# The CrossModel Technique and Pattern Catalog

George Fairbanks
Rhino Research
george.fairbanks@rhinoresearch.com

April 4, 2013

## 1 Introduction

Software architects often must recover, understand, or describe the architecture of an existing system. As a community, architects usually agree that systems should be described using multiple views. However, when it is time to build those views, each architect does it his/her own way and the results are correspondingly idiosyncratic.

This document describes CrossModel, a technique for architecture recovery, understanding, and description that is repeatable and yields a set of self-consistent views. It is described as a set of over 75 patterns plus guidance on how to apply them.

CrossModel is particularly good at revealing the functional aspects of a system and adequate at revealing quality attribute aspects. This is a good fit for architecture discovery projects because most systems are Big Balls of Mud rather than optimized for chosen Architecture Drivers.

If you have never tried to recover the architecture of a system, you might think that it is a simple matter of interviewing experts and writing down your findings in a suitable notation. In practice, however, you get a set of incomplete and flawed descriptions from experts. From these vague clues, the architect must solve the puzzle: synthesize a coherent understanding of the system, recognize inconsistencies and gaps between views, and drive the discovery and creation of a complete and self-consistent model.

CrossModel makes this puzzle easier to solve by providing a mental framework to organize discovered details, techniques to discover inconsistencies and gaps, and a pragmatic process for interviewing experts. The name CrossModel evokes a crossword puzzle, which has scraps of information (clues) provided by experts that are scrutinized with the goal of making a consistent model (the puzzle).

### The problem of architecture discovery

You will often find yourself in need of a useful model of an existing system. Such a model helps you answer the questions you have, such as “How can I make it faster?”, “Can we integrate a third-party module?”, or “Why is it unreliable?” Even if you know these questions in advance, creating a useful model is not the work of a simple scribe who asks questions of the existing system’s stakeholders, developers, maintainers, etc. and writes down their answers. The obstacles are many, and include:

#### Obstacle: Views must be consistent

It’s relatively easy to produce views in isolation but fiendishly difficult to gain confidence that there are no lurking inconsistencies or gaps. An expert modeler is someone with the skill to detect such inconsistencies and gaps, but it is not a skill that is easy to teach. What can be taught is model syntax, such as UML or ADL. But our modeling languages have a limited ability to express consistency between views at the same level of abstraction (refinement) and almost no ability to do it at different levels. Consequently, the goodness of the model depends heavily on the skill of the modeler in keeping the multiple views consistent.

### Obstacle: SMEs have partial expertise

If you are able to interview a Subject Matter Expert (SME) who truly understands all the dimensions of a system then you are very lucky for two reasons: that such a person exists and that you can get his/her time. Most often, no single person understands the whole system and if such a person exists they are too busy to spend days with you scribbling models. So you cope with multiple SMEs and hope for good coverage.

The best metaphor is probably multiple descriptions of an elephant, where one SME may tell you the business market drivers for why the system needs its response times, another tells you which technology platforms were chosen and the future plans, and yet another tells you how the system fits into the workflow of the company.

These descriptions will neither agree nor will they be complete. So your primary challenge is to synthesize them, detecting inconsistencies and gaps. Not every SME is ready to accept their imperfections or your improved and more comprehensive model. So your secondary challenge is to gain consensus and rally them to support your model. Having the right answer is good; having consensus that it's right is better.

### Obstacle: Most systems are big balls of mud

Modeling by definition means eliding details. You want an understanding and description that is accurate but also of tractable size. That works best when the system has overarching themes, policies, and standards. For example, "all drivers use the plug-in API". But since this is a real system, you may find that each rule has more exceptions than you can count. That is part of the reason nobody understands the system and you have been asked to build this model.

You cannot expect architecture or models to be a silver bullet that removes this messy complexity. Your best hope is to merely expose it. (You may also help others understand the long-term price of breaking all those rules, or never creating any). It's not like taking a car to the carwash and discovering it's shiny underneath the mud. You may start washing it and discover it's rocks, dirt, and sticks all the way through.

Nor can you expect architecture drivers or quality attribute goals to appear when they have never been contemplated by the system's designers. Unless a system was designed for X, it probably is not very good at X (where X is a quality of your choice).

In the Big Ball of Mud pattern [3], Foote and Yoder note that there is a perverse incentive for those who have mastered a system's complexity to perpetuate it. Those maintainers are viewed as irreplaceable employees and treated as heroes. Keep this in mind when you interview them. Michael Jackson [5]refers to a similar anti-pattern when he describes companies who respect employees who make things seem complicated instead of those who cut through the clutter. The outcome is the same: systems that are hard to understand, even by their maintainers.

### Obstacle: Limiting scope

Recall when you learned algebra and you had problems like figuring out when two trains would meet. If you include too many details in the model, such as the color of the trains, it can still be a useful model. Omitting a necessary detail is clearly a deadly mistake. So your temptation is to include every detail that you might need. With trivial models this is manageable. But for real-sized models of software systems, over-including details is also a deadly mistake. Consider that you already have the full source code and that is a kind of model – just not a particularly helpful one for your purposes. So you have the challenge of digging in deep enough so that you don't miss essential details, but keeping the model simple enough to be tractable. This

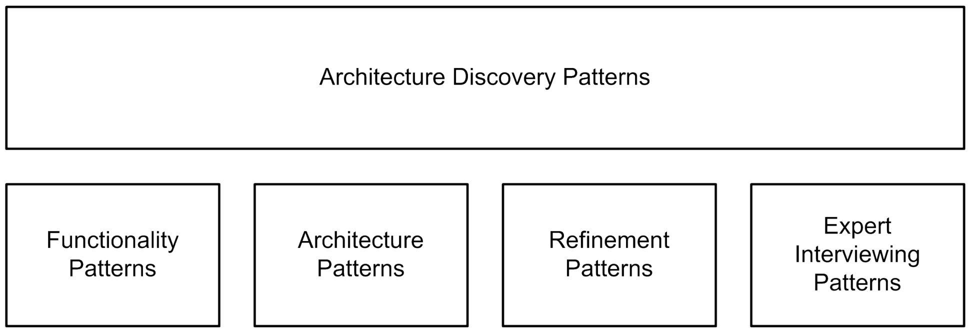

Figure 1: Layer diagram to organize the pattern categories. The only "uses" relationship is higher-to-lower.

is a constant challenge as SMEs provide you with more and more details: which details get included and which are omitted?

## Organization of the patterns

In order to describe CrossModel, the catalog of patterns has been divided into sections:

- Functionality Patterns
- Architecture Patterns
- Refinement Patterns
- Expert Interviewing Patterns
- Architecture Discovery Patterns

As seen in the notional layer diagram in figure 1, the Architecture Discovery Patterns depend on the other four groups of patterns, but not the reverse. You may find each of the pattern collections useful on its own, separately from CrossModel. The Expert Interviewing Patterns could, for example, be used anytime you are interviewing experts.

Some patterns books like the Design Patterns [4] book give each pattern a standard structure. That makes sense with a whole book and only 20 patterns. Here we have many patterns so a shorter, more casual description of each is merely intended to provide the reader with an intuition about when and how to apply the pattern. The expectation is that most individual patterns are simple and need little elaboration, but the collection of them reveals a holistic way to do architecture discovery. Furthermore, the Architecture Discovery Patterns are roughly organized as a pattern language that gives guidance on how and when to use the other patterns.

## 2 Functionality Patterns

Most architecture experts agree that functionality is relevant to architecture but they still do not give it as much attention as quality attributes. When you are faced with an unknown system and are trying to do some architecture discovery, however, the first thing you want to know is what the system does. To build a model of a system's functionality, we will borrow from Catalysis [2].

The goal is to build a self-consistent model of state (what Catalysis calls an Information Model) and behavior. The information model uses UML class diagrams and textual descriptions (i.e., data dictionaries). The behavior model includes (functionality) scenarios that consist of a series of steps and a list of actions defined with inputs, outputs, and pre/post-conditions. (Although we think about all that, often you will just list the actions without all that detail). Each term referenced in the actions must be defined in the information model.

State and behavior are two views of a single, coherent functionality model. So there is a challenge when writing down one view or another: you must keep the views consistent. If the specification for the Debit action says it removes money from a Bank Account, then we expect to see Bank Account in the model of state (the Information Model). As a modeler, you are a human CASE tool with the job of keeping these views consistent.

Each action in the model should cause changes to state. For example, we should have an attribute "balance" on our Bank Account, or else we might not see a change to the information model before and after the Debit action. Sometimes this is unsuitable, and in those cases you can try an action invariant (e.g., manage inventory –> maintain stasis/relationship between predicted orders and items in inventory).

| **Model Defect Type** | **Consistency** | **How to Detect** | **Pattern to Fix** |
|---|---|---|---|
| Synonymous Terms/Actions | Between views | Audit for synonyms | Rationalize Synonyms |
| Ambiguous Terms/Actions | Between views | Requires modeler's insight | Eliminate Ambiguity |
| Under-constrained Types | Between views | Generate snapshots from information model | Constrain Information Model |
| Mixed Abstraction Levels | Single view | Requires modeler's insight | Single Level of Abstraction |
| Repetition | Between views | Requires modeler's insight | Refactor to Standard Pattern |
| Syntactic Gaps (e.g., Missing Terms/Actions Among Views) | Between views | Walkthrough comparing information and action models | Fill Syntactic Gaps |
| Semantic Gaps | Model-World | Walkthrough comparing views, looking for gaps in action input/outpus, pre/post. | Fill Semantic Gaps, Build E2E Scenario |

Figure 2: Common defect types in functionality models

What is meant by consistency is that there are no defects in the model with respect to its views. The model itself could be wrong (for example, an expert tells us that the moon is made of cheese), but consistency says that two different views inside that model do not contradict each other. Figure 2 lists several common defect types along with patterns for fixing them. Of course we prefer it if our models agree with reality too, so there are some patterns to help us catch “moon made of cheese” defects.

Functionality models are not concerned with quality attributes like how fast a system runs, how reliable it is, or how secure it is. But you will find that functionality is important to know about and we can assign functional responsibilities to modules, components, and hardware once we introduce architecture models in the next section, Architecture Patterns.

## Patterns

**Build Functionality Model** A functionality model consists of: Scenarios, Action model, and Info model. You may build all three simultaneously. Another option is to focus on building scenarios, but to keep a preliminary list of types and actions on the side as a precursor to full info and action models.

- **Build Scenario** Collaborate with SME to write down a scenario.
- **Build Information Model** Using an existing scenario, collaborate with SME to populate an information model using the terms mentioned in the scenario. Collaborate with SME to fill in relationships (associations) between the terms.
- **Build Action Model** Using an existing scenario, collaborate with SME to populate an action model using the actions mentioned in the scenario. You can optionally write down pre- and post-conditions, but will think about them even if they are not documented.

**Improve Functionality Model Quality** Once you have the first draft of a functionality model, you will want to improve its quality. These patterns can be applied to make the views consistent and to find missing pieces.

- **Constrain Information Model** Information models are constrained by their list of types, associations, and multiplicities but this is often insufficient. The model can be further constrained by arbitrary invariants.
- **Fill Syntactic Gaps** This is a mechanical, *syntactic* activity, so the gaps we are talking about do not require an understanding of the domain. The multiple views (scenarios, info model, and action model) share a common meta-model that requires them to be consistent. For example, each step in a scenario is an invocation of an action. You can detect gaps by finding actions or terms that are present in one view but not in another.
  - **Fill Information Model Gaps** Mechanically inspect the scenarios and action model. Every term appearing in a scenario, action parameters, or action pre- and post-conditions must also appear in the information model. Add missing terms to the information model as entities, attributes, or named associations. Some judgment is required because you will find terms that are used to make scenarios read clearly, but are not important enough to add to the information model. Mechanically inspect the information model. Ensure that it does not contain anything not mentioned in actions or scenarios. When gaps are detected, add the terms to steps in existing scenarios or actions, or as a last resort add new scenarios.
  - **Fill Action Model Gaps** Mechanically inspect the scenarios. Since each step in a scenario is an invocation of an action, you may find steps that have no corresponding action. When gaps are identified, synchronize the models by adding the needed actions to the action model. Mechanically inspect the action model. Every action should appear in at least one scenario step. When gaps are detected, add steps to existing scenarios or as a last resort add new scenarios.

**Fill Semantic Gaps** This is a *semantic* activity (not mechanical) requiring collaboration with a SME. You, as the modeling expert, will ask leading questions to prod the SME to identify missing actions, terms, etc. Unlike the earlier patterns that are based on a hunch, you can use these patterns to mechanically ask questions of your SME, but ultimately that SME is the one who must identify the missing pieces.

- **Fill Action Input/Output Gaps** Working with the SME, walk through each scenario step and ensure that the corresponding action's input parameters and outputs (return values) are accurate within the scenario.
- **Fill Action Pre/Post Gaps** Working with the SME, walk through each scenario step and ensure that the corresponding action's pre- and post-conditions are accurate within the scenario. This can be done by just talking about what the conditions must be before and after the action. Often you will discover that the info model is not detailed enough to express the conditions. You may also find that the info model shows no change after the action, which is another clue that the info model should be improved.
- **Fill Lifecycle Gaps** These are gaps relating to lifecycle of info model terms. For example, how is an account created? You can work through your domain model asking about the lifecycle of each item (term, association, etc.).
- **Build E2E Scenario** Building an end-to-end scenario helps ensure steps are not missed (e.g., in setup, teardown, lifecycle).

**Improve Functional Consistency** This is a *semantic* activity (not mechanical) requiring collaboration with a SME. You, as the modeling expert, will ask leading questions to prod the SME. Often you will have a hunch that something is not quite right (e.g., two terms in your model are synonyms) and you must clean up the model, for example by distinguishing the synonyms or documenting clearly how they are not synonyms.

- **Rationalize Synonyms** SMEs will often use synonyms. It is best if your model uses one of those synonyms consistently and has documentation listing the others. For example, an SME (or different SMEs, or different divisions within the company) could use the terms "account" and "user record" as synonyms. Choose one and use it consistently. When you define it in the info model, list any synonyms there.
- **Eliminate Ambiguity** SMEs will often use a single term (or action) for multiple distinct ideas. It is best if your model uses different names for those distinct ideas, even if you must invent vocabulary. This is the opposite of the synonym problem and much harder to detect.
- **Refactor to Standard Pattern** Recognize a pattern and apply it consistently across models by refactoring model. Pattern may be a sequence of actions, terms, etc. Solution is to use consistent language each time, not synonyms, so that readers recognize the pattern. Consider documenting the pattern explicitly.

# 3 Architecture Patterns

The core ideas for this section come from the architecture community, especially the SEI. The patterns emphasize the creation of multiple architecture views including representative views in the module, runtime, and allocation viewtypes.

The set of architecture views is open-ended. There are some standard ones but on most interesting systems you will see some uncommon views that provide insights for that system in particular. Consequently, the meta-model for architecture models is less clear than with functionality views. The expertise and understanding of the architect or modeler will have a big impact on the quality and usefulness of the models.

As with Functionality Patterns, we see patterns relating to building and improving quality.

## Patterns

**Build Architecture Model** Most useful architecture models eventually include at least one view from each of the primary viewtypes: module, runtime, and allocation. If you have not yet looked at each viewtype, you should consider it seriously.

- **Build Module View** You can learn a lot about a system by discussing its modules. But details about code can also be a tarpit, cluttering your model and obscuring insight.
- **Build Runtime View** One way to do this is to start with an existing functionality scenario, augment it to include runtime architecture details (such as systems used, connectors used, ports used), build the runtime view, then discard the additions to the scenario.
  - **Build System Context Diagram** One common variant of the runtime view is to build a System Context Diagram showing the system as a single runtime component and the external systems it interacts with.
- **Build Allocation View** Allocation views are often forgotten but provide key insights into quality attributes (performance and security in particular).
- **Build Spanning Views** Views that span viewtypes help glue together the views, showing how they relate.
  - **Build Actions-on-Allocation View** A view that is primarily an allocation view but also contains actions allows you to easily see both behavior and runtime/allocation elements. It can be used to walk through your understanding of how things behave.
- **Build Custom View** Build a new view to clarify, based on discussion of scenarios, pre-existing or domain-specific views.

**Improve Architecture Model Quality** Just like functionality models, views in architecture models must be kept consistent, but the architecture views should also be consistent with the functionality views too.

- **Reconcile Functionality and Architecture** One way to catch defects is to ensure the functionality and architecture views are consistent.
  - **Runtime-Functionality Walkthrough** With a SME, walk through a normal functionality scenario and animate it across the runtime architecture view. This can help you identify responsibilities of the systems and may expose gaps: systems not yet in your model.

- **Allocation-Functionality Walkthrough** With a SME, walk through a normal functionality scenario and animate it across the allocation architecture view. This can help you identify connectivity between hardware, capacities of hardware, and even rationales for why systems were allocated to particular hardware (which helps you elicit quality attribute requirements or goals). It may also expose gaps in your allocation model: hardware and links that you have not modeled or are lacking details (bandwidth, operating system, etc.)

**Cross-Cutting Analysis** If you have two architecture views and are unsure how to check them for consistency, look for ways to analyze their intersection. Often you can do this by creating a new view (often a throw-away temporary view) that cross-cuts other views. For example, asking how deployments or upgrades work is a way to crosscut module, runtime, and allocation views.

- **Technical Rationale** Ask the SME why technical choices were made. Examples include why languages, operating systems, architectural patterns, or server configurations were made. Exactly which details you ask about is dependent on what questions you need your model to answer.
- **Golden Sources** In collaboration with a SME, determine for each significant type in the information model which hardware or component is the golden source of that type. A “golden source” is the source of record, the canonical information. The list of users might be kept in many places, but usually one of the places is the golden source. This can provide insights as to workflows that span systems/components, help you reason about latencies, etc.
- **Deployment Scenario** With a SME, walk through how software is upgraded or deployed. You may not want to keep this scenario, but it will force everyone to evaluate module, runtime, and allocation views of the system and may expose gaps or inconsistencies.

## 4 Refinement Patterns

Before we can solve a problem we must understand it. We build models so that we can better understand the essence of the problem. When we encumber our models with unneeded details, we are less likely to understand the core problem and therefore less likely to solve it. So we must not only build consistent models, as the previous two sections of patterns have taught us, but also build simple models.

Refinement is a relationship between a high-resolution and low-resolution model of the same thing. For example, a report abstract and the report body exhibit a refinement relationship. No one would consider either a complete replacement for the other – instead they serve different audiences and purposes. Refinement enables us to have our cake and eat it too, at least in that we can have both a simple, clear model and a detailed, complex model so that we can use whichever suits our purposes.

When modeling software systems, we often want the "ten thousand foot view" that summarizes the system as well as a more detailed view that can be used to analyze more specific engineering questions. Refinement lets us have both.

Note that while most refinements are one-to-many, some are few-to-many. This is particularly noteworthy for our technique of expanding a single scenario step into many, which works most of the time. Sometimes, however, it's necessary to refine N scenario steps into M.

### Patterns

**Blackbox-Whitebox Refinement** Black/white boxes are a technique used in every engineering field to to hide and reveal detail. The basic idea is that one view shows you a bunch of black boxes but their inner workings are hidden. A different view shows those same boxes, but this time as white boxes and their inner workings are revealed. This technique is used in software modeling in many places and the "boxes" can be modules, components, hardware, scenario steps, etc.

**Single Level of Abstraction** Refactor the model to single level of abstraction. Regardless of what interesting details you discover, it is best if every model is at a single level of abstraction. It is fine to dig into detail as needed, and refinement gives us the ability to have one model with high abstraction and a second model that elaborates the detail in the area of our interest. Some examples follow:

- **Hide Action Details** (e.g., set of actions –> one action)
- **Hide Type Details** (e.g., set of types –> one type)
- **Hide System Details** (e.g., set of systems –> one system)
- **Refine Scenario** Scenarios in particular tend to accumulate details at mixed levels of abstraction. Here are some patterns to clean them up and achieve a single level of abstraction.
  - **Split Scenario** One scenario with mixed detail levels becomes two different scenarios related by refinement, much like a blackbox version and a whitebox version.
  - **Refine-Up/Down Scenario** Generalize and simplify scenario details via indentation. A single step of a scenario is refactored into a single high-level step and one or more detailed (refined) steps. The refinement is revealed using indentation.

**Reveal Intent** Refinement can be used to highlight intent. Not every refinement does highlight intent, but when you recognize intent, you can use refinement to make it visible. Often times the intent will not be visible to the SME you are interviewing, for example because the SME does not know as many architectural or design patterns as you do as an expert modeler.

- **Combine Actions to Reveal Intent** (e.g., series of actions and data movement –> replication)

**Generalize Types to Reveal Intent** (e.g., set of files –> staging area)

**Generalize to Tech Neutral** The jargon and mechanics of a technology can obscure intent. Refactor the model to tech-neutrality via refinement to make the intent more visible.

**Invent Vocabulary** Simplify a model by introducing new, generalized concepts. This is risky because you would prefer if your model matched the existing system as best possible, including any terms currently in use. But sometimes the terms in use do not clarify a complex idea or activity, and providing a new name for it will.

## 5 Expert Interviewing Patterns

When you build a model, rarely will you be the Subject Matter Expert (SME). That means you must interview SMEs so that you can build a good model. It's not as simple as asking them how things work and writing it down, like a scribe. Instead, it's a collaboration where you bring your modeling and analytical skills and they bring their expertise.

These patterns are not as technical as the ones that come before because they are my bag of tricks on how to engage SMEs productively, rather than a technical modeling formalism. Engaging with SMEs is not a math problem where the only thing that matters is the answer at the end. For better or worse, the value of your model will be judged, implicitly, every minute you work with the SMEs. They are judging whether they should bother with this effort or feign an excuse to get back to "real work". Your ability to demonstrate value throughout the process, not just at the end, depends on how well you facilitate the sessions and the progress in the model you demonstrate.

### Patterns

**Passive Facilitation** While you are not strictly silent or passive, these patterns emphasize not imposing your will on the situation and instead working with what is there and how the SMEs choose to describe it. You will recognize that you are documenting ambiguities, contradictions, gaps, etc., but with these patterns you want to absorb first and fix defects later.

- **Let Experts Talk** Capture what they say verbatim. Plan to fix it up afterwards, maybe long after. For some SMEs, you may only get a short amount of their time and breadth is more important than depth or accuracy. You may be able to use less knowledgeable SMEs to help you fix the model.
- **Use What Exists** Adapt existing diagrams and documents into your model. These are almost always cartoons (meaning not intended to be formally accurate) but can be a good starting point. Some diagrams (deployment and network diagrams for example) tend to be more accurate than others (runtime architecture). Companies often have training materials for new hires that work well.

**Active Facilitation** With these patterns, you are imposing your will so that the model is higher quality even as you are documenting information from SMEs. Instead of just writing down something that you suspect is a defect, you ask the SME immediately. Your skill and judgment will determine how well these patterns work.

- **Propose Unmentioned** If you have a hunch that something is missing (actions, types, actors, etc.) then raise it as a question. Validate with SME.
- **Propose Refinement** If you notice a model has a mixed level of abstraction, propose that some details be kept while others are withheld until a more detailed model is built. Note that you run the risk of SMEs withholding details that end up being crucial, so use with caution. You can always follow the Let Experts Talk pattern then refactor afterwards.
- **Educate Experts** If this is a big modeling project, it may be time-effective to educate some of the SMEs about your modeling process, modeling techniques, and the idea of view consistency via a meta-model so that your time with them is more productive.

**Throw-Away Scenarios** Most scenarios exist only on whiteboard. Scenarios are cheap to build once but expensive to maintain, so choose only a few scenarios to keep around. But use scenarios as your primary way to interact with SMEs because they are a technique that engages even non-modeling-expert SMEs and yet yields good detail for your models.

**Tangible Starting Point** We are all more accurate when we can consult our references. Asking an SME to list all of the systems in the company is likely to yield an incomplete list. But suggesting that the SME start with tangible artifacts (e.g., version control repository, organization chart) will help him/her give you a complete list.

**Gain Consensus** You are unlikely to have much impact if you build the perfect model but all the SMEs reject it. A model is an implicit statement that "this is the essence of what is going on", at least from the perspective of the questions you want the model to answer. So when your model deviates from the way the SMEs describe the system, it is best to explain why and to try to convince them that the model is accurate.

**Many Eyes** Cross-check model accuracy/completeness with multiple SMEs. Rarely will one expert be all-knowing.

## 6 Architecture Discovery Patterns

The patterns so far have described how to build individual models and interview SMEs, but not yet how to build a model of an existing system. The patterns in this section rely on the earlier patterns and show how they can be productively used together.

When setting out to document a system, it's important to know when to stop. You will invariably discover more details than you have time to include in a model, and including too many details may be counterproductive anyway. So your first task is to be clear about your modeling goals. I phrase this as "question first and model second", meaning that you should write down what questions you need your model to answer, then proceed to build that model. Should your model include the detail that some of the code is written in a proprietary language? There's no way to tell unless you know which questions you want the model to answer. If you jump straight into modeling, you may build an interesting model but it's unlikely to have the right details and structure for your particular questions.

You may be able to infer the questions from what are called "business drivers". These are things that management talks about, like the potential upcoming changes to the company, nightmare business situations, or past problems that management is trying to avoid. If they are considering merging with another company, they will likely want your models to help answer questions about the merger.

Consider who will be using your models. Certainly you, and you have a great tolerance for intricate models and notation. But you'll probably have to interview SMEs and explain your models to them. Do not be surprised if sections of your models reappear in all kinds of unexpected places, because they are the best expression of the system. So it's probably best to limit the model notation to a bare minimum. You will think about pre- and post-conditions on actions but writing them down is both time-consuming and likely to scare off others. So stick with simple notations and strive for a simple model.

The metronome of the architecture discovery patterns is the cadence of expansion and contraction. There are times when you need to grow your model and collect information from SMEs. Other times your focus is on re-working what you have into a form that facilitates analysis. It's like the old saying about computer security that the most secure computer is one that's turned off. Before you write anything down, you don't have to worry about errors in your model. But of course you have to add details! Then you have to fix everything back up so it's useful. But you find that your repairs inspire questions for the SMEs, which in turn causes more defects for you to later fix up.

It's useful to focus on the goals of expansion and contraction separately. The goal of expansion is to learn more and feed the model. You will write down whatever you lean and accept the defects. Contraction, on the other hand, has two goals: a concise model (i.e., simple) and consistency between its views. In practice you will work on both simplicity and consistency at the same time.

One way to think about architecture discovery using CrossModel is that it's a game with a goal of reducing the number of defects to zero. You let in details and the concomitant defects, then proceed to fix up the defects with the model. You repeat this until your model seems to be in good enough shape to answer your questions.

Some of the defects you encounter will be syntactic and you could imagine a CASE tool helping you catch and fix them. Others require you to understand the system you are modeling, not just notation and syntax. As you build models, don't forget that not every defect has to be fixed the hard way. If you discover a deep pit of detail that could cause lots of defects, ask if you really need to dig into it, or if you could simply omit it. That is, you could build a less detailed model which is correct "as far as it goes", which could be perfectly fine to answer your questions.

The patterns here can be seen as a pattern language that coordinates or sequences the application of the other patterns. Its basic structure is (1) choose scope, then (2) alternate between expansion and contraction.

## Patterns

**Choose Model Scope** Architecture Discovery is not one-size-fits all, so you must choose the kind of model you want to build. It is best when the people paying the money get what they expect, so listening to what they call "business drivers" is worthwhile, and you should add your engineering judgment to that.

- **Learn Business Drivers** Learn or choose the business drivers that motivates doing any of this work (discovery/modeling). Some examples: We want to understand system sufficiently to add a new feature, we want to change technology; we want to make software development more efficient. Often the business driver is not technical, such as the desire to pass a due diligence audit or enter a new market. Technology plays a big role in those decisions so a suitable model can help inform management decisions.
- **Choose Questions to Answer** Choose the exact questions the model should answer. For example, how can we: calculate performance estimates, explain architecture to new hires, or move to a new technology? As discussed, it is easy to build an accurate model that answers the wrong questions. For example, a model that predicts train arrival times is useless for estimating the financial depreciation of running the train network.
- **Choose Document Template** To ensure that you get what you want, you may create an outline of the final deliverables before you start. That way you will be prompted to fill in each section.
  - **Choose Target Refinements** Before you begin modeling, you may choose which refinements you want. For example, you may want a tech-neutral view, a blackbox and a whitebox view, or a system context view.
  - **Document Placeholders** When scoping the model, introduce placeholders into your outline or template, for example for desired views.
- **Scope Model to Time Constraints** Only build the models that you have time to build. This means scoping your ambitions to the available time.

**Alternate Expansion and Contraction** Expanding and contracting is the key to incrementally building a good model as you learn from SMEs. During expansion, you introduce details into models. During contraction, you remove unwanted details, refactor for simplicity (including using refinement), and make views consistent.

- **Expand Model** Expansion applies existing patterns (for example: Tangible Starting Point, Use What Exists, Fill Lifecycle Gaps) with the goal to: Learn more and feed that into model. During early learning, you will write down what SMEs say without much editing (see: Passive Facilitation). Later, you are more actively involved in screening out potential trouble before you write it down (see: Active Facilitation).
  - **Create Inventories** One way to learn about the full scope is to create inventories, often from a Tangible Starting Point. For example, have SMEs list sub-actions, systems and DB/datasets, hardware and platforms, languages, actors, or teams within company. Walk through the lists with SMEs to ensure that your model covers needed details.
  - **Build Coverage Scenarios** Build or augment existing scenarios to cover all actions, systems, hardware, languages. Use in conjunction with Create Inventories and Throw-Away Scenarios to quickly skim through inventories yet avoid scenarios an upkeep burden.
  - **Drive Towards Questions** When pulling new details in, you sometimes just Use What Exists, but also actively seek out details that seem relevant to the questions in scope.

**Contract Model** During contraction of the model, your goal is to minimize the number of defects in your model (see figure 2, plus architecture model defects). Oftentimes this is an activity you do between sessions with SMEs, then validate your updates (the revised model) when you next meet with them. Contraction is a kind of refactoring where you rework the model into a state of improved quality (fewer defects). Throughout, you strive to achieve Simple Models.

> **Contract to Improve Quality** Apply patterns to improve model quality and reduce defects. See: Reconcile Functionality and Architecture, Cross-Cutting Analysis, Fill Syntactic Gaps, Fill Semantic Gaps, and Improve Functional Consistency.
>
> **Contract Model via Refinement** Use refinement to make the model simpler or clearer. See: Blackbox-Whitebox Refinement, Reveal Intent, and Single Level of Abstraction.
>
> **Simple Models** Strive for simple models, since those are the kind that let your brain do the work of analysis. A cluttered or confusing model inhibits “aha” moments and insight. Furthermore, models are used not only by you (the expert modeler), and keeping them simple makes it easier for others to use them, and to tell you about defects they find.
>
> > **Remove Model Details** You are striving not for a complete model, but one that minimally answers the questions you wish to ask. So remove details that are unrelated to the business driver / questions you chose.
> >
> > **Choose Kept Scenarios** Scenarios are a workhorse for interacting with SMEs but a burden to keep updated. So choose which scenarios to keep short/medium/long term. End-to-end (e2e) scenarios (see: Build E2E Scenario) are good candidates.
> >
> > **Notation Subset** Choose a subset of the modeling language notation (UML, ADL, etc.). This allows you to quickly educate others about your models. My UML subset includes only: types, attributes, associations, association multiplicities, named associations, invariants.

## Example applying patterns

In order to better envision how you could apply this pattern catalog, here is an example sequence of applying the patterns. Every architecture discovery effort is different because each one has different questions it asks of its models, has different SMEs to interview, etc. But this concrete sequence will give you an idea of how you might want to apply the patterns.

Note that in most engagements you will get the SMEs in the order they are available, not the order you want to build the models in. So you may well start with architecture models and back-fill the functionality. And within any given kind of model, you may find that one SME is available now but the others you must interview are on vacation this week. So in reality you will proceed according to what is available.

- Project scoping
  - Patterns: Learn Business Drivers, Choose Questions to Answer, Choose Document Template
  - You now understand what kind of model you want to build, so you are less likely to go down rat holes and waste time.
- Expansion 1: Functional views
  - Patterns: Build Functionality Model, Passive Facilitation, Tangible Starting Point, Throw-Away Scenarios
  - Now you have a draft functional model obtained by interviewing SMEs but it has lots of defects.

- Contraction 1: Fix functional views
  - Patterns: Fill Syntactic Gaps, Fill Semantic Gaps, Improve Functional Consistency, Active Facilitation, Gain Consensus
  - Defects have been largely removed from your functional model with the help of SMEs. You have convinced the SMEs (and other stakeholders present) that you are on the right track and this process is resulting in a helpful model.
- Contraction 2: Add refinement to simplify model
  - Patterns: Blackbox-Whitebox Refinement, Single Level of Abstraction, Reveal Intent
  - Before this refactoring, your scenarios and perhaps other views were a mixture of high- and low-level details. The refactoring enabled you to better achieve a single level of abstraction in each view.
- Expansion 2: Architecture views
  - Patterns: Build Architecture Model, Tangible Starting Point
  - Starting from existing documents and elaborated by interviewing SMEs, you build out architecture views of the system.
- Contraction 2: Fix architecture views, reconcile with functional views
  - Patterns: Reconcile Functionality and Architecture, Architecture patterns, Cross-Cutting Analysis
  - Your model should now have both architecture and functional views, all of which are reasonably consistent. Since you started by scoping the activity and writing down the questions the model must answer, your active facilitation should have driven the conversation to include needed details. This is just the starting point and your model will still need your attention and imagination before it is helpful, insightful, and answers the questions you ask of it.

A criticism of this example is that it ignores the critical element your analysis and judgment plays throughout the engagement. You will be steering the conversation and model so that it becomes what you need, rather than just a junkyard of details. Following the CrossModel patterns gets you closer to that goal but you will need to be an active participant and your skills will have a big impact on the quality of the outcome.

# 7 History

The ideas in the Functionality Patterns and Refinement Patterns of the CrossModel technique are directly traceable to Catalysis by D'Souza and Wills [2] and have been practiced in the Catalysis community for fifteen years across many companies, domains, and architects. Meta-models for creating consistent behavior and terminology models trace back further to formal modeling languages including Z and VDM. The ideas in the Architecture Patterns are shared across the architecture community but many originate with SEI authors. Many ideas no doubt are present in other communities but I have acknowledged here the source from which I learned the idea.

CrossModel as a whole, including (1) the generalization of the Functionality Patterns to architecture modeling, (2) the Expert Interviewing Patterns, and (3) the expression of these ideas as a pattern catalog and guidance, is a novel synthesis. The CrossModel technique has been applied successfully at several companies, but so far only by me. This is the first public documentation of the technique.

## Related work

Most architects have performed and architecture discovery or reconstruction of an existing system. The literature on how to do this is relatively slim. Here are a few references (this is not a proper literature survey):

- Ruth Malan:
  http://www.ruthmalan.com/ActionGuide/VisualArchitectingActionGuideToC.html
- Effective architecture sketches, Simon Brown:
  http://www.codingthearchitecture.com/presentations/wicsa2012-effective-architecture-sketches
- Rebecca Wirfs-Brock:
  http://www.wirfs-brock.com/art_of_design_story.html

These do not seem to have the same intent as CrossModel. In particular, CrossModel is a process for not only building multiple views but includes patterns for driving those views to be consistent. The architecture community tends to focus on quality attributes over functionality, yet CrossModel has a strong functional flavor to it. This may make it more suitable for architecture discovery, since projects that have been evolved for years may not have any clear quality attribute goals or drivers.

# 8 Appendix: Book excerpts

Reproduced from the book *Just Enough Software Architecture: A Risk-Driven Approach* by George Fairbanks (2010), with permission of Marshall & Brainerd. These are included to provide context and additional detail on functionality modeling.

## 8.1 Functionality scenarios

*Functionality scenarios* describe the behavior of a system. In other architecture models, the system is described as a collection of components, modules, ports, interfaces, allocation elements, and so on. Functionality scenarios tell a story of how those elements can change over time and interact with each other. For example, a component assembly of the library system, like the ones shown earlier in Figures 3 and 4, only shows which component instances exist, not their behavior. A functionality scenario can describe how that component assembly model, or another model, changes over time. Functionality scenarios can be written textually, as in figure 5, or they can be written graphically, as a UML sequence diagram.

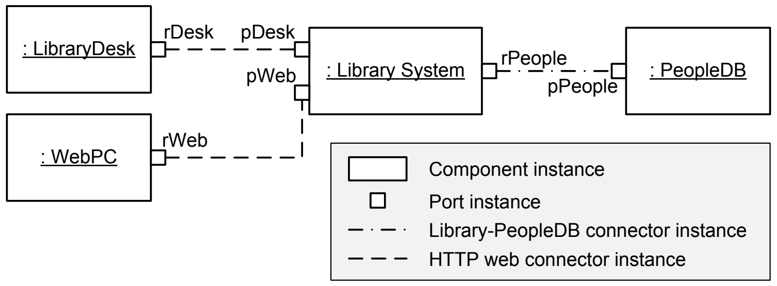

Figure 3: A system context diagram of the Library System. A system context diagram is a kind of component assembly that shows the system to be built (here, the Library System) and the external systems it connects to. The Library System component instance shown here is refined in figure 4.

figure 5 shows an example functionality scenario of the life of a copy of Moby-Dick in a library. The scenario shows a single legal trace of behavior through a model, but it cannot describe every possible behavior. For example, this functionality scenario does not say what happens when borrowers lose the copy that they checked out.

*Use cases* are another popular way of describing behavior. They are largely equivalent to functionality scenarios, but there are some important differences. Use cases are activities that are high-level and visible to the users of the system. Use cases are often defined to be accomplishing a goal of an actor outside the system, so internal system activities would not count as use cases. Where functionality scenarios are a single trace of behavior, use cases can include variation steps that allow them to describe multiple traces. Because of these potential differences, this book uses the term *functionality scenario* to describe traces, but so long as you are clear about the possible misunderstandings, you may call them *use cases*.

Functionality scenarios and quality attribute (QA) scenarios, despite the similarity in names[1], are quite different. QA scenarios are similar to a single step in a functionality scenario. The term *quality attribute scenarios* comes from [1]. The term *functionality scenarios* (or just *scenarios*) comes from [2], whose use of scenarios inspired the approach and techniques presented here. When it is clear from context, you can refer to functionality scenarios just as *scenarios*.

**Structure.** Functionality scenarios are easy to read because of their story-like quality, similar to fiction, but a useful scenario is non-fiction. It is structured and has checkable references to other models. Step 5

[1]This book refrains from changing existing terminology because it seems to be the lesser of the evils.

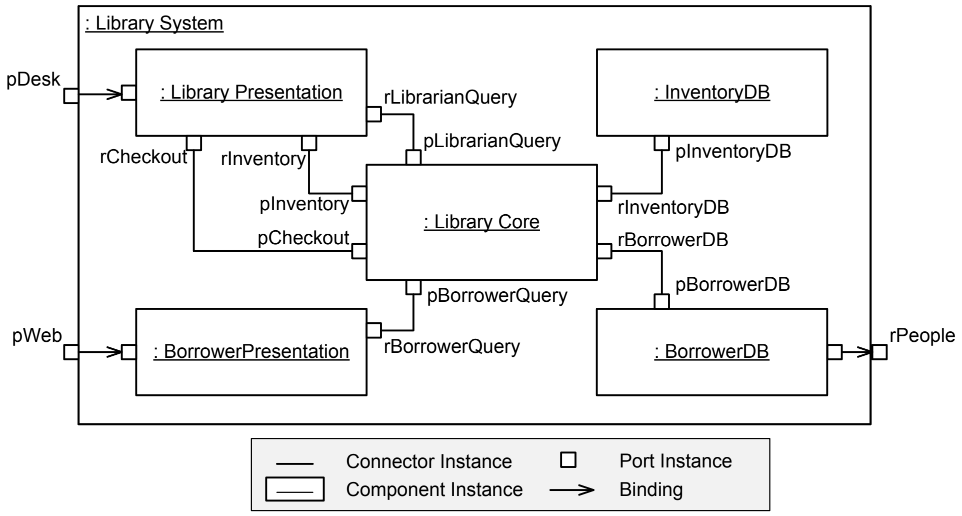

Figure 4: A component assembly of the Library System component instance. External ports are bound to internal ones. It is the same component instance seen in figure 3 and this diagram additionally shows its internal details.

in the scenario above cannot be "Larry the librarian conjures elves from the computer system / the elves tie up Bart" because that (presumably) is not something the Library System can do, however fun it might be to build that system.

In this case, the story is about libraries and books, not elves. We know that because the other design models define the *vocabulary* we can use in this story, and those models talk about books, but not elves. The other design models further constrain the *actions* in the story: actions like adding copies and checking out copies, but not tying up library patrons. To achieve this connection with the other design models, a functionality scenario consists of the five parts shown in figure 6: target model, scenario name, initial state, actors, and steps.

The meta-model of a functionality scenario is shown in figure 7, and it formalizes the description above. It shows that each functionality scenario has one target model and a sequence of steps. Each step has an actor who initiates it and a single action that is invoked. Each step transitions a model from a begin state to an end state. Actions belong to a model and reference some of the model's elements.

What is an *action*, exactly? Each model has ways that it can be changed, and actions are the mechanism for manipulating the model. For some models, the actions are obvious. When the scenario applies to modules with interfaces, actions are the operations that are defined on the interfaces. Component assemblies work similarly because ports define the behavior. But other times the action is less clearly defined and more abstract. For example, a scenario could discuss reconfiguring a router, which is a step done by a person. A developer compiling code could also be an action, or starting up a new data center. When the model already has clearly defined actions, it is easy to write precise scenarios, but when the model is less formal then writing a good scenario requires more discipline.

**Two-level scenarios.** Scenarios should be written at a consistent level of abstraction. Following a structure enforces this, because a scenario refers only to elements of the target model, which is itself presumably at a consistent level of abstraction. This is neat and tidy, but leads to the difficulty of understanding how scenarios at one level of abstraction relate to those at a different level. For example, a scenario for the Library System could refer to the system and its publicly visible operations (as in figure 5), but not to the Library System's internal components. Separately, another scenario could be written that refers to the subcomponents inside the Library System.

To see how two scenarios at different levels of abstraction connect, the two can be merged and related.

Name: End-to-end copy of Moby-Dick
Initial state: Larry is a Librarian; Bart is a Borrower
Actors: Larry, Bart
Steps:

1. Larry lists all Books about "fishing" / No matching books are found.
2. Larry adds a Copy of the Book "Moby-Dick" by Herman Melville to the Library. A record of the Book is also added.
3. Larry lists all Books by Herman Melville / "Moby-Dick" is returned.
4. Larry lists all Books about "fishing" / "Moby-Dick" is returned.
5. Larry (with Bart) checks out the Copy of "Moby-Dick" to Bart. Its due date is set to 6 September.
6. Larry lists who last checked out the Copy of "Moby-Dick" / Bart.
7. Larry lists the Copies currently checked out by Bart / Copy of "Moby-Dick".
8. Larry (with Bart) returns the Copy of "Moby-Dick" to the Library.
9. Larry removes the Copy of "Moby-Dick" from the Library.

Figure 5: An end-to-end functionality scenario showing the initial addition of a book copy to the library, it being checked out, and eventually removed from the library. It applies to the system context diagram for the Library System shown in figure 3.

figure 8 shows the first couple steps of the earlier scenario plus a second column that describes what happens to the internal subcomponents. The first column is the more abstract scenario that cannot see the subcomponents inside the Library System. The second column is the more detailed scenario that explains how the subcomponents accomplish the action described in the abstract scenario.

**Generalizing functionality scenarios.** A functionality scenario is just a trace, meaning that is it just one possible legal sequence involving the model, the actors, and the actions. You may need to build a general model that expresses every possible trace. For example, general models are useful for documenting or analyzing protocols between components. If you are publishing your component for use outside of your group then you may want to provide documentation that goes beyond a few example scenarios.

There are many options for describing general behavior, including state diagrams, activity diagrams, and sequence diagrams. Note that sequence diagrams have traditionally been used to describe traces, but they can be augmented with annotations like "loop up to five times" to generalize the trace.

General behavior models can be difficult and expensive to build. Getting them somewhat right is easy, but 100% right, including exceptional paths, is hard. You may need them if you want to rigorously analyze a protocol or provide exact documentation. A general behavior model should be accompanied by at least one scenario, if only because of the analysis benefits from animating the scenario. This book recommends using scenarios when possible because they are cheap and effective in many situations, and because their story-like quality makes them approachable by both architecture experts and non-experts.

## 8.2 Invariants (constraints)

*Invariants*, also known as *constraints*, restrict the system, specifying either how it must be, or must not be. A defining characteristic of an architectural style is the constraints it places on the elements in a system. A pipe-and-filter style, for example, constrains the ordering of items in pipes and constrains the topology of how pipes and filters may be connected.

Developers impose guide rails (as constraints) on their designs so that they can understand them better. An unconstrained system can do anything, and therefore it is impossible to reason about what it may or may not do. Seemingly simple constraints like "Clients must not connect directly to the database, and must

| | | |
|---|---|---|
| **Name** | The scenario name can be anything that is helpfully descriptive. | |
| **Target model** | The model that the scenario applies to. Functionality scenarios are often applied to component assemblies and port type models, but could apply to any model with elements that can change over time. | |
| **Initial state** | The initial state describes the state of the model before the scenario starts. For example, the initial state could describe the contents of the library and the existing loans. | |
| **Actors** | The list of actors who are involved in and initiate the scenario steps. | |
| **Steps** | Steps consist of the following: | |
| | **Actor** | Each step has an actor who initiates the step. Scheduled or timed events can be modeled as a timer actor. |
| | **Action** | Each step represents an invocation of an action that is defined on the target model. For example, Step 1 in the library scenario corresponds to a ListBooksAbout(topic) action. |
| | **Referenced model elements** | Each step may refer to model elements. For example, Step 5 in the library scenario refers to “Copy of Book” and “due date”, which must be defined in the model. |
| | **Return value** | Each step has an optional return value, or response, which is described after a slash, as in Step 1 in the library scenario. |

Figure 6: The parts of a functionality scenario. A functionality scenario refers to a target model, and consists of a scenario name, an initial state, a list of actors, and steps. Steps are broken down into the actor that initiates it, the action performed, references to model elements, and an optional return value.

instead connect only to the business tier” enable developers to better reason about caching and performance. In short, no constraints = no analysis.

Invariants on class diagrams are written in UML notes, and can be written in Object Constraint Language (OCL) by putting the OCL expression inside curly braces. Architectural constraints are more often written down separately from diagrams as text. *Static invariants* deal with structure and *dynamic invariants* deal with behavior.

**Static invariants.** A static invariant is a restriction on the arrangement or quantity of instances (e.g., objects, component instances, connector instances) that can be created. An example of static invariants is that every truck must have an even number of wheels. In this case, you would have types that represent trucks and wheels, and the invariant restricts the how the instances of trucks and wheels are arranged. Another example static invariant is that every piece of data collected from a user must exist on at least two hard drives in separate server rooms. Static invariants can appear in many models and in different forms. In UML class diagrams, cardinalities on associations are static invariants, as are the ordering constraints like {sorted}.

**Dynamic invariants.** A dynamic invariant is a restriction on the behavior of instances. Examples of

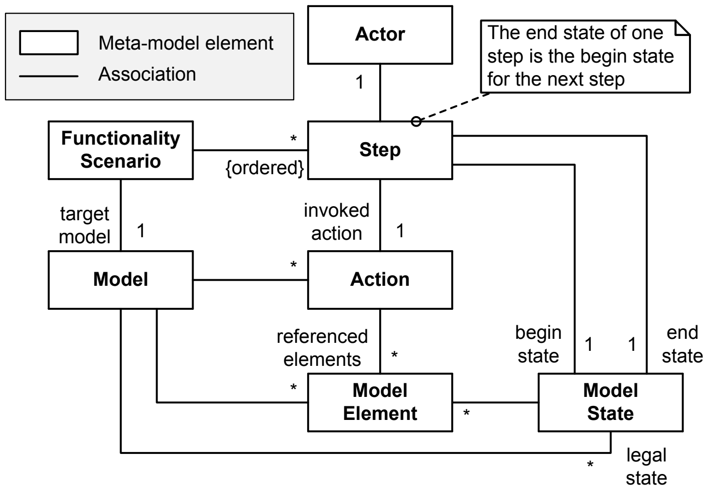

Figure 7: The meta-model for a functionality scenario. A functionality scenario is a behavior trace consisting of a sequence of steps. Each step is an occurrence of an action, initiated by an actor, and transitions the model from one legal state to another.

dynamic invariants include: only the print driver may send commands to the printer, every opening of a drawer is afterwards paired with a single closing of that drawer, or every ticket submitted by a user results in a response email being sent. In practice, you see many more static invariants documented because it is rather hard for humans to reason precisely about behavior.

## 8.3 Improving view quality

At this point you may be a bit worried about your ability to handle many views of your architecture. Perhaps you are even wondering if the divide and conquer strategy was such a good idea. Fortunately, there are techniques that help you manage those views, detect inconsistencies, and steer them to become consistent. Three techniques are discussed here: writing functionality scenarios, animating scenarios, and writing action specifications. The techniques on functionality scenarios augment the advice given in section §8.1.

## 8.4 Functionality scenarios stitch together views

What views show in isolation, functionality scenarios reassemble into a whole, like a thread that connects separate pieces of cloth into a quilt. This insight is critical to Philippe Kruchten's 4+1 views of architecture [6], where the +1 view is scenarios that connect the other four views. A single scenario can refer to elements that appear in different views, and even different viewtypes, so scenarios help the reader to relate the pieces and understand the whole of the model.

In its most common use, a functionality scenario applies to a single model in a single view, such as applying to a domain model, a port, an allocation model, or a component assembly. However, it is easy to write a scenario that applies across models and even viewtypes. For example, a scenario for packaging and deploying source code would describe how it is compiled (module viewtype), tested (runtime viewtype), and distributed onto servers (allocation viewtype). Strictly speaking, a scenario applies to just one model, so when they are used to stitch together several views, those views must be of the same model, perhaps the master model.

Regardless of how casually you write them, functionality scenarios always have the advantage of reading like a story. However, if you take care in how you structure them, they are effective at tying together views.

Name: End-to-end copy of Moby-Dick
Initial state: Larry is a Librarian; Bart is a Borrower
Actors: Larry, Bart
Steps:

| | |
|---|---|
| 1. Larry lists all Books about "fishing" / No matching books are found. | Library Presentation (LP) extracts and sanitizes input from user form<br>LP queries books about "fishing" from Library Core (LC)<br>LC queries Inventory Database (ID) for all entries in Book table whose subject contains the substring "fishing". LC returns list of book objects to LP<br>LP renders list of book objects as a result screen |
| 2. Larry adds a Copy of the Book "Moby-Dick" by Herman Melville to the Library. A record of the Book is also added. | LP extracts and sanitizes input from user form<br>LP adds book "Moby-Dick" to LC<br>LC queries for existence of book "Moby-Dick" in ID. ID says no.<br>LC inserts new book "Moby-Dick" into ID<br>LC inserts new copy of book "Moby-Dick" into ID<br>LP renders success screen |
| ... | |

Figure 8: A fragment of a two-level functionality scenario, which elaborates the first two steps of the scenario from figure 5. The right column references subcomponents of the Library System: The Library Presentation (LP), Library Core (LC), Inventory Database (ID).

This section describes that structure and the rigor you should apply. As you grow to appreciate why the structure exists, it will feel less like a burden and more like an opportunity.

**Informal dialogue.** Here is an example of how you can keep your scenario tightly connected to your model. If you were working on the scenario for the library system (from Chapter 12, Design Model Elements), your inner dialogue, or perhaps even spoken dialogue if you are collaborating with someone else, might sound like this:

> OK, this step deals with how the borrower returns the copy. Is there an operation defined on the port for that? Yes, it's called Return(). So the borrower, Bart, will "return the copy of Moby-Dick" via the pDesk port. Actually that would be Bart and Larry, since only Larry the librarian can use the pDesk port. Then the system will need to match that up with the loan. Of course Bart doesn't know the loan ID, so the system will need to look the loan up and then change the loan's state from ... hmm, I haven't defined loan states yet.

This is just a snippet of dialogue but should give you an idea of how the writing of a scenario immerses you in the details of the target model. It might even feel a bit like writing code.

**Checklist.** As a reference, a checklist can be quite helpful in helping you learn to write a good scenario. Below is a checklist that can be used either while writing a scenario or when checking one after-the-fact.

- **Actor**. The actor that initiates each step should be clear, as should the recipient. To ensure this, always use the present tense, which will avoid linguistic constructions that hide the initiator, as in, "The copy is checked in." While you are thinking about the initiating actor, ensure it already knows about any data it must pass as a parameter. Also consider if the actor is allowed to initiate this action. The actor must also have a path of communication to the recipient, so a connector, dependency, or a communication channel may need to exist.

- **Action**. Each step should refer clearly to a single action defined on the target model. A good scenario has a single level of abstraction across steps. For example, if one step is, "Larry adds a copy of Moby-Dick to the library," another step should be at about that same level of detail, and not "Larry enters his username and password." Action names in the scenario should be as close as possible to the action name in the target model, but you can allow minor differences, especially if they improve readability.

- **References**. Scenario steps refer to model elements, such as parameters passed in or return values. The scenario should have no dangling references, so all references must be defined in the target model, including associations, attributes, and states (or other details relevant to the model type). However, the scenario should avoid referencing "things inside of things," so a scenario at the boundary of your system should not refer to subcomponents inside the system.

- **Target Model**. Each step should transform the target model from one legal configuration, or state, to another. An example of an illegal state would be a stack containing -1 items, or a loan that is not associated with a borrower. You should insist that each step causes a visible change to the target model. If it does not, you may need a more detailed model or a less detailed scenario. An exception to this is query operations, which rarely change the model. No step should cause invariants or constraints on the model to be broken.

- **Overall**. Does the scenario make sense overall? Has it skipped over any steps or difficulties? Does any actor in the scenario "just know" where something is, or "just know" which other object or actor to talk to, one that it should have to look up? Does it omit any difficult start-up or tear-down steps? As you write a scenario step, think, "is this exactly the right word; does it match the rest of the model?" You can also start with an empty target model and use the scenario to begin populating it, adding items as you mention them in the scenarios.

Scenarios without this careful attention to references are useful for understanding and documenting the design, but carefully structured scenarios help you while you write them to catch errors or omissions, and help you to think about how the views are stitched together to reveal the whole of the design.

## 8.5 Animating functionality scenarios

You have just learned how to write structured functionality scenarios such that you can catch problems. Your models will change after you write your scenarios, so you would like to be able to go back and re-check the scenarios. This section describes how to *animate scenarios* to check for problems in your models.

Developers often mentally animate programs in their minds in order to debug them. They walk through the program line-by-line and think about what impact each line of code will have and consider possibilities for bugs. The result is higher quality code with fewer bugs.

Animating scenarios is analogous to animating a program and often feels the same. Animating a scenario means that developers walk through a scenario and mentally animate it step-by-step. With each step, they imagine the changes that are taking place to the model. Animation promotes a close mental connection with the model, a perspective that helps you to catch inconsistencies and errors of omission.

The simplest version of animating a scenario is simple syntax and reference checking, but you can do much more. To do so requires you to use the scenario to examine your understanding of the system. Recall that at the pinnacle of the pyramid of modeling competence (from figure 9) developers use models to amplify their reasoning. Each step in the scenario can be used as a context from which to examine the system and

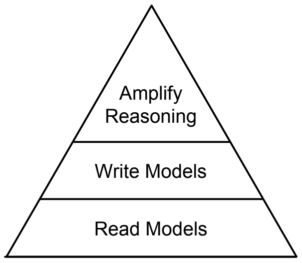

Figure 9: Everyone who works with models must be able to read them. Some people will be able to write models, but the goal of designers should be to use models to amplify their reasoning abilities.

see if it is reasonable and complete. The following are some questions that you can ask as you animate a scenario that will help you go beyond simple syntax checking.

- **Communication.** Does the actor have a choice about which port or connector to use? Should another port be added? Are the properties on the port or connector appropriate for the kind of message being sent (e.g., an insecure channel or a daily batch)? Does/should the actor know how to contact the recipient, or know how to choose the right recipient?
- **Before and after.** Should this action initiate any other messages? Is something returned or should it be? What should the state change of the model look like? Is this step dependent on something that must have happened before? Do the actors and the system have access to the data they are required to pass?
- **Beyond the scenario.** Is there a variant of this scenario step that would be more challenging to the system? Is there interesting behavior involved with startup, shutdown, empty collections, or deleting elements? How many scenarios are needed to give confidence in the system behavior? Is the behavior of each element reasonable given its allocated responsibilities?

These kinds of questions could be answered without a scenario, but the concrete context of the scenario can help uncover problems and can open up new avenues of thinking. These questions do not have right or wrong answers based on simple checks — do the connectors have *appropriate* properties? — so developers use scenarios to augment their analysis. When you animate scenarios across your model with the intention of detecting problems, you will find that reading through a single step is a rich mental activity that reinforces the interconnected nature of your models.

## 8.6 Writing action specifications

A third way to tie together your models is to use *action specifications*. Action specifications can tie together various views in much the same way that functionality scenarios can. Consider the action check_out_copy that describes how a borrower could check out a copy of a book from a library.

> void **check_out_copy** (Copy c, Borrower b)
>
> **pre-condition**: c is not removed, c has no current loans
>
> **post-condition**: new Loan l, linked to b and c, out = today, in = null, due = c.library.loanLength + today

Reading the action spec itself gives you some understanding of how the system must work: copies can be removed from the library, there are loans recorded, some loans are “current,” loans identify the book and the copy, and there is a standard loan length.

You can use the action specifications to limit the size of your model by only including details that are required in the action specifications. You might be tempted to include the age of the book copy in the model, but if it is not mentioned or needed in any action spec then you would omit it from the model.

Action specifications make specific demands on other models. This spec requires that the following terms be defined: Copy, Borrower, Loan, and Library. Those terms have additional attributes: current loans, out, in, due, and loan length. And some states are referenced: removed books and current loans. A complete model would describe all of the states and transitions, and how the actions drive the state transitions. It should also describe how attributes relate to states, for example that a Copy has an attribute called isRemoved that corresponds to its state. You would also expect to see this action appear as a step in at least one use case.

Despite their utility, action specifications are time consuming and therefore expensive, so this book refrains from advocating that you regularly include them in your models. The idea underlying them, however, is that all the views of your model are interrelated, so knowing how action specifications work will improve your modeling ability. Even when you do not write down the spec, you may be thinking, “Have I defined all the terms I would need to satisfy the pre-conditions and post-conditions?”

## References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[2] Desmond F. D’Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[3] Brian Foote and Joseph Yoder. *Pattern Languages of Program Design 4*, chapter 29, Big Ball of Mud. Addison-Wesley, 2000.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-WesleyProfessional Computing Series)*. Addison-Wesley Professional, 1995.

[5] Michael Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[6] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.