
Chapter 12

Inter-model relationships

Over the years, software modelers have developed a useful set of modeling ideas. A clear understanding of these inter-model relationships is important, since you will use them in building models of software architecture. You are most likely using these ideas already: “This diagram is a zoomed-in view of this part of the other diagram,” or “This diagram shows the system from a performance perspective.”

To reinforce that these relationships are more general than software architecture or UML, you will see neither of those here. The chapter uses a running example of a house, with blueprints and other plans, that should be familiar domain to all readers.

This chapter explains these essential modeling relationships with sufficient precision that you should be able to understand and apply them. Increasing the formality will reduce the clarity of presentation for most readers. For example, binding is an easy concept, but to do it right requires that the symbol names are unique, and refinement is even more complex. If you were designing an architecture description language or a CASE tool to draw these models, you would require more precision than is used here.

The following sections discuss seven relationships between models, summarized in Table 12.1: projection, partition, composition, classification, generalization, refinement, and binding. Also discussed is the relationship between the real world and a model: designation. The chapter concludes with an example showing how all the relationships can be used together. We begin with projection, the most commonly used relationship.

12.1 Projection (view)

Cartographers have experimented with various projections of the Earth’s curved surface onto a flat map. Many projections have been invented and each requires a mathematical function that maps (i.e., projects) a spherical surface onto a planar surface. Perhaps the best

Relationship	From-To	Description
Projection	Model-Model	Subset of details with optional transformation
Partition	Model-Models	Subdividing a model
Composition	Models-Model	Combining models
Classification	Type-Instance	Categorization of instances
Generalization	Supertype-Subtype	Subsuming relationship between categories
Designation	World/Model-Model	Correspondences between models
Refinement	Model-Model	Low-detail to high-detail
Binding	Model-Model	Conforming to a pattern
Dependency	Model-Model	Change to one may imply change to other

Table 12.1: Summary of relationships

known is the Mercator projection, invented in 1569, which has the property that all lines of longitude and latitude intersect at right angles. On the Earth’s surface, such intersections occur only at the equator, so as a consequence the Mercator projection exaggerates the size of countries that are further from the equator. While falsely depicting Greenland as larger than Africa, this projection had the beneficial property that sailors wanting to get from one place to another could draw a line between the two and simply sail at that compass bearing. The Gall-Peters projection seeks to represent accurate sizes of countries but sacrifices the property of simple navigation.

A *projection*, which you can use interchangeably with the term *view*, can be informally thought of as what something looks like from a particular perspective. More formally, a projection shows a defined subset of a model’s details, possibly transformed. A projection can remove details, such as a map that omits country boundaries. It can also transform the model, as seen in how the Mercator and Gall-Peters projections choose their transformations for easy navigation or accurate area. However, a projection cannot add information that does not already exist — it would be quite surprising to project the globe onto a piece of paper and discover an eighth continent.

Projections are used when creating blueprints of houses. While a three-dimensional house is being designed, two-dimensional drawings of the house are produced. Each of these 2-D drawings is a projection of the entire house, as shown in Figure 12.1.

Consider a computer aided design (CAD) program that stores an internal representation of your 3-D house plans and can compute any 2-D view that you ask. It may seem that this contradicts the rule that “projections cannot add information that does not already exist,” because not every possible view already exists in the 3-D internal representation. But it is OK for the CAD program to transform its internal representation to show a view, which may require it to calculate and display cross-section. What the rule prohibits is views that would be impossible to derive from the 3-D internal representation, such as a new room, or a garage.

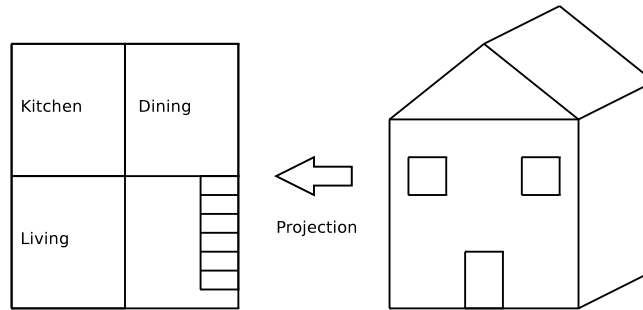


Figure 12.1: Projection showing 2-D floorplan view and 3-D model

Roof	\$17
...	...
Driveway	\$35
Total	\$100

Table 12.2: House cost view

Although you will use a lot of graphical views, some of the most useful are views textual or tabular. Table 12.2 shows a view of the house that lists the costs. You could draw it graphically, but it would be easier and better to use a spreadsheet program.

Consistency across views

Having more than one view introduces a challenging problem: maintaining *consistency* across multiple views. If you use a 3-D CAD program to edit your 3-D objects, the problem of inconsistent views will not arise because the program mechanically computes the views for you, and presumably does not make mistakes. However, designers often work with 2-D views and use their own brains to keep the various views consistent. My brother builds buildings and encountered this problem. The front-facing view of the house he was building showed rainspouts, which he built as designed. Once he started grading the terrain using a top-down view of the yard, however, he discovered that the design had the rainspouts exiting several feet below ground instead of at the surface. You would like to detect inconsistencies between views before you have bulldozers on site.

View consistency is one of the harder problems in software architecture (see Section 6.3). Section 13.1 discusses specific kinds of checks you can perform on architecture views. You can use techniques to check consistency between particular pairings of views, such as 2-D-floorplan to 2-D-side-view, but the number of specific pairings increases combinatorially with the number of views, so you should prefer general techniques when possible.

If keeping views consistent is a difficult problem, then there had better be good reasons for building views at all. Views help you cope with two primary foes: complexity and scale. By showing a subset of the details of the full model, a view necessarily reduces the amount

you need to comprehend. A view often highlights a single *concern* of the model, such as speed, airflow, or navigability. A specialist can use the view instead of the full model, for example as an electrician uses a wiring diagram to trace circuits. Section 13.4 discusses the benefits and challenges of separating concerns in software development. Views often correspond to *quality attributes*, which are described further in Section 11.10.

A view of what?

Let's start with two views of a house, like the views in Figure 12.2. If you look at these two floorplans, an problem jumps out: the staircases do not match up. When you build multiple views, you will sooner or later encounter the problem of the views not matching up. Interpreting the problem requires you to tackle a tricky question: where does the error lie?

There are a few ways to approach the problem of view consistency. One approach is to think of each view as a statement of the designer's intent, and then aggregate all the views as the combined design intent. For example, when designing the plans for a house, the architect may have desired the bathroom over the kitchen so that the plumbing lines are easy, and desired the master bedroom to face East. Each of these views acts as a constraint on the house. Let's refer to this approach as the *views as requirements* approach. Each view expresses requirements for the solution, and the question is whether or not you can design a model that satisfies the requirements from all the views. In the case of Figure 12.2, solutions exist for each view, but not for the combined views. My friend illustrates the challenge of conflicting requirements with the following example: "I want a 20-inch display that folds up into my pocket".

Another way of approaching the problem of view consistency is to proceed from the assumption that each view is a projection from a coherent, realizable design in the designer's head, and that inconsistencies between views are a result of an incorrect transfer between his mind and paper, or evidence of a design defect. This approach assumes that the design is a given, and that the views are derived from it mechanically. Let's refer to this approach as the *master model* approach. If the views are inconsistent, it is not the fault of the mechanically generated view, but a reflection of a design flaw. In the case of house floorplans, it is possible an error was made that flipped one of the floorplans, or the designer simply did not notice that the master model has an error.

The master model must contain all of the details needed to project the views, and all the details needed to design the real artifact. However, designers may choose to only draw a 2-D top-down view of a house before starting construction. The master model often does not exist except in the minds of the designers. But there is evidence it does exist: if you ask the designers questions about the design that go beyond that 2D view, you will receive answers that reveal the existence of the master model.

Another interpretation of views is that they are not projections of another model at all, but instead are projections of the real-world artifact. Using this interpretation, the floorplan blueprint of a house is either a projection of the to-be-built future house (using the views as requirements approach), or a projection of the actual house (using the master model

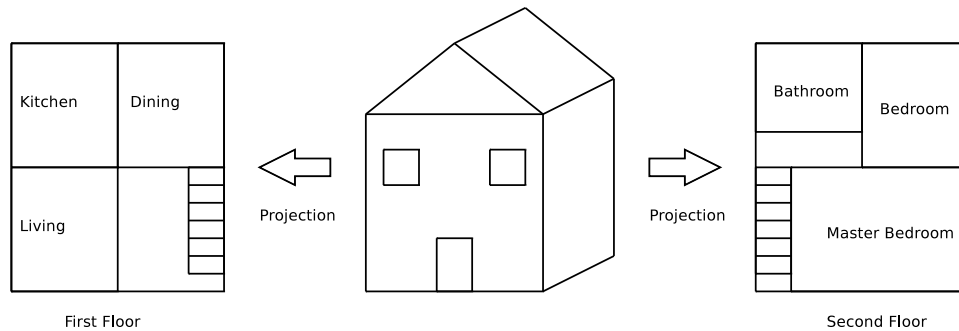


Figure 12.2: Multiple views

approach). This interpretation combines the idea of projection and designation into one step, like the way that experienced mathematicians will skip over or combine steps in their work.

This book uses the master model approach and show views that are projections of the master model. This way you avoid the possibility of creating a set of views for which there is no solution, and you keep the explicit step of showing correspondence between your models and the real world.

Analysis

A well-chosen view can aid in analyzing a model, and this analysis is often informal and visual. If you are trying to schedule contractors to work on your house, you would start with a list of the contractors and their availability dates. Conflicts are not obvious if the list is unordered, but if you place them onto a chart the conflicting appointments will jump out because your brain is good at detecting 2-D overlaps. Leveraging our built-in skills as humans to do architecture analysis is discussed in more detail in Section 13.3.

Other analyses can be done algorithmically by computers. If your new house is subject to local taxes based upon its square footage, number of windows, and energy efficiency then a specialized view can be used to calculate the tax burden of various design options. Each specialist, such as heating and air conditioning or electrical, likely has specialized analyses they perform using a custom view of the house.

Viewtypes

You may have noticed that views fall into natural groupings based on similarity. All of the physical views of the house can be reconciled by using a detailed 3-D design. But you can also view your house from various legal perspectives: tax liabilities, mineral rights beneath the house, and whether or not you can keep chickens in the backyard. It is difficult to imagine how these fit into a 3-D physical model, but perhaps you could make a legal model of the neighborhood and reconcile the additional views there. These clumpings of

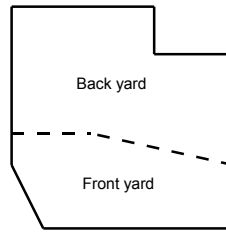


Figure 12.3: Partition

views are called *viewtypes*. A characteristic of a viewtype is that it is hard to reconcile it with another viewtype. In software architecture, the standard viewtypes are the module viewtype, runtime viewtype, and deployment viewtype. Section 8.4 discusses viewtypes in more detail.

12.2 Partition

One of the chores my brothers and I had to do growing up was to cut the grass. There was a lot of it, and we were using a push-mower, so we would *partition* the chore into the front yard and the back yard and take turns cutting the parts. Between those two parts, the entire yard was covered. We could have chosen different partitions, but our father mostly cared about complete coverage.

If you delight in corner cases, you may have thought, “What if I define a projection that only shows the front yard, and another projection that only shows the back yard?” Indeed, that would yield the same result as our partitioning. But a partition requires that the pieces comprise a complete set that covers the domain without overlap, while a projection has no such requirement.

12.3 Composition

Composition is almost the mirror opposite of partitioning. Where partitioning takes a model and describes how it can be divided into smaller models, composition takes smaller models and creates a larger model. The difference is that the parts combined through composition need not be the parts that made up the whole — so I could compose the front yard, the back yard, and the neighbor’s yard (i.e., not part of the original partitioning of my yard) to make a huge yard. In modeling, this is quite useful when you have some parts of model you would like to share, for example shared datatypes used by both a frontend and backend.

12.4 Classification

A *system of classification* allows you to pick up something and decide what category it belongs to. Using the definition originating with Plato (Bowker and Star, 1999), an ideal

system of classification would have three properties. First, it would be unambiguous. Second, each thing would fit into one and only one category. And third, any item could be sorted into a category. Much like other ideas about ideals from Plato, like perfect geometric forms, you almost never see a classification system that strictly conforms to these properties¹.

Despite Plato's scheme for organizing, people are quite comfortable sorting things into multiple categories simultaneously. A drywall screw used in a house is in the categories of "fastener" and also "magnetic". According to Plato, these cannot be part of the same system of classification because a thing must fall into just one category. You can solve this problem either by deciding that there are really two classification systems — one by function and one by electromagnetism — or by simply dropping the one-and-only-one category requirement.

This book uses the word *type* to refer to a category and *instance* to refer to the thing itself, and allow an instance to have multiple types. *Classification* is the relationship between a type and an instance. The classification relationship can apply to component types and components, classes and objects, or other pairs of "category" and "categorized thing".

Resist the temptation to use the terms *class* and *type* interchangeably because it can cause confusion between the concept of classifying something and the implementation of that concept in an object-oriented programming language. Note that in most object-oriented programming languages, an object has a single *class*, even if that class is derived from many other classes (multiple inheritance).

It may come as a surprise that *specifications* and *implementations* also follow the classification relationship. If you write a specification (often shortened to just *spec*), like "A piece of furniture I can sit on", it may be satisfied by an implementation, like "this brown sofa" or "this red chair". Many implementations may satisfy, or *meet*, the specification.

You should be accustomed to having an instance, then asking what type it is. Specifications and implementations reverse this, because the spec usually exists before the implementation. Writing the spec first introduces the possibility of an overly broad definition, which means that spec (or type) would be satisfied by implementations (or instances) that the spec author did not intend. Similarly, a spec can be too narrow such that too few or even no implementations will satisfy it. In engineering, as opposed to math or philosophy, no specifications are perfect, and the effort spent writing them must be traded off against effort spent building the implementation.

12.5 Generalization

While classification describes how a type may categorize an instance, *generalization* describes how one type can subsume another. My house (an instance of a house) might be in

¹Another way to classify is to define a category with a *prototype* which exemplifies the category, and determine inclusion by similarity to that prototype. Empirical studies by Rosch (Rosch and Lloyd, 1978) indicate that this is likely how our brains work: we think that birds are small, quick moving, flying things that resemble sparrows. Ostriches and penguins make lousy birds and so it takes us longer to recognize them as belonging in the category "bird", but crows are pretty typical birds and robins are even more so. In this system of categorization, categorization is not boolean in or out, but rather degree of inclusion.

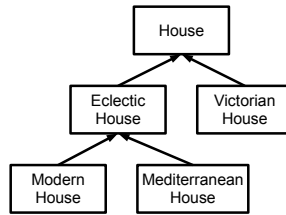


Figure 12.4: A partial taxonomy of houses

the category of “Modern” houses, but that means that my house is also of the type “Eclectic” and “House”, because “house” generalizes “Eclectic”, which generalizes “Modern”. The more general type is called the *supertype* and the less general type as the *subtype*.

The Liskov substitution principle (Liskov, 1987) provides an easy test for generalization: a subtype must be substitutable for its supertype. If a Modern house is a subtype of an Eclectic house, and you can sleep in an Eclectic house, then you can sleep in a Modern house. Note that in object-oriented programs you will often encounter subclasses that do not pass this test.

Supertypes and subtypes can be organized into a hierarchy, called a *taxonomy* (see Figure 12.4). Common examples include geometric shapes and the Linnean taxonomy used to classify living things. Taxonomies are unquestionably useful but a few cautions are in order. A common experience is that the initial construction of a taxonomy is easy but it becomes progressively more difficult as it approaches completeness. Taxonomies can become brittle over time because the instances being categorized change, and the way the taxonomy is used changes. Many useful categories crosscut established taxonomies; consider birds, bugs, bats, and biplanes as “flying things”. A final caution is that taxonomies are subjective — recall the drywall screws could be placed into a taxonomy organized their purpose as fasteners, or one organized by what can be picked up with magnets.

We have been describing how one type can generalize another. It is also possible for one type to classify another. When you diagrammed sentences in school, categorizing parts of speech as nouns and verbs, you were classifying types. The types “Modern”, “Eclectic”, and “House” are related to each other by a generalization relationship, but they are all related to the type “noun” by a classification relationship. In the field of models, this is referred to as *meta-modeling*. The Unified Modeling Language (UML) has a defined meta-model called the Meta Object Facility that classifies all the UML boxes and lines.

12.6 Designation

A *designation* allows you to bridge between two domains, for example between the real world and a problem domain model. A house made from bricks can keep you dry during a storm, while a box drawn with a pen on a piece of paper labeled “House” cannot. It is your intention, however, that the box in your domain model corresponds to the brick house in the real world. A designation identifies the two things and declares them to correspond.

Designation can also show correspondence between two models, for example between a problem domain model and a solution model.

You do not need to designate everything you put into the model. You should designate as few things as possible, what Michael Jackson refers to as a *narrow bridge* (Jackson, 1995). You can *define* the rest. So if you designate how your model house corresponds to your real house, you could define how the arrangement of walls in the house determines the square footage, or the taxes owed. You can think of the designations as a minimal set of variables you need to act as the base for your model, like some raw data you enter into a spreadsheet. The equations in your spreadsheet act as definitions, and they compute the rest of what you need based on the input data.

Designation is surprisingly common since computer systems are often used to keep track of what is happening in the real world. There must be a real thing and a computer representation of that real thing. Perhaps in the past you have had a difficult conversation with a clerk or customer service representative, trying to convince them that their designation relationship is wrong: perhaps they believe that you live at your former address or owe them some money. Similarly, conflating the real thing and the designated thing in the model can be a source of errors.

12.7 Refinement

Refinement is a relationship between a high- and low-detail representation of the same thing. A pencil-drawn picture of a house can be refined into a photo-realistic picture of a house. Do not dwell on whether the high- or low-detail representation is created first, because refinement is the relationship between the two representations. You could start with the low-detail version (also called the *abstract* version) and add detail, or you could do the reverse. Either way you have two representations of the same thing, one high-detail, one low-detail, as shown in Figure 12.5.

The higher-detail representation is not always the more useful one. Consider how an executive summary is used compared to the whole document, meeting minutes compared to a recording of the whole meeting, or an architecture model compared to a 10MLOC implementation.

If the two representations are of the same thing, there should be correspondences between elements in each. The roof in a sketch of a house corresponds to the roof seen in the oil painting. The collection of these correspondences is called a *refinement map*. The refinement map is not always written down because most correspondences are simple.

Comparison with other relationships

Refinement is sufficiently similar to the other relationships that it is worth comparing them.

Designation. You may have already noticed the similarities between designation and refinement. That is because refinement is a special case of designation where one model is

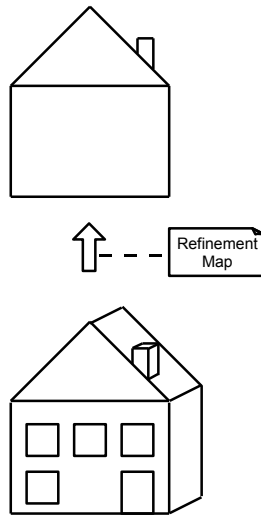


Figure 12.5: Refinement of a house

high-detail and the other is low-detail. In both designation and refinement, you are showing the relationship between two models of the same thing.²

Projection. A projection can always be automatically generated by applying the projection function, and the view usually shows just part of the model, unlike refinement that always shows the full model.

Generalization. Generalization is quite different as it is a parent-child relationship between categories.

Classification. Classification, unlike refinement, is a relationship between two different things; my house is not the same as the category named “Modern House”. Said like this, the distinction between classification and refinement is clear, but it is easier to conflate them when looking at actual models. You may slip into the idea that the low-detail model of a house is the specification and the high-detail model is the implementation. But both are specifications, just with different levels of detail.

Zooming in. Refinement is similar to the informal idea of “zooming in”, but zooming in provides a refinement of just one part not the whole model.

Refinement semantics

A complaint you may have heard about abstract models is, “I’m afraid you will put something in the detailed model that will surprise me”. This is a reasonable concern. Imagine

²Note: Not sure this is true. Designation is on elements in a model, not the model itself. With refinement, the two models correspond.

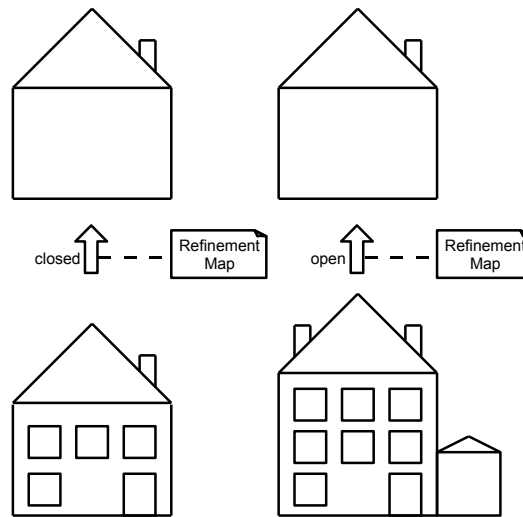


Figure 12.6: Closed semantics (left) and open semantics (right)

you show someone a diagram of a house, like the one at the top of Figure 12.5, that has no garage. A trusting person might assume that the house is not destined to have a garage. But is it wrong if you reveal a high-detail model, which now shows a garage, and a new storey? If it is not wrong, it may still be confusing. The ideas of open and closed semantics will reduce the confusion.

In refinement with *open semantics*, the refinement can introduce whatever new items it pleases. Adding a new garage or storey is fair game, as would be chicken coops and windmills. In contrast, *closed semantics* restricts what kinds of new items can be introduced by listing the kinds of items that will not change.

In the house example, using closed semantics, you might categorically restrict the refinement so that no new garages or storeys are introduced. Things you do not mention in the list are OK to introduce, such as new windows or chimneys, so you still have an opportunity to add details. A common choice is to prohibit new “top-level” items, where what counts as top-level is understood in your domain. In component and connector diagrams, you would refrain from introducing new components or connectors unless they are completely contained in an existing component — so you can add the internal design but not change the existing arrangement of components and connectors. This book uses closed semantics and recommends that you do too.

12.8 Binding

Neighborhoods and houses follow patterns where individual houses, or parts of houses, are substituted in the general pattern. For example, some neighborhoods have alleys where garages are behind the houses, while others lack alleys and so have garages facing front.

Similarly, the architectural style of a house may dictate double-hung or sliding windows. At a smaller scale, electrical outlets follow the pattern set by electrical codes.

In all of these examples, a general model is established and individual elements are bound for the placeholders in the pattern. A *binding* relationship between two models pulls in the concepts from the source model and replaces the placeholders with elements from the destination model.

Imagine you have a model with a house and a garage. You are free to place the garage anywhere with respect to the house — it could face the front or face an alley or perhaps face the side. If the style in the neighborhood is to have garages that abut the house, you could bind your house-and-garage model with a model that defines the garage-abuts-house pattern. The pattern would have three elements:

- a constraint saying the garage must abut the house
- a garage placeholder
- a house placeholder

When you bind this pattern to your model, you bind the two placeholders to the house and garage, and now also have the “abut” constraint in the model.

Explicitly writing out the details of the binding can be tedious, but the intuition is clear. When you are binding in a *pattern* (also called a *style*), you must describe what in the pattern corresponds to what in the source model. The resulting model includes all of the elements and constraints from both the pattern and the source model.

12.9 Dependency

A *dependency* relationship exists when changes to one model may cause changes to another model. For example, you can express a dependency between an estimated price for constructing a house and the current prices of raw materials.

12.10 Using the relationships

Throughout this chapter, as each relationship was described, you have seen how the relationships can be used in isolation, but their utility is clearer in context. Figure 12.7 shows all of the relationships in the context of plans for a house and garage. Each model is shown inside of an icon that resembles a folder.

At the bottom right, the house and garage model is mapped into the real world, which in this example should be quite evident. That same model also binds in the “Garage Abuts House Style”, and accordingly shows the house and garage adjacent to each other. The model is partitioned to show the garage and house separately. The model of the house is refined (using closed semantics) into a more detailed model. The detailed house model is projected to show the floorplan for the first floor with a kitchen, dining room, and living room, which are labeled R1, R2, and R3. The relationship between R1 and Kitchen is a

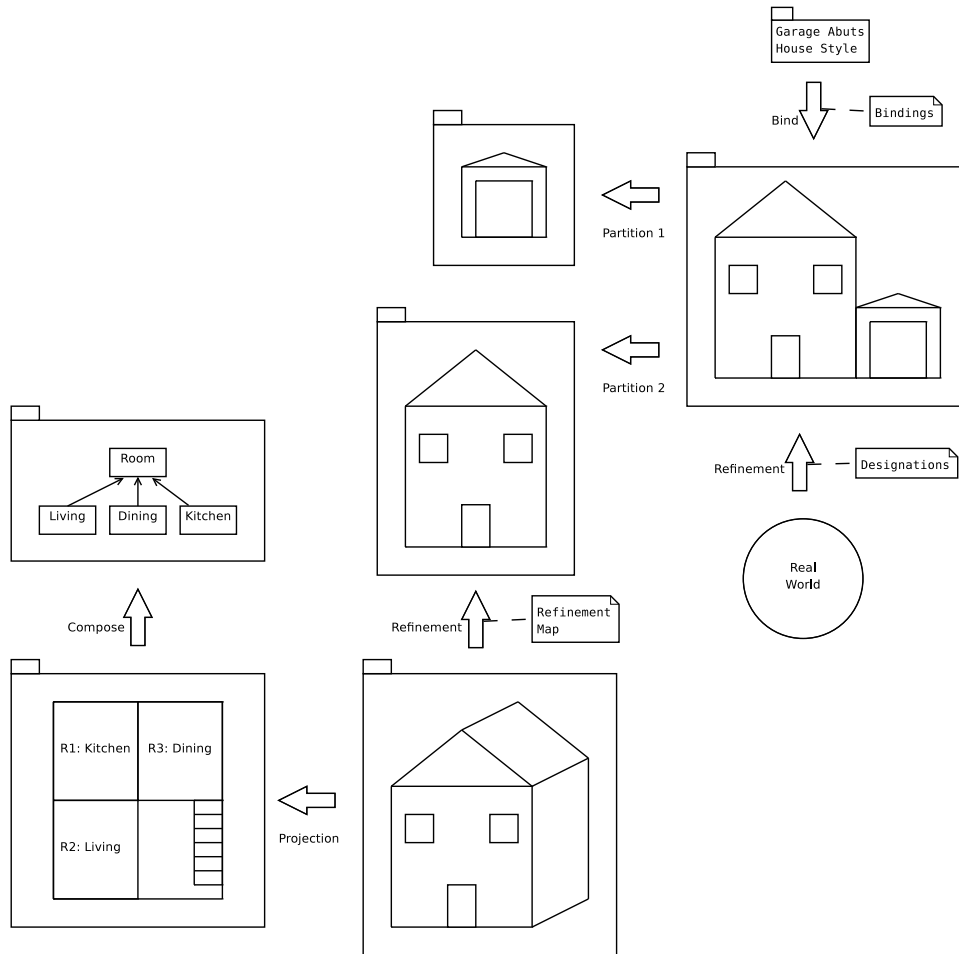


Figure 12.7: House with many relationships

classification, a type-instance relationship, where the instance is called R1 and the type is Kitchen. And this floorplan model incorporates a taxonomy of rooms, which shows that a Kitchen is a kind of Room.

It is easy to see how this diagram could be extended. If someone asks for more details on the garage, you could build a detailed model of the garage and relate it with refinement to the partition showing the garage. Similarly, additional projections could show views from different perspectives, or to enable analysis.

12.11 Conclusion

This chapter has covered relationships that you will use in building models: projection, partition, composition, classification, generalization, designation, refinement, binding, and

dependency. You have most likely already used these relationships informally. By shining some light on them specifically, you should now understand better what each is for and thereby avoid some modeling errors or confusion. You should understand that there are options you must choose: open or closed semantics, and master model or views as requirements.

The final example from Figure 12.7 shows a common occurrence: a collection of diagrams that relate to each other. Knowing the relationships from this chapter will help you put those diagrams together into a coherent and comprehensible model.

12.12 Further reading

Michael Jackson provides detailed discussions on a number of relationships including projection, partition, definition, and designation in (Jackson, 1995). Desmond D'Souza and Alan Wills base the Catalysis approach around the refinement relationship and discuss its use in analysis, design, and code. A more formal treatment of refinement is found in (Moriconi, Qian and Riemenschneider, 1995).