiles all four executables.

- **Factory to Abstract Factory (executables factory and factory-converted)**
  - `widget-test.cpp`: The main file for both executables.
  - `factory.hpp`: An abstract class that defines the factory interface (class Factory) in the Factory pattern, implemented here by the WidgetFactory class defined in `widgetfactory.hpp`.
  - `widgetfactory.hpp`: The WidgetFactory class, implementing the Factory interface, which creates widgetswithincreasingserialnumbersandreturnsthem, in keeping with the Factory and Abstract Factory patterns.
  - `widget.hpp`: A simple product class called Widget, representing a widget with a serial number and the ability to print itself.
  - `abstractwidget.hpp`: An abstract interface to the Widget class, used only in `factory-converted`, since the Abstract Factory pattern requires that product classes like Widget be subclasses of an abstract base product class.

- **Abstract Factory to Factory (executables abstractfactory and abstractfactory-converted)**
  - `abswidget-test.cpp`: The main file for both executables.
  - `absfactory.hpp`: An abstract class that defines the factory interface (class Factory) in the Factory pattern, implemented here by the RedWidgetFactory and GreenWidgetFactory classes when we are using the Abstract Factory pattern and WidgetFactory class defined in `abswidgetfactory.hpp` when we are using the simple Factory pattern.
  - `redwidgetfactory.hpp`: An implementation of the Factory interface that makes RedWidgets.
- `redwidget.hpp`: A special type of widget (i.e. a product class), the RedWidget, again with only a serial number and the ability to present itself.

- `greenwidgetfactory.hpp`: An implementation of the Factory interface that makes GreenWidgets.

- `greenwidget.hpp`: A special type of widget (i.e. a product class), the GreenWidget, again with only a serial number and the ability to present itself.

- `abs_widgetfactory.hpp`: The WidgetFactory class, implementing the Factory interface, which creates widgets with increasing serial numbers and returns them. This class is only used when we are implementing the Factory pattern, so it must shoehorn the two different product types (RedWidget and GreenWidget) into a single factory without the benefit of a common superclass AbstractWidget. We resort to void*’s and type flags, showing how unnatural the transition from Abstract Factory to Factory is.

- `abstractwidget.hpp`: The same superclass for widgets as in the Factory to Abstract Factory transition, only this time it is actually necessary because we have both RedWidgets and GreenWidgets.
References


