InferData Corporation, USA

High-quality modeling in UML

Richard Mitchell, Petter Graff, and Vladimir Bacvanski

www.inferdata.com

rmitchell | pgraff | vbacvanski@inferdata.com

Short overview

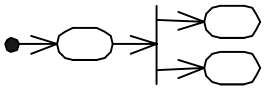
Modelers at InferData use an approach to creating and maintaining UML models that yields high-quality models. The high level of quality comes from:

- extensive cross-checking between different models (which requires us to pay attention to the possible meanings you can give to certain models)
- maintaining continuity of structure and behavior (from models of the problem domain, through black-box models of the system, to design models of software)
- careful testing of models (example-level models are used to test general-level models).

This paper focuses on modeling the problem domain. It is intended for developers and managers who use, or are considering using, the Unified Modeling Language in their development process. The paper assumes an elementary knowledge of object-oriented concepts and their use in OO modeling.

Overview

UML is a widely-accepted language for building object-oriented models. Development processes have been built around it, and tools have been built to support it.

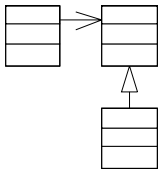


A development **PROCESS** is really two processes: a technical process and a management process

This tutorial presents key elements from a layer we think of as coming between the modeling language and your favorite development process. The layer clarifies and constrains the use of UML, with the highly-focused aim of enhancing the quality of models built and maintained during the life of a software product.

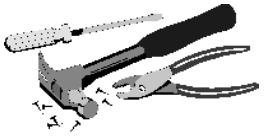
- maintain *continuity*
- *cross-check* models
- *test* models

Our **APPROACH** constrains the process and the modeling language to enable cross-checks and tests



The de facto modeling **LANGUAGE** is UML

We think of the layer as offering an approach to system development. The approach can be used with different processes and different tools (and even a different modeling language).



TOOLS support the use of the language within the process, and, ideally, help apply cross-checks and tests

The tutorial addresses:

Problem domain modeling. Modeling the business events a system must respond to, and the types of objects affected by these events.

Continuity. Carrying the structure and behavior of a problem domain into system and design models.

Cross-checking. Ensuring that every model can be cross-checked against other models.

Model testing. Using example-level models to test general-level models of structure and behavior.

Architecture. Making the architectural view of a system provide placeholders for different modeling viewpoints.

Process. Distinguishing between development activities and phases of development.

Vocabulary. Using models of object types to provide a vocabulary for behavioral models.

The approach presented here does not cover all aspects of system development. For example, we don't address support for project-management through time-boxing of development activities, or support for mathematical precision through the use of UML's Object Constraint Language, OCL. Instead, our particular selection focuses on high-quality modeling, presenting what we have found in practice adds considerable value to many kinds of projects with only a small investment in learning.

The approach owes much to pioneering work by the designers of Fusion (Coleman et al 1994) and Syntropy (Cook and Daniels 1994), and it has points in common with Catalysis (D'Souza and Wills 1999).

The InferData Approach

The InferData approach has these key characteristics:

- UML-based
- incremental and iterative
- architecture-supported
- behavior-driven
- two-level
- cross-checked
- continuity-focused
- lightweight
- sufficiently precise.

These characteristics are briefly outlined in the remainder of this section. Later sections explore the characteristics and their underlying principles more fully.

UML-Based

Visual modeling uses UML diagrams, such as use case diagrams, sequence diagrams, state models, and class diagrams (which are used to document types as well as classes). We also exploit UML's placeholders for text attached to diagram elements (such as the placeholders for preconditions and postconditions).

Incremental and Iterative

Nothing comes out right first time, so model-building has to be an iterative process. Also, a long modeling phase followed by a long implementation phase is generally disastrous, so models and code need to be developed incrementally.

Architecture-Supported

All modeling and implementation take place within the context of architectural models. These include models based on subject domains, and models based on technical domains.

Domains divide the overall problem into manageable parts, which can be worked on by separate teams. For example, a system concerned with financial instruments could have subject domains concerned with *instrument reference data* and *pricing*, where the pricing domain depends upon the reference data. And the system could have technical domains such as the core of the system versus its user interaction and persistence domains. One team could work on, for instance, the user interaction technical domain within the pricing subject domain.

Behavior-Driven

The state of a world or a system is modeled using objects, their attributes, and the links between them. Behavior is concerned with changes to this state.

Changes in a business domain are modeled as events. Changes in the externally-visible state of a system are modeled as use cases. Changes inside a system are the result of objects sending messages to other objects.

When modeling a system, use cases are described first as essential use cases, capturing the required functionality of the system. Later, concrete use cases capture the designer's choice of interaction sequences that present this functionality to the system's users.

Two-Level

Models are built at both the general level and the example level, supporting how we think (have you ever said "could you give me an example of that?"). Key general-level models are supported by examples. For instance, type models are supported by example snapshots, and behavior models are supported by example scenarios.

There are several advantages to working with examples. It can be easier to begin modeling with examples, and generalize later. A general model can be explained to others using a few examples. A general model can be tested by checking that it supports carefully-chosen examples.

Example-level models can use actual (or plausible) individual examples (e.g., Kim rents copy 12 of the video version of Titanic from LowPrice at 8.30 p.m. on June 4th, 2001, paying $4.50 in cash) or prototypical examples (e.g., member m borrows copy c of title t from video store s at timepoint p paying an amount a).

Cross-Checked

In principle, a project produces a single model. In practice, this single underlying model is never seen, because it is too big to present. Instead, modelers construct a number of individual models, each of which is a view, or projection, of the underlying single model. Even a modest-sized project produces many individual models: type models of different aspects of the system, supporting snapshots, specifications of essential use cases, state models of key types, and so on. It is important to ensure that all the different models are talking about the same underlying model. This can be achieved through an extensive set of cross checks. To make cross-checking workable, the meanings of UML diagrams must be constrained.

Cross-checking has several important benefits:

- during early model construction, inconsistencies revealed by cross-checks raise important analysis questions
- later, cross-checks that do not reveal inconsistencies show us that the models are maturing
- during model evaluation, models can be tested for consistency and completeness.

Continuity-Focused

The roots of object-orientation are in simulation. Whenever a modeler is faced with a question of the form "should we model it this way or that way?" the answer is sought in the problem domain, so that models exhibit continuity with the domain. Relevant properties of the domain are retained in system specifications and internal designs. As a result, changes that appear small in the business domain are more likely to result in small changes to the system.

Lightweight

Rather than present a heavyweight process with many ordered steps, each with milestones and deliverables, we explain the content of different models, and how they should be cross-checked.

Developers are encouraged to switch modeling activities frequently, because this aids thinking. For example, if you get stuck on one aspect of a type model, try building a snapshot, or exploring what events change the state of the types of objects in question. There is advice on what to address when, but there are no absolute constraints. For example, a high-risk area could be designed before other areas are analyzed.

Sufficiently Precise

Different projects, and different phases within one project, need different levels of precision in the various descriptions. Descriptions of behavior, for example, can be written:

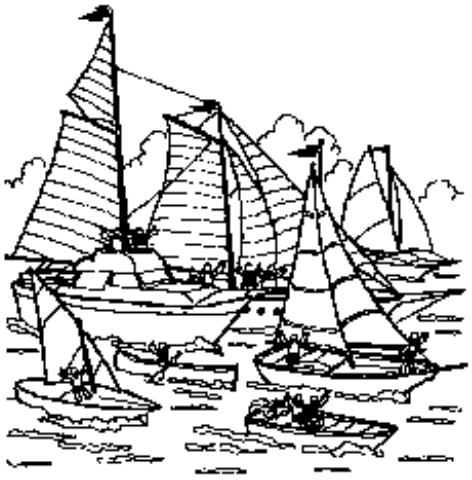
- informally
- semi-formally, using a vocabulary defined in a type model
- fully formally, using the UML's Object Constraint Language, and supported by a type model.

We strongly advocate reaching the semi-formal level, supporting all descriptions of behavior with a type model—we shall see examples later. Many projects do not need a fully formal approach.

Explaining the Approach

Here are the key aspects of the approach, in more detail.

Continuity



Some models lack continuity with the situation in the world being modeled. The model does not correspond well with the situation. This lack of continuity might not prevent us from building the system in the first place. The danger is that it will prevent us from adapting the system to new requirements. Here is an excellent example, retold with permission from (Jackson 1983).

Imagine we are software developers, and we have a client who runs a boating lake, where members of the public can rent a boat for a session. Sessions vary in length. We have been contracted to build a system that can report each day's average session length. Our analysis of the problem reveals that the key output is defined thus:

Analysis model:

$$\text{average session length} = \sum_i (\text{finishTime}_i - \text{startTime}_i) / n$$

The start and finish times of sessions are indexed by i, which can take values in the range 1 to n, where n is the number of sessions in the day. By subtracting the start time of the i-th session from its finish time, we obtain the length of the i-th session (we can choose a suitable way of recording start and finish times for a session so that subtraction yields the correct result). To calculate the

average session length, we sum the lengths of all the sessions, and divide by how many sessions there were.

Let's begin designing the internal workings of the system. Consider the following tabular representation of start and finish times, using simplified values for these times:

Session number	Start time	Finish time	Session length
1	10	30	20
2	20	50	30
3	30	50	20
COLUMN TOTALS	60	130	70

Examination of the table reveals that we can find out the sum of the lengths of all sessions in two different ways:

- either by subtracting each start time from its corresponding finish time, and adding up the resulting column of session lengths (as in the analysis model)
- or by keeping running totals of the start times and the finish times, and doing a single subtraction of these two running totals.

Therefore, we can base our design on the following revised model of the average length of a session.

Design model:

$$\text{average session length} = (\text{sum}(\text{finishTime}_i) - \text{sum}(\text{startTime}_i)) / n$$

Implementing the earlier analysis model directly would entail pairing up the finish time of every session with its start time, without knowing in advance how many unfinished sessions there would be at any one time. The design model is considerably simpler to implement: just keep two running totals and a counter, and perform the calculation of the average at the end of the business day. We implement this design.

Our client is delighted with the system, and returns after a week asking for a small change. As well as providing the average session length, the system should output the length of the longest session. We quickly notice that this would necessitate a complete re-design, and we have to persuade the client that this will be an expensive modification. The client is not happy, but, optimistically, returns to ask if we can do something simpler: make the system provide the shortest session instead. This, too, would necessitate a complete redesign, and, by now, we've probably lost a client.

What went wrong? The key change we made when moving from the analysis model to the design model was to remove an explicit representation of a session. Yet sessions are what the client's world is about. It is not surprising, then, that a change that seems small to the client, in her world, might prove impossible to make in our model of her world, since we have not modeled her world adequately.

So, we shall be careful to understand the world into which we are putting software. Then we shall strive to retain this understanding in all our models, right down into the internal design of the system. If our client's world is about sessions, our running system will certainly contain objects of type session.

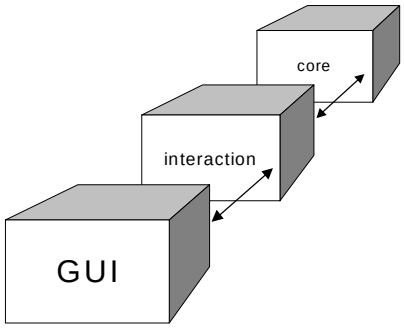
This example has focused on a structural concept—the existence of individual sessions. The same principle applies to behavioral concepts. Behavior is all about how objects change over time. If certain changes can happen to objects in a model of a problem domain, these changes must be supported in software.

Continuity

- Discover the key problem domain concepts
  - Look for structural concepts *and* behavioral concepts
- Make sure these concepts are modeled in software, from systems analysis models, through design, and into code

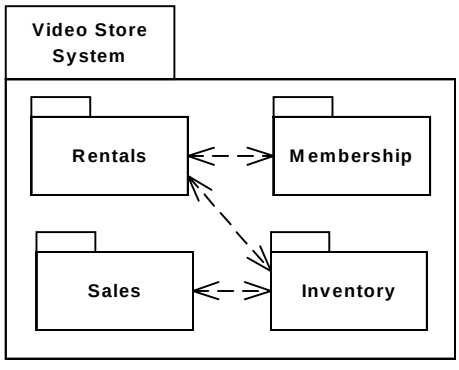
Architectures

We can organize the descriptions of a system into what we shall call technical domains. This picture shows three such domains, and it provides a logical architecture that is suitable for small programs that do not have a persistency requirement.



The user interacts with the program through controls or widgets in a graphical user interface (GUI). The essential functionality of the program is captured in its core. The interaction domain's responsibility is to manage the interaction between the GUI and the core, translating the user's manipulations of GUI elements into requests for services from the core, and continually updating what the user can see on the screen in response to changes in the core.

In practice, many more technical domains are needed, to allow for a core system that can also have persistent objects in a database, be connected to physical devices, communicate with other systems, and so on.



Development can also be about different subject areas, or subject domains, within an overall system. For example, if we were modeling a video store system, we might consider four subject areas, shown here using UML's package notation.

The rentals side of the business can be thought about largely separately from the problems of handling the membership of the store, and so rentals and membership can be two separate subject areas, or subject domains. Similarly, we can work on the sales and inventory management largely as separate problems.

The four subject areas are not entirely independent, however. The diagram shows the key dependencies as dashed arrows. For instance, the rentals subject area depends upon membership because, for example, you cannot rent videos to non-members. Membership depends upon rentals

because, for example, a member can be in poor standing if he or she has too many overdue rentals or outstanding fines.

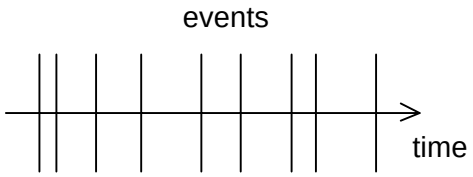
Subject domains and technical domains are different. One particular developer, at one moment in time, might be working on the GUI aspects of the membership domain, or working on the core of the rentals domain, and so on.

Architecture

- You'll need several organizing principles for architectural models
- Two examples: divide a system into technical domains and into subject domains.

Model of Time

An approach to building object-oriented models needs an underlying model of time. This one is described in (Jackson 1995). There is a timeline, with time increasing from left to right. Each vertical line represents a moment in time at which the state of the world changes.



In between the vertical lines, the world remains in some state for an interval of time. We model the state of the world using objects, and the changes of state using events. (Of course, this model of time, and hence the use of events for modeling, are not appropriate for all aspects of all systems, such as aspects of process control systems.)

Events are atomic. Changes to the state of the world can occur only in events. No changes occur in the intervals of time between the events.

When we are modeling the core of a system, the events that interest us are those events happening in the world that are to affect the system. For example, suppose that we are building a system for a video store called LowPrice. Then we must model the core so that the event "Kim rents copy 12 of the video version of Titanic from LowPrice at 8.30 p.m. on June 4th, 2001, paying $4.50 in cash" is an event that changes the state of the system. But the event "Kim borrows $5 from Heidi in order to go to LowPrice to rent Titanic" is one we might choose to ignore.

We shall, of course, classify the objects and events into types. So we shall model with object types such as Member and Copy, and event types such as *rent* and *return*.

Here are two quotes suggesting that those who do not model events directly still recognize their importance.

"A good source for identifying use cases is external events. Think about all the events from the outside world to which you want to react. A given event may cause a system reaction that does not involve users, or it may cause a reaction primarily from the users. Identifying the events that you need to react to will help you identifying the use cases." (Fowler 1997, page 48, paragraph 4).

"The actor may also inform the system about certain external events or the other way round…" (Jacobson, Booch and Rumbaugh 1999, page 148).

In the InferData approach, we do more than identify the external events, we capture the changes in the world that occur in these events, and specify them carefully. Then we build models of systems so that changes inside systems show continuity with allowed changes in the world.

A changing world

- As the world goes about its business, it changes
- We model these changes explicitly
  - Each discrete change is called an event
- We have a model of time to support the concept of discrete changes.

Development Activities

A developer using UML might, at some moment, be constructing a class diagram.

What might the class diagram describe? It might describe the different types of object in the world surrounding the software. Or it might describe the types of object needed in a definition of the required functionality of a system. Or it might describe a design of how to deliver some of that functionality.

When might the diagram be produced? It might be produced early in a project, when most attention is focused on understanding the problem domain, and the needs of users, even though some attention might be focused on high-risk aspects of a proposed design. Or it might be produced later in the project, when most attention is focused on the internal workings of the system, even though some questions concerning the problem domain are still surfacing.

Some methodologists name their models by when they are produced. We prefer to distinguish models by what they describe, rather than by when they are built. Then development activities can be distinguished by what kinds of models they lead to. We can distinguish models of the world, models of the required properties of a system, and models of the internal workings of a system. We call these domain models, system models, and design models. They are the result of activities called domain analysis, systems analysis, and design. It's perfectly okay to do some design in the early stages of a project. And you'll still be doing (a little) domain analysis when most of the effort is on coding. If you uncover a domain fact while coding, don't assume it's a design issue. Add it to the domain model.

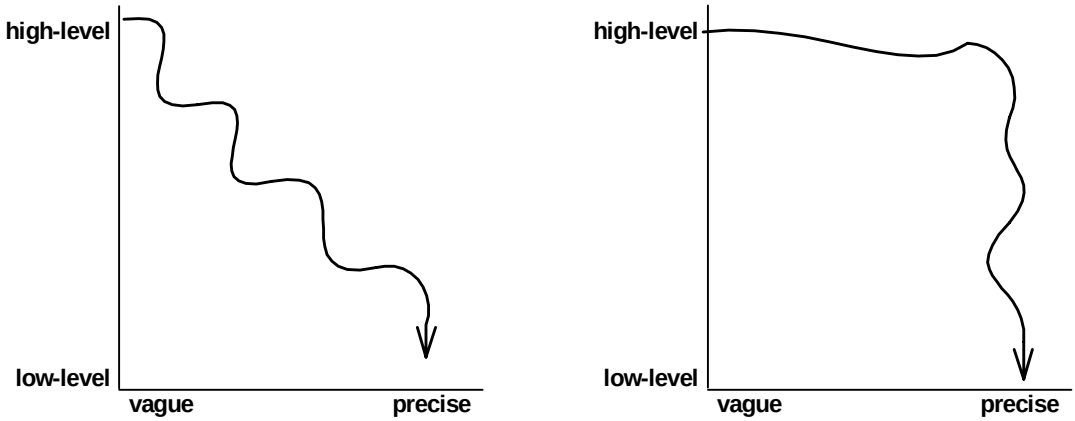
Concern for the internal workings of software pervades most activities in the Rational Unified Process, whose authors, Jacobson, Booch and Rumbaugh (1999, page 175) have this to say about the analysis model in RUP (the phrase "how to realize" is significant):

"Outlines how to realize the functionality within the system, including architecturally significant functionality; works as a first cut at design."

A popular view of development is represented by the line in the left-hand diagram below. Development begins with vague, high-level descriptions of aspects of a system. Development proceeds in a more-or-less straight line by adding detail at progressively lower-levels.

This view of development became known as top-down development, and is epitomized by approaches based on data-flow diagrams. In fact, the original proponents of top-down develop-

ment never suggested that a description at one level of abstraction should have its terms defined in the next level down.



Rather, they argued that the development space is at least two-dimensional. Level of precision and level of abstraction are not on the same axis. It is possible to add precision without moving to a lower-level of abstraction.

The line in the right-hand diagram becomes a feasible path. Start with models based on a vague but high-level view of the world and the required software. Work on these models, making them more precise whilst remaining at a high level of abstraction. Then start to become more concrete by designing the internal workings of the required system.

Many UML-based methods retain the view that we progressively add detail to our descriptions, and that, to do so, we become more concrete. There is a danger associated with such top-down approaches. Precision is added by adding detail at lower levels. So some analysis-level decisions are not made until we are working at a lower level. Many developers will recognize the problem—they are left to finish the analysis while writing the code.

In our approach, we distinguish between domain analysis, systems analysis (which is largely black-box), and internal design (including architecture). In each activity, we can build models at whatever level of precision is appropriate (we'll see an example soon). And, most importantly, we do not prescribe the order in which you should undertake the activities. On a given project, you might, for example, do some design work to see if you can connect two technologies. Knowing that you can, you develop a specification for a system. When the system is in place, you explore the domain, looking for opportunities to re-engineer the business around new, related systems. So, you carried out design, then systems analysis, then more design, and then domain analysis.

Of course, you cannot build software that exhibits continuity with the world unless you are prepared to model that world. So, even if you do not deliberately explore the domain looking for new opportunities to introduce systems, you must bring world knowledge into your systems analysis models and your design models.

Distinguish phases and activities

- Development activities include domain modeling, system modeling, and design
- Don't assume that all domain modeling must happen in the first phase, and all design in later phases

Detail is not necessarily implementation

Here is another way to view the topic of the previous section, which argued that you can add precision to a model *without* going to a lower-level. What sometimes confuses modelers, and business people, is that adding detail in order to be precise can seem like adding the kind of detail that should appear in a program. Yes, sometimes it is the same kind of detail. But that's because programs implement the rules of the business. Here's an example, taken from a problem domain concerned with bank accounts and cash withdrawals. Consider this function, written in Java:

```
/**
 * Can amount 'm' be withdrawn from account 'a'?
 */
Boolean cashWithdrawalAuthorized( Account a, Money m ) {
      return
            (     m <= a.balance
              &   m <= a.amountRemainingFromDailyCashLimit
              &   m <= a.amountRemainingFromMonthlyCashLimit )
}
```

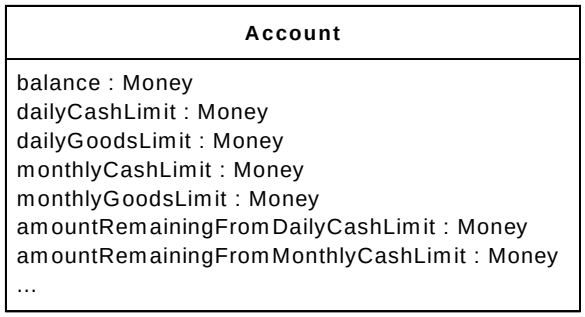
Which details of this function do not belong in a business-oriented domain model? Here are some:

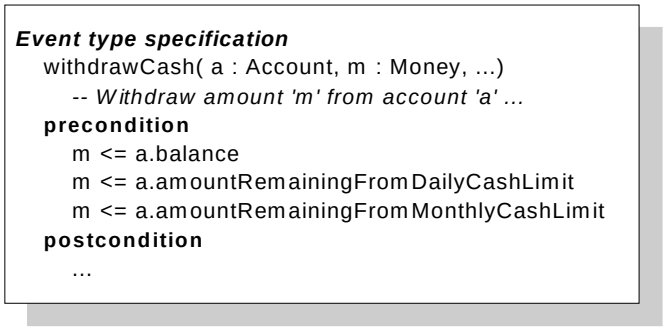
- the detail that it is written in Java, and hence all the details of the syntax (although the choice of programming language could be a non-functional requirement on a system).

Here are some other details that certainly *do* belong in a domain model:

- there are 3 separate checks on cash withdrawals
- more precisely: the amount to be withdrawn must be less than or equal to
  - the account balance
  - the amount remaining from a daily cash withdrawal limit
  - the amount remaining from a monthly cash withdrawal limit.

Here's how these details might show up in a domain model. We might have a type model showing the type Account and its properties (only some are shown in the diagram). The type model supports the specification of the event of withdrawing cash (again, only some details of this event specification are shown).



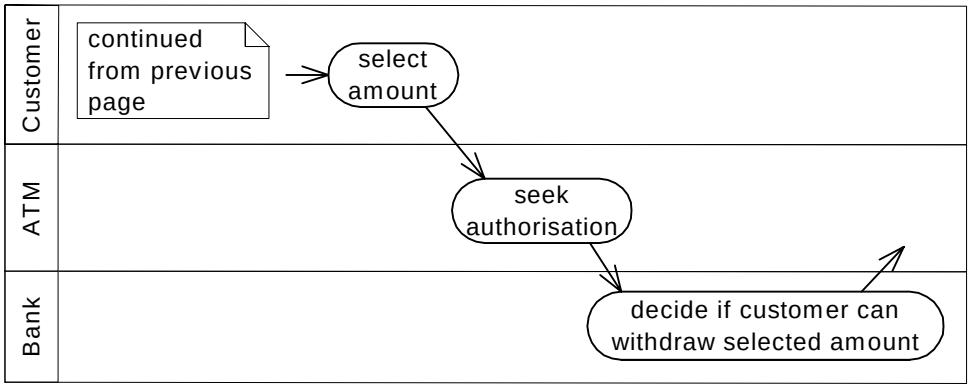


As you would expect, the rules the business imposes on cash withdrawals show up in domain models *and* in program code. The rules are the same in both worlds.

Of course, there are differences. The syntax is different. And where the rules show up is different. For instance, in our example, the rules show up as a precondition on an

event in a domain model, and as a checking function in a Java program. The domain model and the program could both be expressed in other languages, and organized in other ways, but the rules would still show up somewhere, and somehow, in both the domain model and the code.

One final point: naturally, you will want to build domain models that do not show all this detail. You'll want to build them in the early days of modeling, when you would otherwise sink under the weight of too much detail. And you'll want to build them at various stages in the life of the domain models, to provide summary models for those who want an overview. Below is a fragment of an activity diagram that discusses cash withdrawal, without any details of what makes a withdrawal acceptable. The model is still high-level, but it is imprecise.



But when you expand the step in the activity diagram that says

```
decide if customer can withdraw selected amount
```

into a precondition on an event specification, saying

```
the amount to be withdrawn must be less than or equal to
      the account balance, and
      the amount remaining from the daily cash limit, and
      the amount remaining from the monthly cash limit
```

you are moving towards a more precise domain model. You are not adding implementation detail.

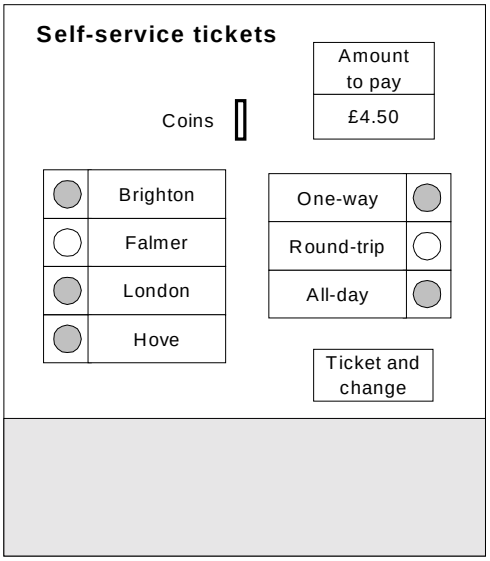
Level of precision v. level of abstraction

- When you add detail to a model, it can be detail that adds precision to an imprecise model
- Adding detail doesn't necessarily mean you've moved to a lower level of abstraction.

Use Case Descriptions

The UML User Guide (Booch et al 1999, p220) says that a use case *"is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor."*

In this section, we shall begin to write a use case description that abstracts away from *"sequences of actions."* In a later section we shall explore how to connect descriptions of behavior to descriptions of structure, and we shall be able to enrich the example begun here. The goal is to avoid making design decisions when building models of the external view of a system.



When you travel on the railway network in Britain, you can buy tickets from vending machines. Here is a simplified, schematic diagram of such a machine.

To buy a ticket, you select a destination, such as Brighton or London, and you select which type of ticket you wish to buy, such as a one-way or a round-trip ticket. If you are going to London, you can buy an all-day ticket, which lets you travel on the London Underground as well.

The machine does not let you select a type of ticket until you have selected a destination. When you have selected a destination, it illuminates the buttons for just the valid ticket types, and waits for you to press one of them. When you have selected your destination and a valid ticket type, the machine displays the cost of the ticket. When you have inserted enough money to pay for the ticket, the machine prints the ticket, dispenses the ticket, and dispenses any change you are due.

Customer selects destination;
Machine illuminates valid ticket-type buttons;
Customer selects valid ticket type;
Machine displays cost of ticket;
Customer inserts coins to at least the cost;
Machine prints ticket;
Machine dispenses ticket;
Machine dispenses any necessary change.

There is more to buying a ticket from the real machine than this. You can pay using banknotes, you can cancel when you are part-way through inserting money, and so on. But these additional details are not essential to our discussion. The box alongside shows the main sequence of actions in the "buy ticket" use case.

A developer following a traditional, use-case-based method would now explore variants to this flow, to cover cases such as the user canceling. We shall take a different route. We shall begin to rewrite the description of the use case, abstracting away from a *sequence of actions*. To abstract away from a sequence of actions, we use the precondition and postcondition clauses in a UML use case description.

To get started, we need to name the use case, and give it some parameters, thus:

buyTicket(tt : TicketType, here : Station, destination : Station, tendered : Money)
-- A customer buys a ticket of type 'tt' from 'here' to some
-- 'destination' station, supplying a 'tendered' amount of money.

There are constraints on the relationships amongst the ticket type, the two stations, and the amount of money tendered. These can be expressed in a precondition on the use case, which we add to the signature given earlier:

buyTicket(tt : TicketType, here : Station, destination : Station, tendered : Money)
-- A customer buys a ticket of type 'tt' from 'here' to some
-- 'destination' station, supplying a 'tendered' amount of money.
precondition
-- The ticket type 'tt' must be valid for a journey from 'here' to
-- the desired 'destination'.

-- The amount 'tendered' must be at least the cost of a ticket of
-- type 'tt' for a journey from 'here' to 'destination'.

We shall write a postcondition in a later section. For now, let us begin to think about designing the machine that embodies this precondition. We'll concentrate on the first part of the precondition, that the ticket type must be valid for the destination. How might you design a system that presents this precondition to railway passengers in a user-friendly way? For example, how could you show that, if you start in Falmer and go to Brighton, you cannot buy an all day ticket (which is intended for use only in London)? How could you avoid frustrating passengers by giving them error messages saying that the combination of destination and ticket type they have chosen is invalid? You could, of course, build the interface to the machine so that passengers must *first* select a destination, and *then* the machine offers only the valid ticket types for that selected destination (which is exactly how the machine is currently designed).

In other words, the precondition

-- The ticket type 'tt' must be valid for a journey from 'here' to
-- the desired 'destination'.

is a requirement, and the description of the use case as a sequence of actions

Customer selects destination;
Machine illuminates valid ticket-type buttons;
Customer selects valid ticket type;
…

is one possible design for the user interaction domain of a machine that fulfills this requirement (and a very good design, too).

To demonstrate that it really is a design, we need only devise some alternative design that fulfills the same requirement. How about this one: a passenger either selects a destination, in which case only certain ticket-type buttons are illuminated and operational, or a passenger selects a ticket type, in which case only certain destination buttons are illuminated and operational. It might not be such a good design (because it might be more confusing for occasional rail travellers, such as tourists, without being very much better for frequent travelers). But it is an alternative design, enabling us to see more clearly that a *"sequence of actions"* embodies design decisions. It is easier to keep such decisions out of assertions, such as the precondition above, than out of sequences of actions.

Jacobson, Booch and Rumbaugh (1999, page 164) acknowledge that use case descriptions *"often contain implicit decisions about user interfaces,"* and refer to (Constantine and Lockwood 1999) where there is advice on how to avoid this problem. The advice is consistent with our approach, though we are more thorough about defining the terms in our models.

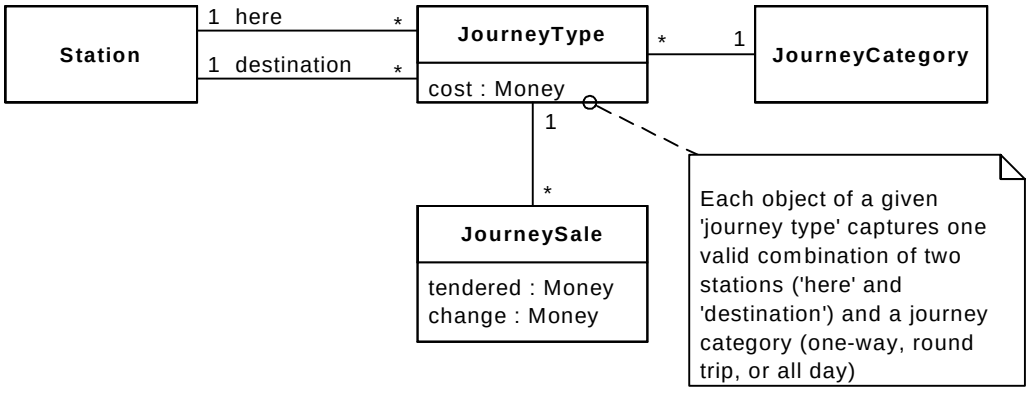
We do sometimes write use cases as sequences of interactions in early modeling. We do it when we are capturing the design of an existing system we want to understand. Having captured the current design in terms of sequences of user interactions, we abstract to higher-level descriptions based on preconditions and postconditions (and, as we shall see shortly, these will be supported by a type model). At this higher level, we can see more clearly what to re-engineer.

Use case descriptions

- A use case described as a sequence of interactions necessarily includes some design decisions
- The same use case described using a precondition and a postcondition can be free of such design decisions.

Connecting Models to Architecture

In this section, we continue the example of buying tickets from a vending machine, begun in the section entitled Use Case Descriptions. After some false starts, and after growing our understanding of the requirements for the vending machine, we produce the following model of the types of objects we are interested in (we'll show you how we produced it in the section entitled Introducing Cross-Checking).



Each time a passenger buys a ticket, we have sold a journey (more precisely, we have sold the right to make a journey). Each sale of a journey is for one possible type of journey. Different types of journey are distinguished by where they are from, their destination, and their category (one-way, round-trip, or all-day). For instance, a one-way from Hove to Brighton is one type of journey, but a round-trip from Hove to Brighton is another type. Each type of journey has an associated cost. Each journey sale has an amount tendered and an amount of change.

Using this model of the structure of our world, we can write an essential use case description, using a precondition and a postcondition. We begin by renaming the use case as "buy journey," since a ticket is just a token demonstrating an entitlement to a journey, and by renaming "ticket type" as "journey category."

Here is the use case description, consisting of a signature, a precondition, and a postcondition.

buyJourney(category : JourneyCateory, here : Station, destination : Station, tendered : Money)
-- A customer buys a journey of a certain 'category' from 'here' to some
-- 'destination' station, supplying a 'tendered' amount of money.

precondition
-- The selection must be valid: there must be a journey type 'jt'
-- from 'here' to the desired 'destination' of the given 'category'

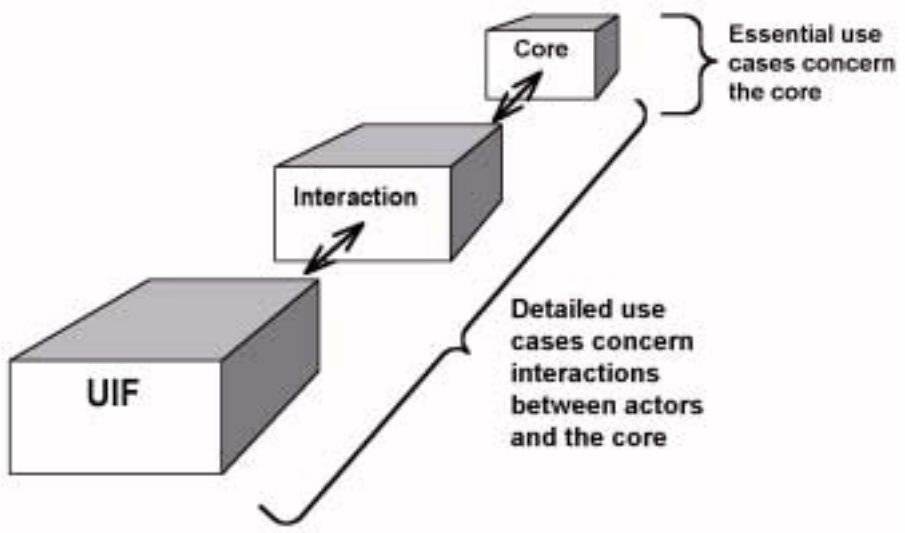
-- The amount 'tendered' must be at least the 'cost' associated with journey type 'jt'.

postcondition
-- There is a new journey sale 's', for which
-- 's.journeyType' is that 'jt' for which
-- 'jt.here' is the given 'here'
-- 'jt.destination' is the given 'destination'
-- 'jt.journeyCategory' is the given 'category'
-- 's.tendered' equals 'tendered'
-- 's.change' equals 'tendered' minus the 'cost' of a journey of type 'jt'.

As we showed in the section entitled Use Case Descriptions, the precondition expresses requirements, and the machine, in its current design, checks these preconditions partly by sequencing the passenger's actions.

The postcondition is similarly abstract (and so deserves more supporting commentary than we have given it here). It defines the required changes in the state within the machine, but only in terms of core concepts. For example, a printed ticket is a report that acts as proof of a completed sale. We could design other ways to allow passengers who buy a journey to board trains. So tickets do not belong in the core domain.

Of course, what gets printed on a ticket is important. We have modeled the information content of a ticket here in the core, but we can explore the details of the layout of a ticket when we address the user interaction domain. This sits between the user interface domain (not a wholly graphical interface, for this machine) and the core. It connects actions at the user interface (such as pressing a button) to use cases in the core, and connects objects in the core (such as a journey type object) to observable phenomena at the user interface (such as displaying the cost).



Having a model of the different domains to be addressed makes it easier to write descriptions that focus on one issue at a time (such as core functionality versus sequencing user interaction).

More on architecture

- Use architectural models that support the need to model different aspects of a system
- Examples of different aspects include the core of a system versus how users interact with the core functionality.

Connecting business and IT

This short section tells the story of the previous section in another way.

IT supports business. You can think of the relationship between business and IT using three layers (see the diagram on the next page). There is a business layer, which is concerned with business processes and business objects flowing through the various functional units of the business. There is a layer containing application logic and data—software that embodies the business processes and business objects. The two layers meet in a presentation layer.

Many UML-based development methods are use-case driven. The requirements of the application layer are expressed in terms of how users interact with the system. In other words, the presentation layer is used for specifying what goes on in the application and data layers.

In the InferData approach, the business layer is modeled explicitly, in a domain model. By the principle of continuity, the application and data layers (which we call the *core* of the system) are modeled so that they respect the structure and behavior of the business layer. The presentation layer is where interaction designers choose how actors will interact with the core of the system. The presentation layer contains requirements on presentation, but not requirements on core functionality.

More on use cases

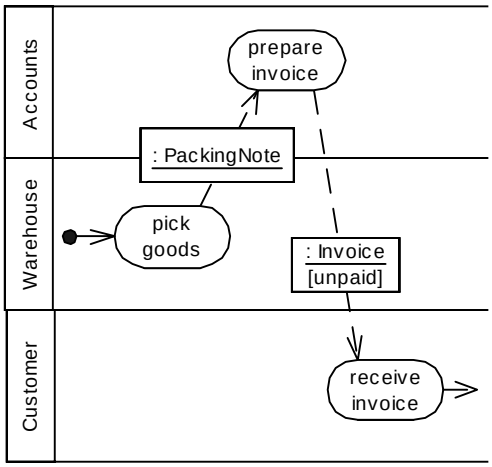
- The presentation layer is important—getting it right makes the system easy to use.
- A system's functionality is accessed via its presentation layer.
- But this functionality need not be described using the presentation-layer concept of interaction between a user and a system.
- Instead, it can be described precisely using a precondition and postcondition style, supported by a type model.

BUSINESS LAYER

Businesses are commonly organized into functional units (often called departments).

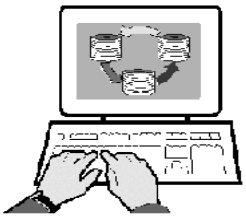
Business processes flow through these units, creating, modifying, transporting, and destroying business objects.

Some steps in some of the business processes are supported by computer systems.



PRESENTATION LAYER

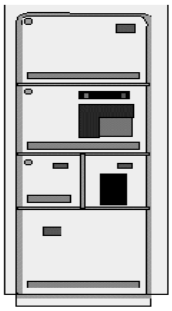
People interact with an IT system through a presentation layer.



APPLICATION AND DATA LAYERS

Code that embodies business logic manipulates software objects corresponding to business objects.

The software objects persist in some kind of storage (such as a relational database).



Introducing Cross-Checking

In an earlier section, entitled Connecting Behavior to Structure, we implicitly cross-checked two models. In this section, we make that cross-checking explicit. In a later section, we list some other valuable cross-checks.

Let's go through the specification of the "buy journey" use case, a step at a time, discussing what implications it has for a model of object types. The discussion gets quite detailed, but you can safely ignore the details and concentrate on the point of what we are doing—we are building a model of structure (a type model) so that it clearly expresses exactly the concepts needed to specify a piece of behavior (an essential use case). Through continual cross-checking, the model of structure and the model of behavior are kept in step, and end up fully consistent.

The specification of the "buy journey" use case begins with a signature:

buyJourney(category : JourneyCateory, here : Station, destination : Station, tendered : Money)
-- A customer buys a journey of a certain 'category' from 'here' to some
-- 'destination' station, supplying a 'tendered' amount of money.

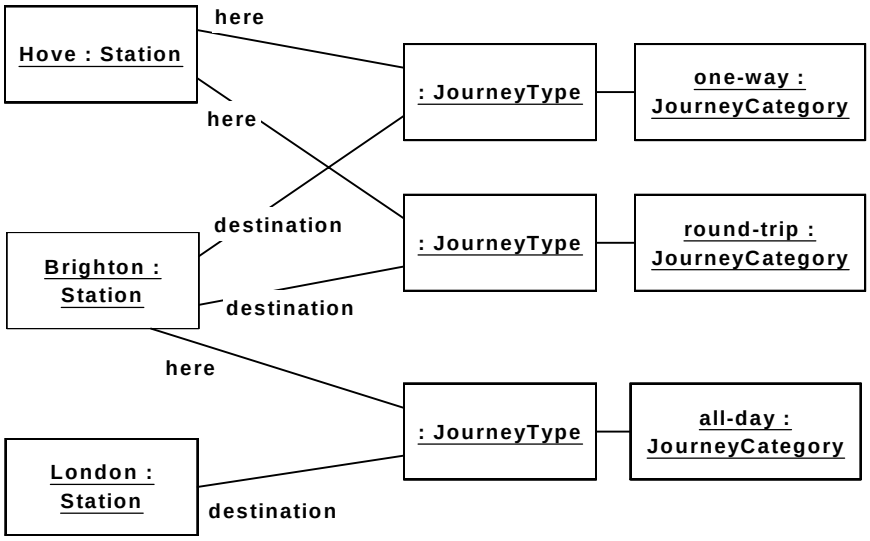
To support this, we need a model that declares three object types: JourneyCategory, Station, and Money (this last type might be considered a utility type to be declared on a separate diagram, but it must be declared somewhere). At this stage, there is no need for any relationships amongst the types.

The precondition begins thus:

precondition
-- The selection must be valid: there must be a journey type 'jt'
-- from 'here' to the desired 'destination' of the given 'category'

Now we need a type JourneyType, and it must have a JourneyCategory property, and two Station properties, with roles 'here' and 'destination.' This object type model does the job:

The project dictionary will define that the existence of a JourneyType object indicates a valid combination of 'here' and 'destination' stations and a journey category. The following snapshot is one of several constructed with the help of domain experts to help test the multiplicity constraints. The snapshot confirms that there can be many types of journey that begin and end at any given station.



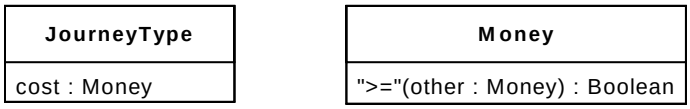
The snapshot shows that you can buy journeys of these types:

- one-way journeys from Hove to Brighton
- round-trip journeys from Hove to Brighton
- all-day journeys from Brighton to London.

The precondition continues thus:

-- The amount 'tendered' must be at least the 'cost' associated with journey type 'jt'

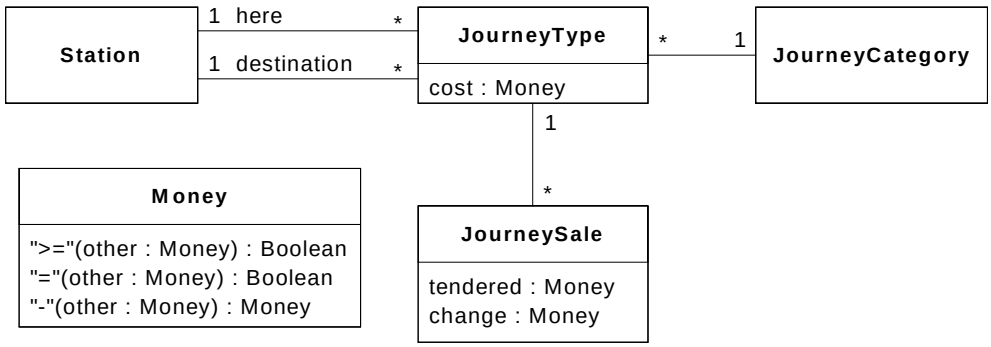
The amount tendered is already modeled, as an input to the event, but we must enrich JourneyType with a cost property. And we must enrich Money with a way to express "at least," so we add a "greater than or equal to" property to Money (we show it as a parameterized attribute with an operator-like name. It could be implemented directly as a function or an operator in class Money).



The postcondition says this:

postcondition
-- There is a new journey sale 's', for which
-- 's.journeyType' is that 'jt' for which
-- 'jt.here' is the given 'here'
-- 'jt.destination' is the given 'destination'
-- 'jt.journeyCategory' is the given 'category'
-- 's.tendered' equals 'tendered'
-- 's.change' equals 'tendered' minus the 'cost' of a journey of type 'jt'.

We need a type JourneySale with properties for "journey type", "tendered", and "change". We model two of these properties as attributes, and one as an association. We also need an "equals" property on type Money, as well as a "minus" property.



We have just explored an example of cross-checking from a specification of a piece of behavior to an object type model. The checks were that:

- Every argument must be of a type in the object type model.
- The terms in every expression in the precondition and every expression in the postcondition must map to properties defined in the type model.

Had we defined any invariants on the type model, there would be another check:

- Behavior specifications must not contradict invariants.

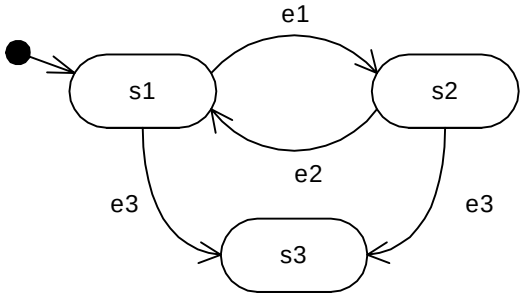
These checks apply to various kinds of behavior specifications: event specifications (in domain models), essential use case specifications (in system models), and method specifications (in design models).

Cross-checking

- Every expression in a specification of behavior can be cross-checked to concepts in a model of structure.
- Indeed, the model of structure can be "driven" by the need to model behavior.
- There are many other useful cross-checks, but this one is particularly powerful.

Constraining the UML

In this section, we continue the theme of cross-checking between models. In order to define effective cross-checks, we must sometimes constrain what UML means. We use state diagrams as an example.



Here is a UML diagram of a state machine forming part of a domain model. It describes objects of some type T, which can be in one of three states: s1, s2 or s3. Objects of type T make transitions from one state to another as a consequence of events e1, e2 and e3.

You can relate the state machine to the type T in different ways. UML doesn't tell you how to relate them. By choosing how they relate, we can build consistency-checks into the process of building models.

A reader from a real-time background might read the diagram as saying that objects of type T must have an event controller. Someone trained in Syntropy (Cook and Daniels 1994) might read the diagram as saying there must be dynamic subtypes of type T for the states s1, s2, s3. Someone trained in Catalysis (D'Souza and Wills 1999) might read the diagram as saying that type T has three boolean attributes, s1, s2 and s3, with an invariant to ensure mutual exclusion. But the interpretation we choose (because we have found it be extremely helpful when building models) is this: every state in the state diagram must be distinguishable from the other states using only existing properties of objects of type T. If the existing properties are inadequate for this task, the model of T needs more work.

In our experience, if you build state machines for key object types, and perform this cross-check very carefully, you will almost always learn something about the world you are modeling. Partly, this is because modeling with state machines entails a significant change of viewpoint. When you are building a model of types of objects, and the associated behavior that changes objects of those types, you are modeling the state of a world or a system, and how that state can change, but you are doing so over the whole system at one time. By contrast, when you build state machines for key

object types, you are still modeling state, and how state can change, but now you are doing it a type-at-a-time. This switch from an holistic view to a one-thing-at-a-time view often helps you to discover important facts you had previously overlooked.

More on cross-checking

- By design, UML is open to number of interpretations.
- By constraining how you interpret UML, you can get the benefit of more cross-checks between models.
- Cross-checks help us
  - ask the right questions during modeling
  - build models that make sense
  - test whether our models are mature
  - avoid silly inconsistencies between models.

Further Cross-Checks

This section lists further checks for consistency amongst UML's various models. Certain checks are annotated with remarks in [square brackets]. The list is not complete (for instance, we have omitted cross-checks concerning UML diagrams not covered in this article, such as activity diagrams and interaction diagrams).

Some of the checks are powerful aids to analysis. For example, building state machines for certain types of objects often reveals overlooked units of behavior. For example, modeling copies of books in a library might reveal that books can be returned after they have been recorded as lost, and books can be stolen while on hold against reservations. If you build systems that do not allow users to record the occurrence of such events, your users will rightly complain that the systems are uncomfortable to use.

Other checks simply tidy up the models and remove small inconsistencies of naming, and so on.

The cross-checks are divided into sections according to what kind of model is being cross-checked against what other kind.

From example to generalization

This is a meta-level check, acknowledging the importance in the InferData approach of working at two levels: the level of particular examples, and the level of generalizations, or rules.

- Every model at the example level must conform to the corresponding general-level model. For example, snapshots must conform to type models.

From snapshot to object type model

[The checks in this section are examples of what might be called syntax checks on diagrams.]

- Every object must be an instance of a type in the type model.
- Every link in a snapshot must be an instance of an association in the type model.

- In a snapshot, the number of links between objects must agree with multiplicity constraints on the type model.
- Every role name on a link in a snapshot must match the role name on the corresponding association end in the type model.

From snapshot to scenario

- For every snapshot, it must be possible to construct a scenario leading to the snapshot. [Otherwise, you might be overlooking some behavior.]

From behavior specification to type model

(Behavior specifications are event specifications, essential use case specifications, and method specifications.)

- Every argument must be of a type in the type model.
- The terms in every expression in the precondition and the postcondition must be defined in the type model. [This is a powerful check. Avoid describing behavior without building a type model, and then always apply the cross-check.]
- Behavior specifications must not contradict invariants.

From behavior specification to snapshot

- The precondition constrains the before state. It must be possible to construct a snapshot satisfying the precondition and conforming to the type model.
- The postcondition constrains the change from the before state to the after state. It must be possible to construct a pair of snapshots satisfying the postcondition and conforming to the type model.

From scenario to behavior list

- Every interesting change of state in a scenario must correspond to a unit of behavior. [Devising scenarios is a powerful way to test models that are thought to be mature. Because of other checks, testing models of behavior also tests models of structure.]

From type model to behavior list

[These are examples of simple checks that can occasionally yield useful insights.]

- For each type of object there must be a unit of behavior in which instances of the type are created.
- For each attribute there must be a unit of behavior in which the value of the attribute is set.
- For each association there must be a unit of behavior in which instances of the association (i.e., links) are created.
- For each property that is not constant there must be a unit of behavior in which the value of the property is changed.

From state diagram to type model

- Every transition must be associated with a unit of behavior. [Building state machines often reveals overlooked units of behavior.]
- Every state must be distinguishable from all other states using only properties defined on the type model. [A very powerful check. Usually yields important insights, albeit detailed ones.]

- For every state and for every event, the next state must be defined, either visually on the state diagram or by a rule (for example, the Syntropy-style “accept all events”).

(On state diagrams with nested states, apply the checks to the lowest-level states. On state diagrams with concurrent states, an object’s state is defined by one state from each concurrent section. Apply the checks accordingly.)

Within a state diagram

- There must be a starting state.
- Every state with no outgoing transitions must model a terminal state in the world being modeled.

Within a type model

- If the set of objects reached by one path through a type model has a fixed relationship to the set reached by another path (e.g., they are the same set), there must be an invariant to capture this fact.

Testing models

We have suggested in various places that models can be tested. In this short section, we review two important ways to test models, one of which concentrates on structure and the other on behavior. Then we show how they can work together.

Testing structure

An object type model is a general-level model of object types, their attributes, and their associations to other types. The corresponding example-level model is a snapshot. A type model can be tested by devising valid snapshots and checking whether they are allowed by the type model, and by devising invalid snapshots and checking whether they are disallowed by the type model. The mind-set needed to perform this testing is exactly that needed to test code. If a model is thought to be mature, devise snapshots to try to break the model. If you fail to break it, you increase your confidence in the model.

Testing behavior

In our approach, behavior is captured in event specifications in a domain model, and essential use case specifications in a system model. Each specification of a unit of behavior will have a precondition and a postcondition.

A scenario is a sequence of happenings in the world (you can think of it as a storyboard). The behavioral aspects of a domain model can be tested by devising scenarios, and checking whether the relevant steps in these scenarios are occurrences of events of known event types. Irrelevant steps are those that can be argued to be out of scope. The behavioral aspects of a system model can be tested by devising scenarios and checking whether the relevant steps are occurrences of events for which there are corresponding essential use cases.

Combining the tests

You can test structural and behavioral aspects of your models by running scenarios and checking all the snapshots you pass through. There are two phases. First, devise a test. To do this, devise a starting situation, and devise a scenario that begins in this starting situation. Then apply the test. To do this, first draw a snapshot showing the starting situation. Then, for each relevant step in the scenario, draw another snapshot to show the change this step brings about. Along the way, check that every snapshot you draw is allowed by the model of object types, that every step is modeled by a unit of behavior (event or essential use case), and that the specification of each unit of behavior correctly describes the change the step brings about.

Example

Here is a scenario for a video hire store, used to test a domain model.

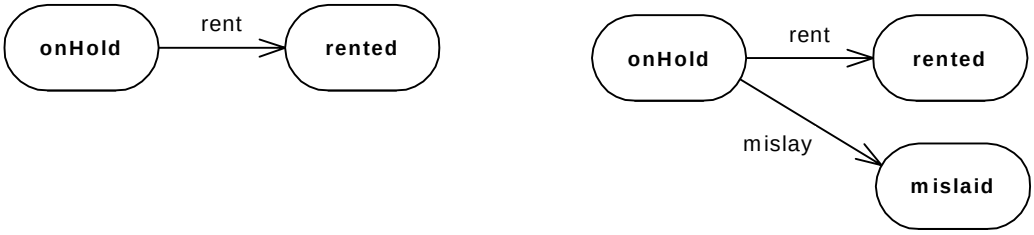
Starting state:

- It is June 4th, 2001
- Kim has copy 12 of Titanic out on rent
- All other copies of Titanic are also out on rent.

Steps in the scenario:

1. Peter reserves Titanic on June 5th, 2001
2. Kim returns copy 12 of Titanic on June 6th, 2001
3. LowPrice puts the copy on hold for Peter
4. Peter receives a notice on June 7th that LowPrice is holding a copy of Titanic
5. Peter calls at the store and asks to rent Titanic
6. LowPrice staff cannot find the copy

Imagine that, in the state model for video copy objects, the only way a copy can leave the "onHold" state is in a "rent" event, as in the left-hand model below. We have just discovered that our model is not a faithful-enough model of the world. We must add an event "mislay," and a state "mislaid," as in the right-hand model.



(There is, of course, yet more work to be done. For instance, we'll need to add a further event to model finding a mislaid copy.)

We now can rewrite a portion of the scenario:

4. Peter receives a notice on 7th June that LowPrice is holding a copy of Titanic
5. (new) LowPrice mislays copy 12 of Titanic
6. Peter calls at the store and asks to rent Titanic

7. The copy is not available to rent, because it has been mislaid

During systems analysis, we determine that it is acceptable to detect an occurrence of the "mislay" event later than when it occurred, specifically, when a customer comes to collect a copy on hold. During the design of the user interaction, we make sure to include the ability to mark a copy as mislaid as part of the interaction concerned with checking out a copy on hold.

Testing

- General-level models include type models, state models and event specifications.
- Example-level models include snapshots and scenarios.
- These example-level models can be used to test general-level models.
- Testing can
  - uncover areas that need further modeling
  - reveal inconsistencies between models
  - indicate the maturity of models.

Coherence

This final section brings together the content of earlier sections, with the goal of showing that we have presented a coherent approach to modeling—its different elements work in concert.

In the section entitled Architectures, we showed that you can divide a model up along technical lines, so that, for instance, you can model the core object types and the core functionality separately from modeling how this functionality is delivered to users. In the section entitled Use Case Descriptions we showed that you can describe functionality abstractly, without including unnecessary design decisions. It is much easier to work with these abstract descriptions if you know that they belong in the core domain, and that somewhere else there is an interaction domain in which to design user interactions with this core. Our architectural models support separation of concerns in modeling.

In the section entitled Connecting Behavior to Structure we showed how to build a model of object types to provide a vocabulary for the expressions in the precondition and postcondition of a specification of a use case. And we have mentioned that a project dictionary would give this vocabulary meaning (although we haven't discussed how to write the necessary definitions). There are three layers at work here. In the middle is a type model that introduces the terms we need in our vocabulary. Above it is a layer in which we use the terms to build models of behavior, by writing preconditions and postconditions that are cross-checked to the type model. Beneath the middle layer is a supporting layer containing the definitions of our terms.

In the section entitled Constraining the UML, we noted that switching from thinking about all the types in a model to thinking about them one at a time can free the mind to uncover new insights. In the section entitled Further Cross-Checks, we noted that there is a meta-level check, that all example-level diagrams agree with the corresponding general-level diagram. Once again, being able to switch between examples and generalizations can free the mind to uncover new insights. And the cross-checks between examples and generalizations ensure that there is still only one underlying model.

In the section entitled Development Activities we were careful not to prescribe the order in which you do domain analysis, systems analysis and design. For example, on your next project, it might be sensible to do a small amount of systems analysis, to find out roughly what is wanted. Then you might do some design, to check that you can make two technologies work together, or to try out a clever algorithm from a handbook. Then you might devote some effort to domain modeling, to make sure you understand the context your software will go into. While building a domain model, you wonder whether to explore some particular aspect of the business domain. You talk to the client to see whether there are plans to automate that aspect—in other words, you go back to doing some systems analysis. Generally, it is not helpful to prescribe an order in which developers should perform activities. It is more helpful to explain what the different activities address and how they interrelate. Every time a developer asks a question or gives an answer, he or she should care whether it concerns the world, the requirements for a system in that world, or a design for such a system. But a developer should not be prevented from asking a question about how the world works because "we are now in the design phase." Nor should a question about how the world works be treated as a design question just because "we are now in the design phase."

Summary

- Distinguish
  - modeling the world
  - modeling requirements on a system
  - modeling designs.
- Use architectural models that support this separation of concerns.
- Model from different points of view (types, events/use cases/methods, states, objects, …).
- Never get stuck—just come at the problem from a new point of view, but …
- Continually cross-check your models.

References

Booch G, Rumbaugh J and Jacobson I (1999). *The Unified Modeling Language User Guide*. Addison Wesley.

Describes the various elements of the UML, and offers advice on how to use them.

Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F and Jeremaes P (1994). *Object-oriented development. The Fusion method*. Prentice Hall.

The Fusion method showed clearly that developers could think about object-oriented systems without thinking in terms of message passing between objects.

Constantine L and Lockwood L (1999). *Software for Use*. Addison Wesley.

Contains excellent advice on modeling systems that will be easy to use. Carefully distinguishes essential use cases from more concrete, interaction-oriented use cases.

Cook S and Daniels J (1994). *Designing object systems. Object-oriented modelling with Syntropy.* Prentice Hall.

The Syntropy method made it clear that we can model three things separately: the world, systems as seen from the outside, and the internal workings of software. When not modeling the internal workings of software, Syntropy uses events to model behavior.

D'Souza D and Wills A (1999). *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley.

Like Fusion and Syntropy before it, Catalysis puts strong emphasis on modeling the types of objects of interest, and using such models as the basis of describing behavior.

Fowler M (1997). UML distilled. *Applying the standard object modeling language*. Addison Wesley (1st edition).

An excellent, short introduction to the UML. There is now a second edition, but the quotation is from the first edition (and we've checked that the author hasn't changed his mind about what the quotation says!)

Jackson M (1983). *System Development*. Prentice Hall.

Describes JSD, which is not a fully object-oriented approach. However, the book is a clear introduction to the principles of modeling.

Jacobson I, Booch G and Rumbaugh J (1999). *The Unified Software Development Process*. Addison Wesley.

Describes Rational Corporation's proprietary process, which uses UML. The process is centered around use cases.

OMG (1999). *OMG Unified Modeling Language Specification*. Object Management Group (Version 1.3).

The OMG's official definition of the UML (available on-line).