9. Rationale

"Don't keep doing what doesn't work."

– Anonymous

Rationale is the justification of decisions. The models we have described until now represent the system. Rationale models represent the reasoning that lead to the system, including its functionality and its implementation. Rationale includes:

- the issues they address,
- the alternatives that were considered,
- the decisions that were made to resolve the issues,
- the criteria that were used to guide decisions, and
- the debate developers went through to reach a decision.

In the context of change, rationale is the most important information in the development process. For example, when requirements change and functionality is added to the system, the rationale enables developers to track which decisions need to be revisited and which alternatives have already been evaluated. When new staff is assigned to the project, new developers can become familiar with past decisions by accessing the rationale of the system.

Unfortunately, rationale is also the most complex information developers generate, and thus, the most difficult to maintain and update. Moreover, capturing rationale and maintaining it up to date represents an up front investment with long term returns. In this chapter we describe issue-modeling, a representation for modeling rationale. We then describe the activities of creating, maintaining, and accessing rationale models. We conclude this chapter by describing management issues related to maintaining rationale models.

9.1. Introduction: a meat loaf example

System models are abstractions of what the system does. The requirements analysis model, including the use case model, the use case model, and the sequence diagrams (see Chapter 6, *Requirements Elicitation* and Chapter 7, *Requirements Analysis*) represents the behavior of the system from the user's point of view. The system design model (see Chapter 8, *System Design*) represents the system in terms of subsystems, design goals, hardware nodes, data stores, access control, and so on. The rationale model represents why a given system is structured and behaves the way it does.[1] Why should we capture the *why*? Consider the following example:[2]

> Mary asks John, her husband, why he always cuts off both ends of the meat loaf before putting it in the oven. John responds that he is following his mother's recipe and that he had always see her cut the ends of the loaf. He never really questioned the practice and thought it was part of the recipe. Mary, intrigued by this answer, calls her mother in law to find out more about this meat loaf recipe.
>
> Ann, John's mother, provides more details on the meat loaf cutting, but no culinary justification: she says that she has always trimmed about an inch off each end of the loaf as her mother did, assuming it had something to do with improving the taste.
>
> Mary continues her investigation and calls John's maternal grandmother, Zoe. At first, Zoe is very surprised: she does not cut the ends of the meat loaf and she cannot imagine how such practice could possibly improve the taste. After much discussion, Zoe eventually remembers that, when Ann was a little girl, she used to cook on a much narrower stove which could not accommodate standard sized meat loaves. To work around this problem, she used to cut off about an inch from each end of the loaf. She stopped this practice once she got a wider stove.

Developers and cooks are good at disseminating new practices and techniques. The rationale behind these techniques, however, is usually lost, making it difficult to improve them as their application context changes. The year 2000 bug is such an example: in the sixties and seventies, memory costs drove developers to represent information as compactly as possible. For this reason, the year was often represented with two characters instead of four (e.g., '1998' was represented as '98'). The assumption of the developers was that the software would only be used for a few years. Arithmetic operations on years represented with two digits assumes that all dates are within the same century. Unfortunately, this shortcut breaks down at the turn of the century for software performing arithmetic on two digit years. For example, when computing the age of a person, a person born in 1949 will be considered 01 - 49 = -48 years old in 2001. The practice of encoding years with two digits became standard, even after memory prices dropped significantly and the year 2000 came

1. Historically, much research about rationale focuses on design and, hence, the term *design rationale* is most often used in the literature. Instead, we use the term *rationale* to avoid confusion and to emphasize that rationale models can be used during all phases of development.
2. Adapted for this chapter, original author unknown.

nearer. Moreover, new systems needed to be backward compatible with older ones. For these reasons, many systems delivered as late as the nineties still have 2000 year bugs.

Rationale models enable developers and cooks to deal with *change*, such as larger stoves or cheaper memory prices. Capturing the justification of decisions effectively models the dependencies between starting assumptions and decisions. When assumptions change, decisions can be revisited. In this chapter, we describe techniques for capturing, maintaining, and accessing rationale models. In this chapter, we

- provide you with a bird view of the activities related with rationale models (Section 9.2),
- describe issue modeling, the technique we use for representing rationale (Section 9.3),
- detail the activities necessary for creating and accessing rationale models (Section 9.4), and
- describe management issues related with maintaining rationale models (Section 9.5).

First, let us define the concept of rationale model.

9.2. An overview of rationale

A ***rationale*** is the motivation behind a decision. More specifically, it includes:

- ***Issue.*** To each decision correspond an issue that needed to be solved for the development to proceed. An important part of the rationale is a description of the specific issue that is being solved. Issues usually phrased as questions: How should a meat loaf be cooked? How should years be represented?
- ***Alternatives***. Alternatives are possible solutions that could address the issue under consideration. These include alternatives that were explored but discarded because they did satisfy with one or more criteria. For example, buying a wide stove costs too much. Representing years with a binary sixteen bit number requires too much processing.
- ***Criteria***. Criteria are desirable qualities that the selected solution should satisfy. For example: A recipe for meat loaf should be realizable on standard kitchen equipment. Developers in the sixties minimized memory foot print. During requirements analysis, criteria are nonfunctional requirements and constraints (e.g., usability, number of input errors per day). During system design, criteria are design goals (e.g., reliability, response time). During project management, criteria are management goals and trade-offs (e.g., timely delivery vs. quality).
- ***Argumentation***. Cooking and software development decisions are not algorithmic. Cooks and developers discover issues, try solutions, and argue their relative benefits. It is only after much argumentation that a consensus is reached or a decision imposed. This argumentation, including argumentation on the criteria, the justifications, the explored alternatives, and the trade-offs to be made is part of the rationale.
- ***Decisions.*** A decision is the resolution of an issue, it represents the selected alternative according to the criteria that were used for evaluation and the justification of the selection. Cutting an inch off each end of a meat loaf and representing years with two digits are decisions. Decisions are already captured in the system models we develop during requirements analysis and system design. Moreover, many decisions are made without exploring alternatives or examining the corresponding issues.

We make decisions throughout the development process, and thus, we can use rationale models during any development activity:

- During *requirements elicitation* and *requirements analysis*, we make decisions about the functionality of the system, most often with the client. Decisions are motivated by user or organizational needs. The justification of these decisions is useful for creating test cases during system integration and user acceptance.

- During *system design*, we select design goals and design the subsystem decomposition. When identifying design goals, for example, we often base our decision on nonfunctional requirements. Capturing the rationale of these decisions enables us to trace dependencies between design goals and nonfunctional requirements. This allows us to revise the design goals when requirements change.
- During *project management*, we make assumptions about the relative risks present in the development process. We are more likely to start development tasks related to a recently released component as opposed to a mature one. Capturing the justifications behind the risks and the fallback plans enable us to better deal when these risks become actual problems.
- During *integration and testing*, we discover interface mismatches between subsystems. Accessing the rationale for the subsystems, we can often determine which change or assumption introduced the mismatch and correct the situation with minimal impact on the rest of the system.

Maintaining rationale is an investment of resources for dealing with change: we capture information *now* in order to make it easier to revise decisions *later*, when changes occur. The amount of resources we are willing to invest depends on the type of project.

If we are building a complex system for a single customer, we will most likely revise and upgrade the system several times over a long period. In this case, the client may even require that rationale be recorded. If we are building a conceptual prototype for a new product, we will most likely throw the prototype once the product development is approved and underway. If we divert development resources to record rationale, we risk delaying the demonstration of the prototype and face the project cancellation altogether. In this case, we do not record rationale since the return on such an investment would be minimal.

More generally, we distinguish four levels of rationale capture:

- *No explicit rationale capture.* Resources are spent only on development. The documentation focuses on the system models only. Rationale information is present only in the developers' memories and in communication records such as email messages, memos, and faxes.
- *Rationale reconstruction.* Resources are spent in recovering design rationale during the documentation effort. The design criteria and the motivation behind major architectural decisions is integrated with the corresponding system models. Discarded alternatives and argumentation are not captured explicitly.
- *Rationale capture.* Major effort is spent in capturing rationale as decisions are made. Rationale information is documented as a separate model and cross referenced with other documents. For example, the motivation for the requirements analysis model is captured in the *Requirements Analysis Rationale Document* (RARD), complementing

the *Requirements Analysis Document* (RAD). Similarly, the motivation for the system design is captured in the *System Design Rationale Document* (SDRD).

- *Rationale integration.* The rationale model becomes the central model developers use. Rationale produced during different phases are integrated into a live and searchable information base. Changes to the system occur first in the information base as a discussion followed by one or more decisions. The system models represent the sum of the decisions captured in the information base.

In the first two levels of rationale capture, *No explicit rationale capture* and *Rationale reconstruction,* we rely on developers memory to capture and store rationale. In the last two levels, *Rationale capture* and *Rationale integration,* we invest resources into constructing an corporate memory that is independent from the developers. The trade-off between these two extremes is the investment of resources during the early phases of development. In this chapter, we focus on the last two levels of rationale capture.

In addition to long term benefits, maintaining rationale can also have short term positive effects: making explicit the rationale of a decision enables us to understand better the criteria others follow. It also encourages us to take rational decisions instead of emotional ones. If nothing else, it helps us distinguish which decisions were carefully evaluated and which were made under pressure and rushed.

Rationale models represent a larger and faster changing body of information than the system models. This introduces issues related to complexity and change as we have seen previously. Hence, we can apply the same modeling techniques for dealing with complexity and change. Next, we describe how we represent rationale with issue models.

9.3. Issue modeling

In this section, we describe issue models, the representation we use for rationale. Issue modeling is based on the assumption that design occurs as a dialectic activity during which developers solve a problem by arguing the pros and cons of different alternatives. We can then capture rationale by modeling the argument that lead to the development decisions. We represent:

- a question or a design problem as an issue node (Section 9.3.2),
- alternative solutions to the problem as proposal nodes (Section 9.3.3),
- pros and cons of different alternatives using argument nodes (Section 9.3.4), and
- decisions we make to resolve an issue as a resolution node (Section 9.3.5).

In Section 9.3.7, we survey several issue representations of historical significance. But first, let us talk about central traffic control, the domain for the examples in this chapter.

9.3.1. Central traffic control

Central traffic control (CTC) systems enable train dispatchers to monitor and route trains remotely. Train tracks are divided into contiguous track circuits which represent the smallest unit a dispatcher can monitor. Signals and other devices ensure that at most one train can occupy a track circuit at any time. When a train enters a track circuit, a sensor detects its presence and the train identification appears on the dispatcher's monitor. The dispatcher operates switches to route trains. The system enables a dispatcher to plan a complete route by aligning a sequence of switches in the corresponding position. The set of track circuits controlled by a single dispatcher is called a track section. Usually, a single dispatcher accesses a given track section at once.

Figure 9-1 is a simplified display of a CTC user interface. Track circuits are represented by lines. Switches are represented by the intersection of three lines. Signals are represented with icons indicating whether a signal is open (i.e., allowing a train to pass) or closed (i.e., forbidding a train to pass). Switches, trains, and signals are numbered for reference in commands issued by the dispatcher. In Figure 9-1, signals are numbered **S1** through **S4**, switches are numbered **SW1** and **SW2**, and trains are numbered **T1291** and **T1515**. Computers near the tracks, called wayside stations, ensure that the state of a group of switches and signals do not present any safety hazard. For example, a wayside station controlling the devices of Figure 9-1 ensures that opposing signals, such as **S1** and **S2**, cannot be open simultaneously. Wayside stations are designed such that the state of the device they control is safe even in the case of failure. Such equipment is called fail-safe. CTC systems communicate with wayside stations to modify the state of the tracks when dispatching

trains. CTC systems are typically highly available but need not be fail-safe, given that the safety of trains is guaranteed by the wayside stations.

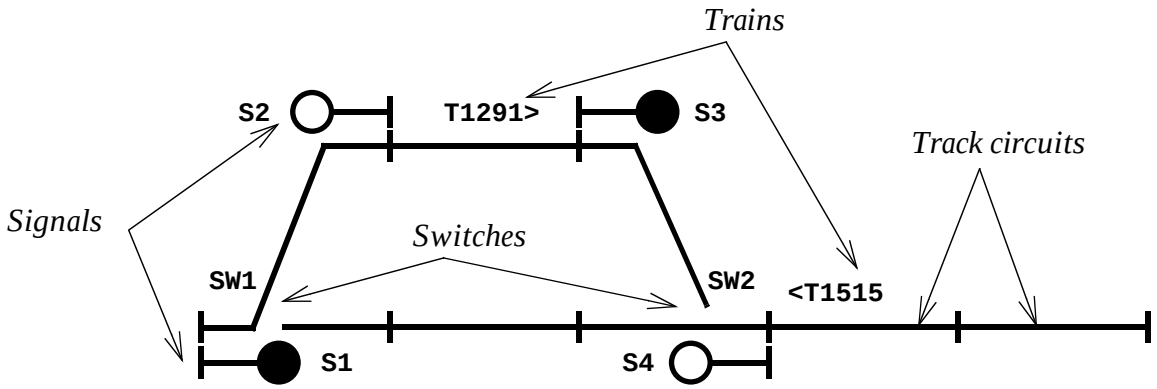


FIGURE 9-1. An example of a CTC track section display (simplified for this example).

In the sixties, CTC systems had a custom display board containing light bulbs displaying the status of the track circuits. Switches and signals were controlled via an input board with many push buttons and toggle switches. In the seventies, CRTs replaced the custom boards and provided dispatchers more detailed information with less real estate. More recently, workstation-based traffic control systems have been introduced, offering the possibility of a more sophisticated user interface to dispatchers and the ability to distribute processing among multiple computers.

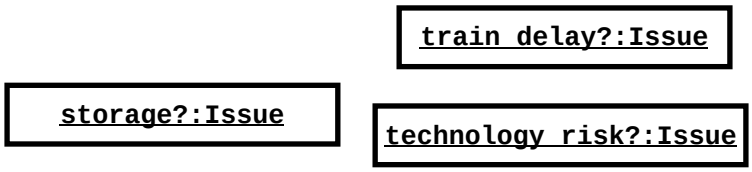
Central traffic control systems need to be highly available. Although traffic control systems are not life critical (safety is ensured by wayside stations), a failure of the system can lead to major traffic disruption in the controlled tracks, thus resulting in a substantial economic loss. Consequently, the transition to a new technology, such as moving from a mainframe to a workstation environment or moving from a textual interface to a graphical user interface, needs to be carefully evaluated and done much more slowly than for other systems. Traffic control is a domain in which capturing rationale is critical, and thus, serves as the basis for the examples of this chapter.

Let us discuss next how issue models are used to represent rationale.

9.3.2. Defining the problem: issues

An ***issue*** represents a concrete problem, such as a requirement, a design, or a management problem. *How soon should a dispatcher be notified of a train delay? How should persistent data be stored? Which technology presents the most risk?* Issues most often represent problems that do not have a single correct solution and that cannot be resolved algorithmically. Issues are typically resolved through discussion and negotiation.

We represent issues in UML with instances of class **`Issue`**. **`Issues`** have a **`subject`** attribute, summarizing the issue, a **`description`** attribute, describing the issue in more detail and referring to supporting material, and a **`status`** attribute, indicating whether the issue has been resolved or not. The **`status`** of an issue is ***open*** if it has not be resolved yet and ***closed*** otherwise. A closed issue can be re-opened if an issue needs to be revisited. By convention, we give a short name to each issue, such as `train delay?:Issue` for reference. For example, Figure 9-2 depicts the three issues we gave as example in the previous paragraph.



`train delay?:Issue`	How soon should a dispatcher be notified of a train delay?
`storage?:Issue`	How should persistent data be stored?
`technology risk?:Issue`	Which technology presents the most risk?

FIGURE 9-2. An example of issues. (UML object diagram).

Issues raised during development are often related. For example, issues can be decomposed into smaller ***subissues***. *What are the response time requirements of the traffic control system?* includes *How soon should a dispatcher be notified of a train delay?* The complete system development can be phrased as a single issue: *Which traffic control system should we build?* that can then be decomposed into numerous subissues. Issues can also be raised by decisions made on other issues. For example, the decision to cache data on a local node raises the issue of maintaining consistency between the central and cached copies of the data. Such issues are called ***consequent issues***.

Consider the central traffic control system we previously described. Assume, we are currently examining the transition from a mainframe system to a desktop based system. In

the future system, each dispatcher will have an individual desktop machine which communicates with a server which manages the communication with field devices. During design discussions, two interface issues are raised: *How should commands be input to the system?* and *How should track circuits displayed to the dispatcher?* Figure 9-3 depicts two issues represented with a UML object diagram.

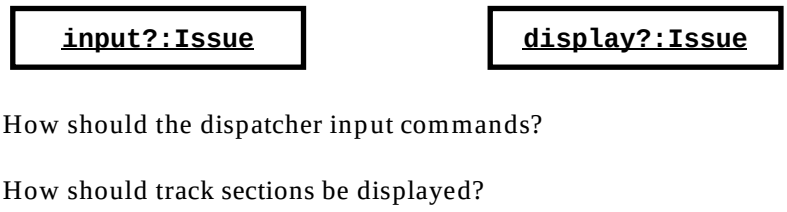


FIGURE 9-3. CTC interface issues. (UML object diagram).

An issue should only focus on the problem, not possible alternatives addressing it. A convention that encourages this is to phrase issues as questions. To reinforce this concept, we also include a question mark at the end of the issue name. Information about the possible alternatives addressing an issue are captured by proposals, which we discuss next.

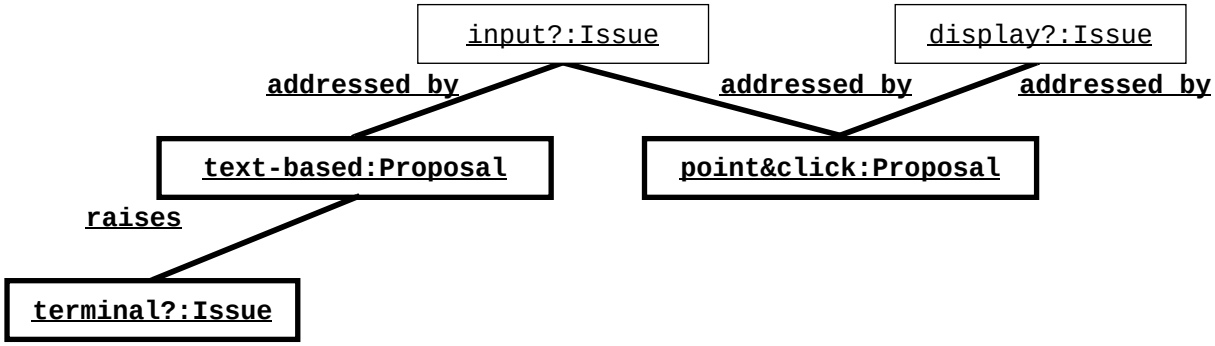
9.3.3. Exploring the solution space: proposals

A ***proposal*** represents a candidate answer to an issue. *A dispatcher need not be notified* is a proposal to the issue *How soon should a dispatcher be notified of a train delay?* A proposal need not be a good or valid answer to the issue it addresses. This enables developers to explore the solution space thoroughly. Often when brainstorming, proposing a flawed solution triggers new ideas and solutions which would not have be thought of otherwise. Different proposals addressing the same issue can overlap. For example, proposals to the issue *How to store persistent data?* could include *Use a relational database* and *Use a relational database for structured data and flat files for images.* Proposals are used to represent the solution to the problem as well as the discarded alternatives.

A proposal can address one or more issues. For example, *Use a Model/View/Dispatcher architecture* can address *How to separate interface objects from entity objects?* and *How to maintain consistency across multiple views?* Proposals can also trigger new issues. For example, in response to the issue *How to minimize memory leaks?* the proposal *Use garbage collection* may trigger the consequent issue *How to minimize response time degradation due to memory management?* When we address an issue, we need to ensure that all consequent issues associated with the selected proposals are addressed as well.

We represent proposals in UML as instances of the class **Proposal**. **Proposals**, like **Issues**, have a **subject** and a **description** attributes. By convention, we give proposals a short name and phrase them as a statement starting with a verb. **Proposals** are related to the **Issues** they address with an addressed by association. **Issues** are related to **Proposals** that triggered them with a raises association.

While discussing the interface issues of our central traffic control system, we consider two proposals, a point & click interface, which allows track circuits to be represented graphically, and a text-based interface, in which track sections are represented with special characters. The text-based proposal raises a consequent issue about which terminal emulation to use. Figure 9-4 depicts the addition of the two proposals and the consequent issue.



point&click:Proposal	The interface for the dispatcher could be realized with a point&click interface.
text only:Proposal	The display used by the dispatcher can be a text only display with graphic characters to represent track segments.
terminal?:Issue	Which terminal emulation should be used for the display?

FIGURE 9-4. An example of proposals and consequent issue. (UML object diagram).

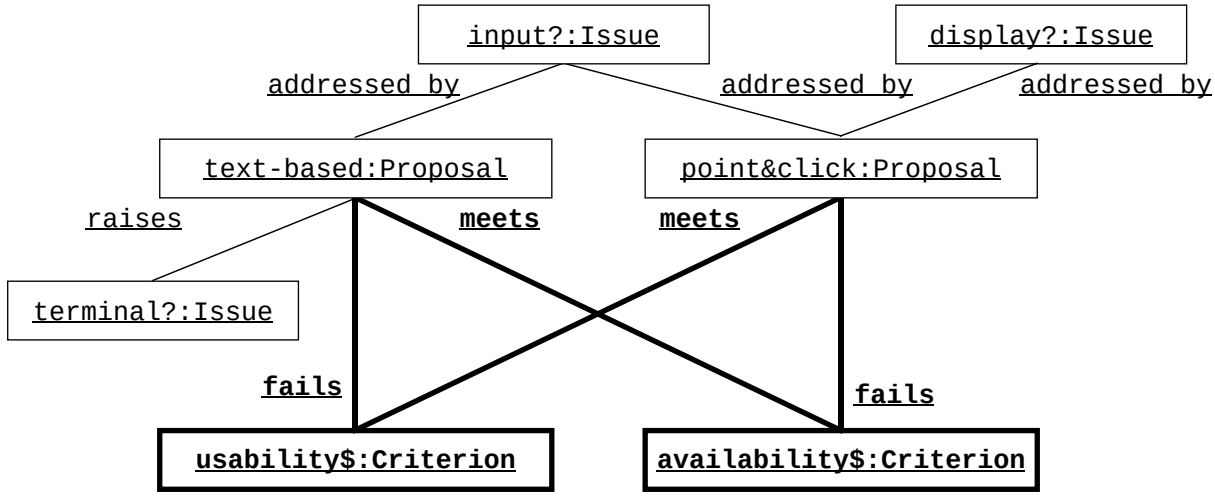
A proposal should only contain information related to the solution, not its value, advantages, and disadvantages. Criteria and arguments are used for this purpose. We describe these next.

9.3.4. Evaluating the solution space: criteria and arguments

A ***criterion*** is a desirable quality that proposals addressing a specific issue should have. Design goals, such as *response time* or *reliability*, are criteria used for addressing design issues. Management goals, such as *minimum cost* or *minimum risk*, are criteria used in addressing management issues. A set of criteria indicates the dimensions against which each proposal needs to be assessed. A proposals that meets a criterion is said to be ***assessed positively*** against that criterion. Similarly, a proposal that fails to meet a criterion is said to be ***assessed negatively*** against that criterion. Criteria may be shared among several issues.

We represent criteria in UML as instances of the **`Criterion`** class. **`Criteria`**, like **`Issues`** and **`Proposals`**, have a **`subject`** and a **`description`** attribute. The **`subject`** attribute is always be phrased positively, that is, state the quality that proposals should maximize. *Fast*, *responsive*, and *cheap* are good **`subject`** attributes. *Cost* and *time* are not. **`Criteria`** are associated to **`Proposals`** with **`assessment`** associations. Assessment associations have a value attribute whether the assessment is positive or negative, and a weight attribute, indicating its strength of the proposal with respect to the criterion. By convention, we append a $ sign to the end of a criterion name, emphasizing that criteria are goodness measures and should not be confused with arguments or issues.

While evaluating the interface of our central traffic control system, we identify two criteria, *availability*, which represents the nonfunctional requirement to maximize the up time of the system, and *usability*, which represents (in this case) the nonfunctional requirement to minimize the time to input valid commands (see Figure 9-5). These criteria are taken from the nonfunctional requirements of the system. We assess both proposals against these criteria: we decide that the point and click interface is negatively assessed against the availability criteria, being more complex than the text interface, and thus, presenting a higher likelihood of bugs. We decide, however, that the point and click interface is more usable than the textual interface, due to an easier selection of commands and input of data. Note that the set of associations linking the proposals and the criteria in Figure 9-5 represent a trade-off: each proposal maximizes one of the two criteria, the issue is to decide which criteria has a higher priority.



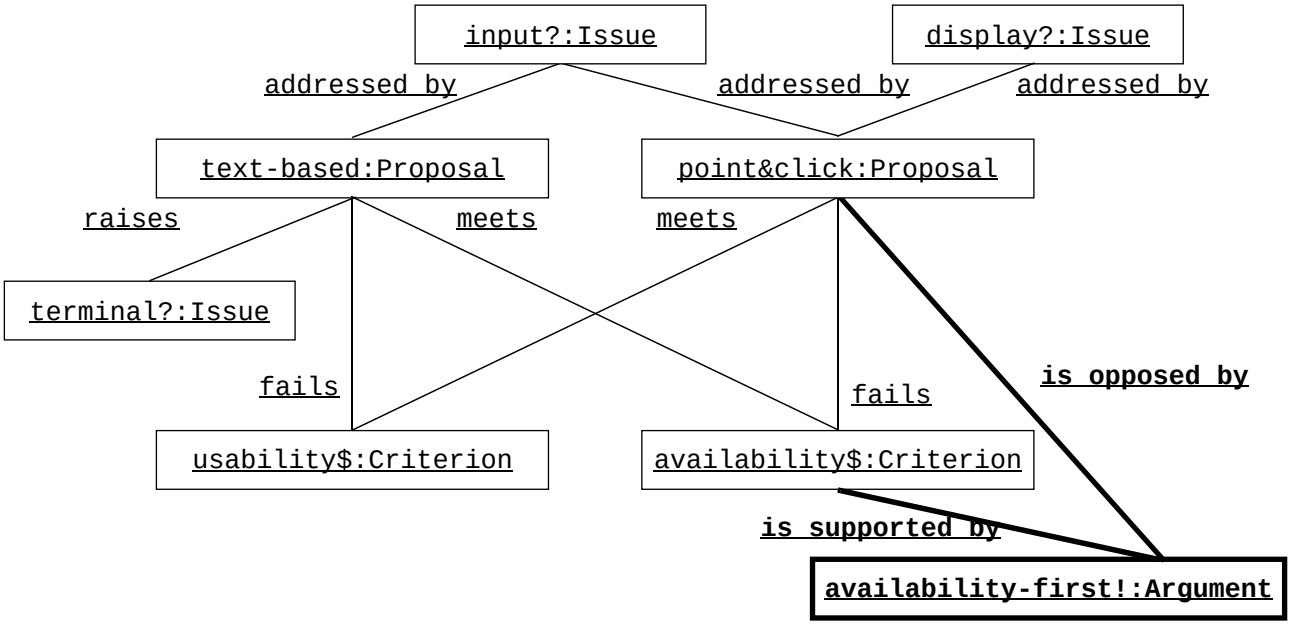
availability$:Criterion The traffic control system should have at least a 99% availability.

usability$:Criterion The time to input commands should be less than two seconds.

FIGURE 9-5. An example of criteria and assessments. (UML object diagram). A negative assessment is indicated by an association labeled fails while positive assessments are indicated with an association labeled meets.

An ***argument*** is an opinion expressed by a person, agreeing or disagreeing with a proposal, a criterion, or an assessment. Arguments capture the debate that drives the exploration of the solution space, defines the goodness measures, and eventually leads to a decision. We represent arguments in UML with instances of the class **Argument** including **subject** and **description** attributes. Arguments are related to the entity they discuss with a is supported by or an is opposed by association.

While discussing the relative priority of the availability and usability criteria, we decide that any benefit on the usability aspect would be offset by a reduced availability of the system. We capture this by creating an argument that supports the availability criterion (see Figure 9-6). Note that an argument can simultaneously support a node while opposing another.



availability-first!:Argument Point and click interfaces are much more complex to implement than text-based interfaces. They are also more difficult to test as the number of actions available to the dispatcher is much larger. The point and click interface risks introducing fatal errors in the system that would offset any usability benefit the interface would provide.

FIGURE 9-6. An example of an argument. (UML object diagram).

When selecting criteria, assessing proposals, and arguing about them, we evaluate the design space. The next step is to use this evaluation to come to closure and resolve the issue.

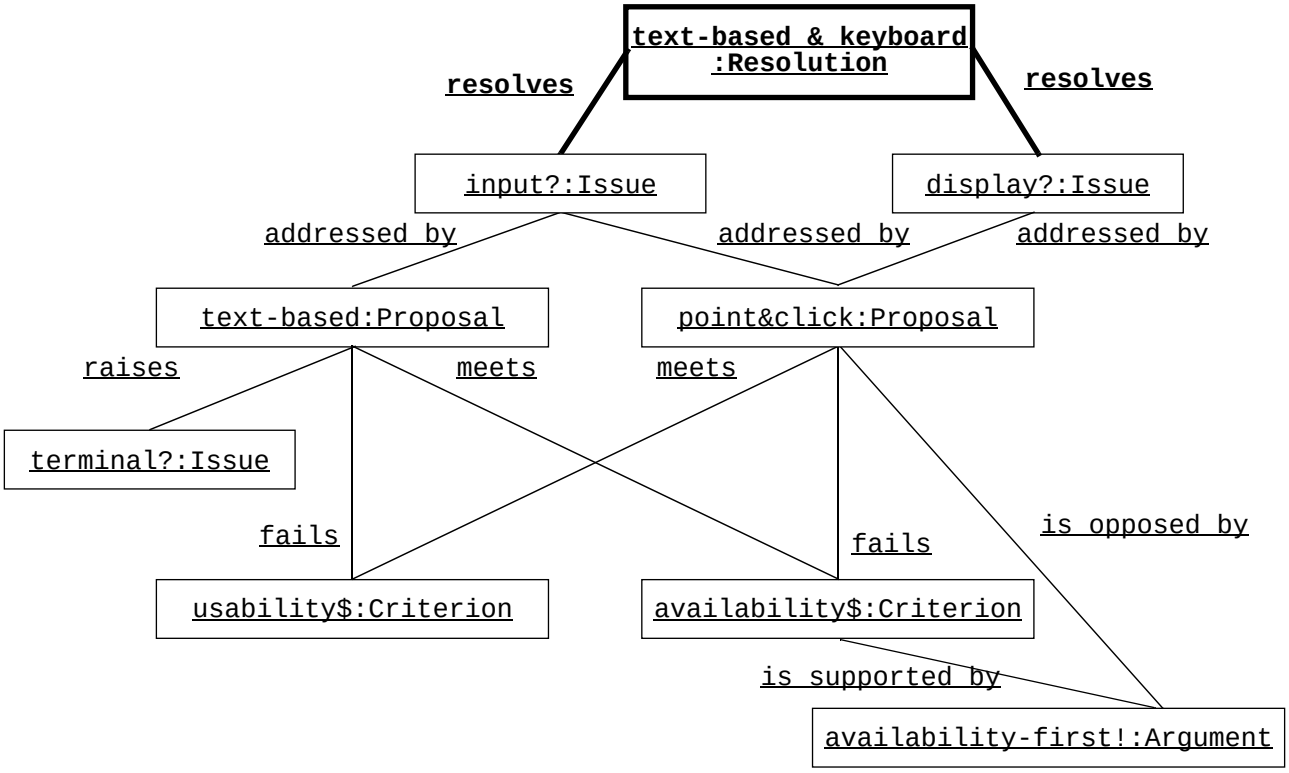
9.3.5. Collapsing the solution space: resolutions

A ***resolution*** represents the alternative selected to close an issue. A resolution represents a decision and has an impact on one of the system models or on the task model. A resolution can be based on several proposals and summarizes the justification that lead to the decision. We represent resolutions in UML with an instance of class **`Resolution`**, including **`subject`**, **`description`**, **`justification`** and **`status`** attributes. A **`Resolution`** can be related with

Proposals with **based-on** associations. A **Resolution** has exactly one **resolves** association to the **Issue** it resolves.

The **status** attribute of a **Resolution** indicates whether the Resolution is still relevant or not. When the Resolution is linked with its corresponding issue, its status is set to **active** and the **status** of the corresponding **Issue** is changed to **closed**. If the **Issue** is reopened, the **status** of the **Issue** is changed to **open** and the **status** of the **Resolution** is changed to **obsolete**. A closed **Issue** has exactly one active **Resolution** and any number of obsolete **Resolutions**.

Finalizing the traffic control interface issue, we select a text-based display and a keyboard interface as a basis for the user interface. This decision is motivated by treating the availability criterion as more important than the usability criterion: a text-based interface will result in much simpler and more reliable user interface code at the cost of some usability. The dispatcher will not be able to see as much data at one time and will not be able to issue commands as fast as using a point and click interface. We create a resolution node which contains the justification of the decision and create links between the resolution and the two issues it addresses (see Figure 9-7).



text-based & keyboard:Resolution

We select a text-based display and a keyboard input for the traffic control user interface. The terminal emulation should provide line characters allowing the drawing of track circuits in text mode. This decision is motivated by the relative simplicity and reliability of text-based interfaces compared to point&click interfaces. We are aware that this decision costs some usability, as fewer data can be presented to the dispatcher and issuing commands by the dispatcher will be slower and more prone to errors.

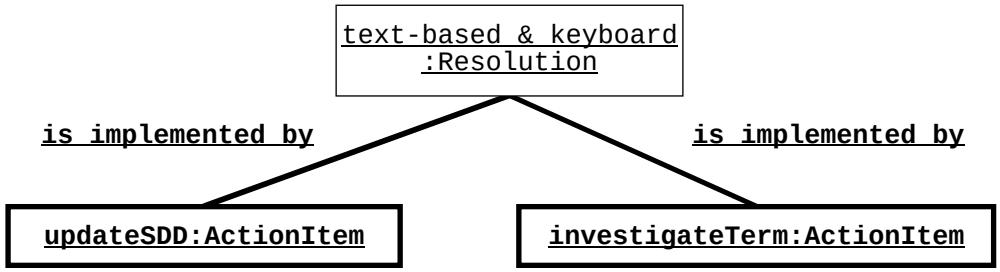
FIGURE 9-7. An example of closed issue. (UML object diagram)

Adding a resolution to an issue model effectively concludes the discussion of the corresponding issue. As development is iterative, it is sometimes necessary to reopen an issue and re-evaluate competing alternatives. At the end of development, however, most issues should be closed or listed as known problems in the documentation.

9.3.6. Implementing resolutions: action items

A resolution is implemented in terms of one or more ***action items***. An action item is a task assigned to a person with a completion date. Action items are not part of the rationale per say, but rather, they are part of the task model (see Chapter 4, *Project Management*). Action items are described here because they are tightly integrated into the issue model.

We represent an action item in UML with an instance of the **`ActionItem`** class. The **`ActionItem`** class has a **`subject`**, **`description`**, **`owner`**, **`deadline`**, and **`status`** attributes. The **`owner`** is the person responsible for the completion of the **`ActionItem`**. The **`status`** of an **`ActionItem`** can be **`todo`**, **`notDoable`**, **`inProgress`**, or **`done`**. A Resolution is associated with the ActionItems with an `is implemented by` link. Figure 9-8 represents the **`ActionItems`** generated after the resolution of Figure 9-7.



`updateSDD:ActionItem`	For Alice. Update the SDD to reflect the **`text-based&keyboard`** resolution.
`investigateTerm:ActionItem`	For Dave. Investigate different terminal emulation and their advantages for displaying **`TrackSections`**.

FIGURE 9-8. An example of implementation of a resolution. (UML object diagram)

The issue notation we described until now and its integration with the task model is the modeling notation we use for describing rationale. Other issue models have been proposed in the literature for representing rationale. Next, let us survey briefly these other models.

9.3.7. Examples of issue-based models and systems

The capture of rationale was originally proposed by Kunz and Rittel. Ever since, many different models have been designed and evaluated in the context of software engineering and other engineering disciplines. Here, we briefly compare three of them, IBIS (Issue-Based

Information System, [Kunz & Rittel, 1970]), DRL (Decision Representation Language, [Lee, 1990]), and QOC (Questions, Options, and Criteria, [MacLean et. al, 1991]).

Issue-Based Information System (IBIS)

IBIS includes an issue model and a design method for addressing ill-structured, or wicked problems (as opposed to tame problems). A ***wicked*** problem is defined as a problem which cannot be solved algorithmically, but rather, has to be resolved through discussion and debate.

The IBIS issue model (Figure 9-9) has three nodes (**`Issues`**, **`Positions`**, and **`Arguments`**) related by eight kinds of links (**`supports`**, **`objects-to`**, **`replaces`**, **`responds-to`**, **`generalizes`**, **`questions`**, and **`suggests`**). Each **`Issue`** describes a design problem under consideration. Developers propose solutions to the problem by creating **`Position`** (similar to the **`Proposal`** nodes we described in Section 9.3.3). While alternatives are being generated, developers argue about their value with **`Argument`** nodes. **`Arguments`** can either support a **`Position`** or object to a **`Position`**. Note that the same node can apply to multiple positions. The IBIS model did not originally include **`Criteria`** or **`Resolutions`**.

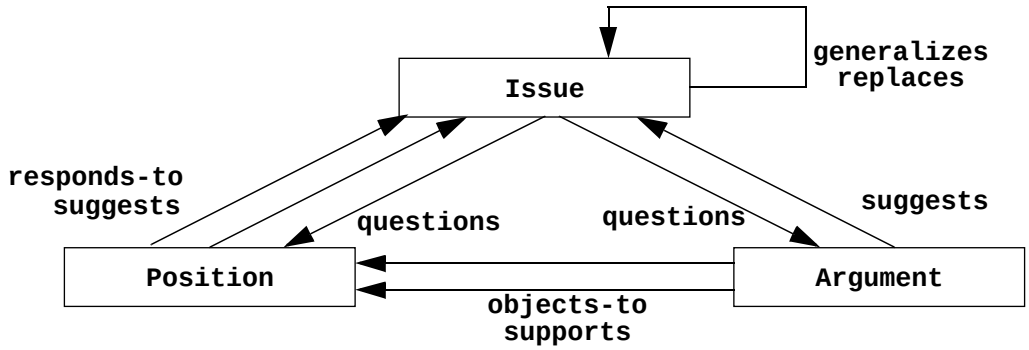


FIGURE 9-9. The IBIS model (UML class diagram, navigation added for clarity).

IBIS was supported by a hypertext tool (gIBIS, [Conklin & Burgess-Yakemovic, 1991]) and used for capturing rationale during face-to-face meetings. It provided the basis for most of the subsequent issue models, including DRL and QOC which we discuss next.

Decision Representation Language (DRL)

DRL (Decision Representation Language [Lee, 1990]) aims at capturing the ***decision rationale*** of a design. A decision rationale is defined by Lee as the representation of the qualitative elements of decision making, including the alternatives being considered, their evaluation, the arguments that led to these evaluations and the criteria used in these evaluations. DRL is supported by SYBIL, a tool that enables the user to track dependencies among elements of the rationale when revising evaluations. DRL elaborates on the original IBIS model by adding nodes to capture **`Design Goals`** and **`Procedures`**. DRL views the construction of the rationale as a comparable task as the design of the artifact itself. DRL is summarized in Figure 9-10. The main drawback of DRL is its complexity (9 types of nodes and 15 types of links) and the effort spent in structuring the captured rationale.

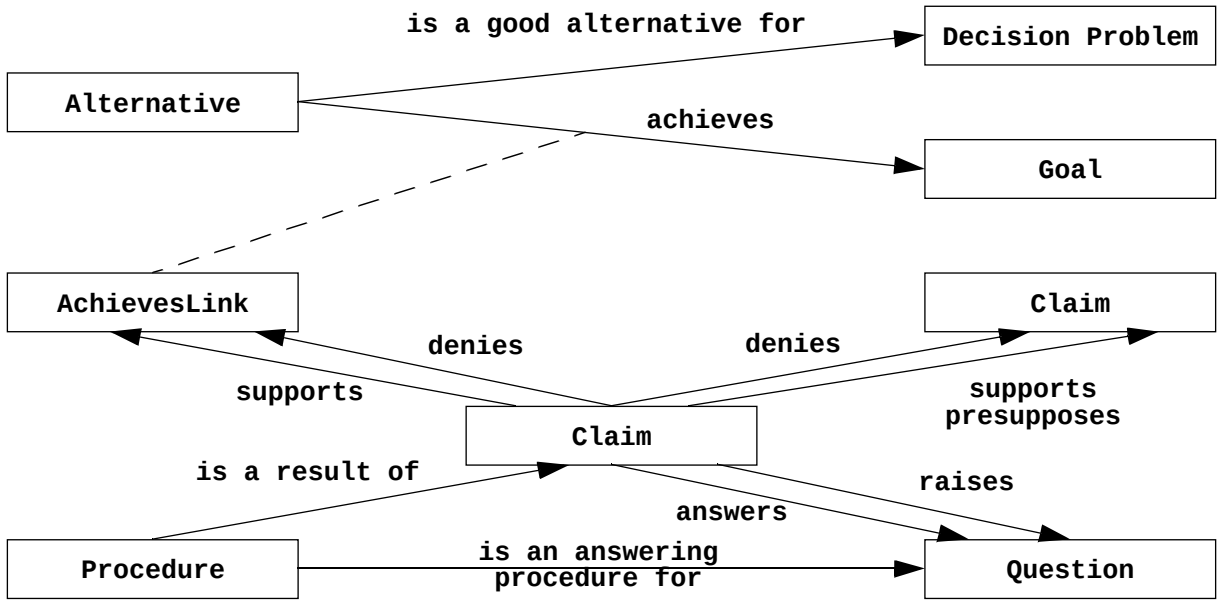


FIGURE 9-10. Decision Representation Language (UML class diagram, navigation added for clarity).

Questions, Options, and Criteria (QOC)

QOC (Questions, Options, and Criteria) is another elaboration of IBIS. **`Questions`** represent design problems to be solved (**`Issues`** in the issue model we presented). **`Options`** are

possible answers to **Questions** (**Proposals** in our model). **Options** can trigger other **Consequent Questions. Options** are assessed negatively and positively against **Criteria** which are relative measures of goodness defined by the developers. Also, **Arguments** can support or challenge any **Question**, **Option**, **Criteria**, or relationship among those. **Arguments** may also support and challenge **Arguments**. Figure 9-11 depicts the QOC model.

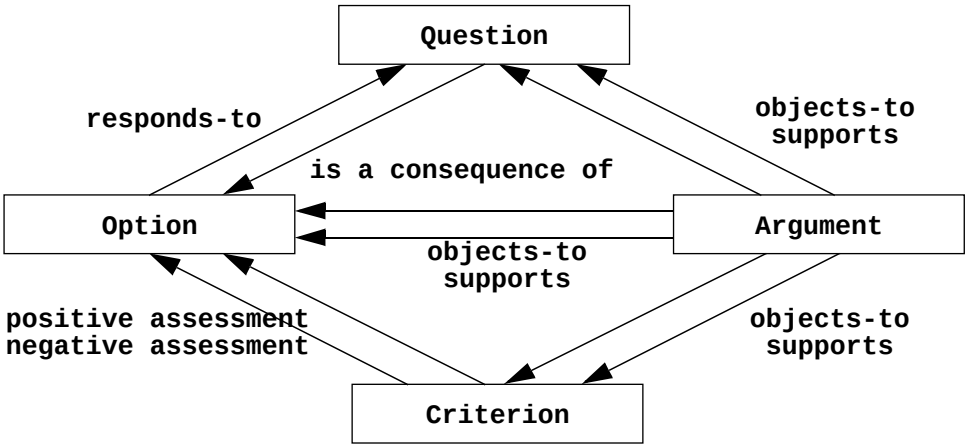


FIGURE 9-11. Questions, Options, Criteria model (UML class diagram, navigation added for clarity).

DSA and IBIS differ at the process level. IBIS's aim, on the one hand, has been to capture design argumentation as it occurs (e.g., gIBIS was used for capturing design meetings). QOC structures, on the other hand, are constructed as an act of reflection on the current state of the design. This conceptual separation of the construction and argumentation phases of the design process emphasizes the systematic elaboration and structuring of rationale, as opposed to capturing it as a side effect of deliberation. Rationale, from QOC's perspective, is a description of the design space explored by the developers. From IBIS's perspective, rationale is a historical record of the analysis leading to a specific design. In practice, both approaches can be applied to capture sufficient rationale. We describe the activities related to capturing and maintaining rationale next.

9.4. From issues to decisions

Maintaining rationale helps developers deal with change. By capturing the justification of decisions, they can more easily revisit important decisions when user requirements or the target environment changes. For rationale models to be useful, however, they need to be captured, structured, and easily accessible. In this section we describe these activities, including:

- capturing rationale during design meetings (Section 9.4.2),
- revising rationale models with subsequent clarifications (Section 9.4.3),
- capturing additional rationale during revisions (Section 9.4.4), and
- reconstructing rationale that was not captured (Section 9.4.5).

The most critical ratinonale information is generated during system design: decisions during system design can impact every subsystem and their revision is costly, especially when done late in the design process. Moreover, the rationale behind subsystem decomposition is usually complex, as it spans many different issues such as hardware allocation, persistent storage, access control, global control flow, and boundary conditions.

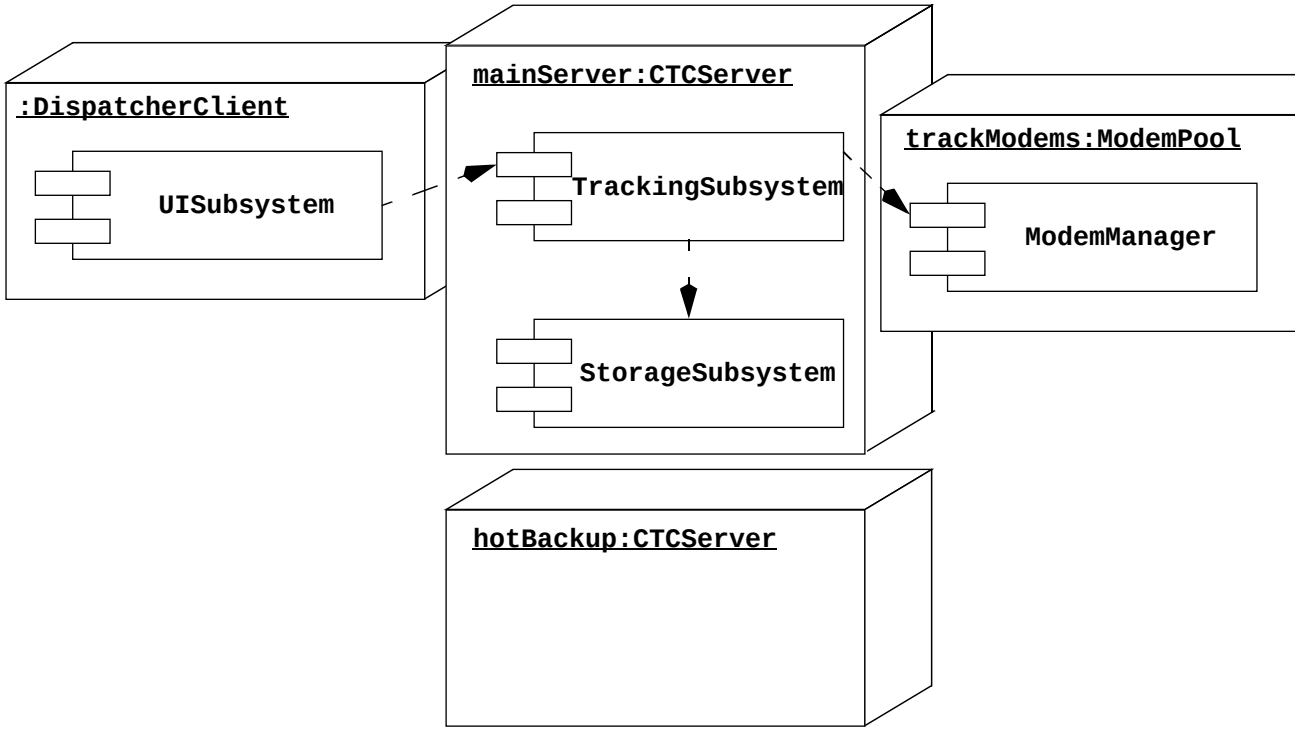
For these reasons, we focus on system design in this chapter. Note, however, that maintaining rationale can be done similarly applied throughout the development, from requirements elicitation to field testing. We illustrate rationale activities with issues from the system design of CTC, a central traffic control system for freight trains. We describe the current system design model of CTC next.

9.4.1. CTC system design

Consider the CTC system we described in Section 9.3.1. We are in the process of re-engineering a legacy system, replacing a mainframe computer with a network of workstations. We are also enhancing the system, such as adding access control and more focus on security and usability. We are in the middle of system design. So far, we identified from the nonfunctional requirements several design goals (ordered by descending priority):

- *Availability:* the system should be crash less than once per month and recover completely from a crash within 10 minutes.
- *Security:* no entity outside the control room should be able to access the state of the controlled tracks or manipulate any of their devices.
- *Usability:* once trained, a dispatcher should input no more than two erroneous command per day.

We have allocated a client node per dispatcher. Two redundant server nodes maintain the global state of the system (see Figure 9-12). The servers are also responsible for persistent storage. Data is stored in flat files that can be copied off-line and imported into a relational database for off-line processing by other systems. Communication with devices on the tracks is done via modems managed by a dedicated machine. A middleware supports two types of communication among subsystems: method invocation, for handling requests, and notification of state changes, for informing subsystems of state changes. Each subsystem subscribes to the events it is interested in. Presently, we must address access control issues and define the mechanisms that prevent dispatchers from manipulating **`TrackSections`** of other dispatchers. We describe in the following sections how the access control issue is debated and resolved while capturing its rationale.



`DispatcherClient` Each dispatcher is assigned a **`DispatcherClient`** node running the user interface to the system.

FIGURE 9-12. Subsystem decomposition for CTC (UML deployment diagram). The state of the system is maintained by a `mainServer`. A `hotBackup` of the `mainServer` stands by in case the `mainServer` fails. The `mainServer` sends commands and receives state transitions from the tracks via the **`ModemPool`**.

CTCServer	A **CTCServer** is responsible for maintaining the state of the system. It transmits dispatcher commands to the field devices and receives state information from the field via the **ModemPool** node. A **CTCServer** is also responsible for storing persistent state (e.g., device addresses, device names, dispatcher assignments, train schedules). Two **CTCServers**, a main server and a hot backup are used to increase availability.
ModemPool	The **ModemPool** manages the modems used to communicate with the field devices.
ModemManager	The **ModemManager** is responsible for connecting to field devices and transmitting field commands.
StorageSubsystem	The **StorageSubsystem** is responsible for maintaining persistent state.
TrackingSubsystem	The **TrackingSubsystem** is responsible for maintaining track state, as notices of state changes are received from the field, and for issuing device commands via the **ModemManager** based on user level commands received from the **UISubsystem**.
UISubsystem	The **UISubsystem** is responsible for receiving commands and displaying track state to the dispatcher. The **UISubsystem** controls the validity of the dispatcher's commands before forwarding them to the **CTCServer**.

FIGURE 9-12. Subsystem decomposition for CTC (UML deployment diagram). The state of the system is maintained by a mainServer. A hotBackup of the mainServer stands by in case the mainServer fails. The mainServer sends commands and receives state transitions from the tracks via the **ModemPool**.

9.4.2. Capturing rationale in meetings

Meetings enables developers to present, negotiate, and resolve issues face-to-face. The physical presence of the respective developers involved in the discussion is important, adding the benefits of nonverbal communication: it allows people to assess the relative positions of each other and the trade-offs they are willing to make. Conversely, negotiating and making decisions via email, for example, is difficult as misunderstandings can easily occur. Face-to-face meetings, then, are a natural starting point for capturing rationale.

We described procedures for organizing and capturing meetings with minutes and agendas in Chapter 5, *Project Communication*,. An agenda, posted in advance of the meeting, describes the status and points to be discussed. The meeting is recorded in minutes that are made available shortly after the meeting. Using the issue modeling concepts we described in Section 9.3, we write an agenda in terms of *issues* that we need to discuss and resolve. We

state the objective of the meeting as coming to a resolution on these issues and any related subissue that is raised in the discussion. We structure the meeting minutes in terms of *proposals* that we explore during the meeting, *criteria* that we agree on, and *arguments* we use to support or oppose proposals. We capture decisions as *resolutions* and *action items* that implement resolutions. During the meeting we review status in terms of the action items that we produced in the previous meetings.

For example, consider the access control issue of the CTC system. We need to organize a meeting of the architecture team, including the developers responsible for the **`UISubsystem`**, the **`TrackingSubsystem`**, and the **`NotificationService`**. Alice, the facilitator for the architecture team, posts the agenda depicted in Figure 9-13.

AGENDA: Integration of access control and notification

When and Where		**Role**
Date:	9/13/1998	Primary Facilitator: alice
Start:	4:30pm	Timekeeper: dave
End:	5:30pm	Minute Taker: ed
Building:	Train Hall	
Room:	3420	

1. Purpose

The first revisions of the hardware/software mapping and the persistent storage design have been completed. The access control model needs to be defined and its integration with the current subsystems, in particular, **`NotificationService`** and **`TrackingSubsystem`**, needs to be defined.

2. Desired outcome

Resolve issues about the integration of access control with notification

3. Information sharing [Allocated time: 15 minutes]

`AI[1]:` Dave: Investigate the access control model provided by the middleware.

4. Discussion [Allocated time: 35 minutes]

`I[1]:` Can a dispatcher see other dispatchers' **`TrackSections`**?
`I[2]:` Can a dispatcher modify another dispatchers' **`TrackSections`**?
`I[3]:` How should access control be integrated with **`TrackSections`** and **`NotificationService`**?

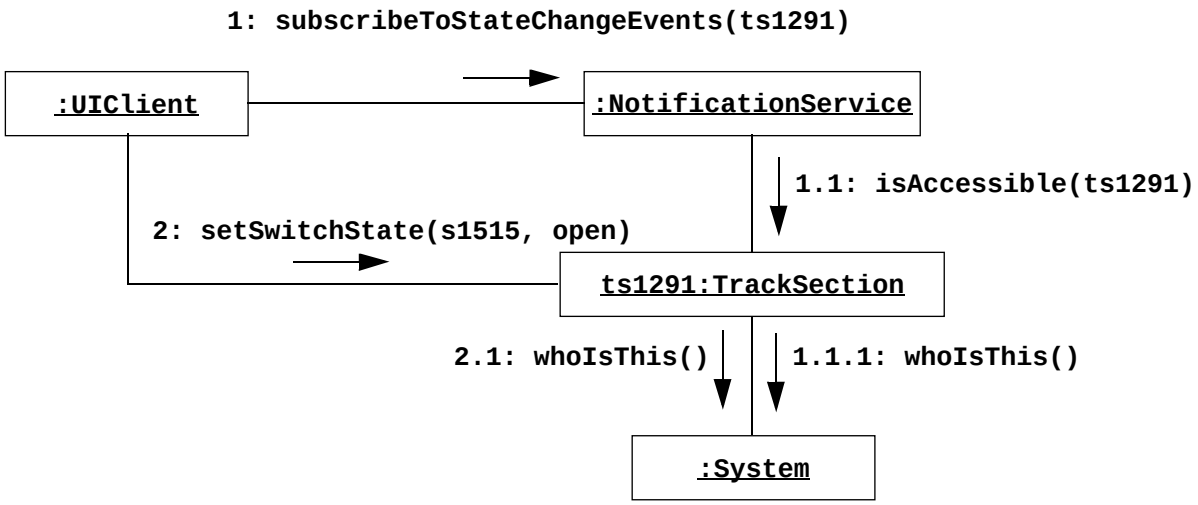
5. Wrap up [Allocated time: 5 minutes]

Review and assign new action items.
Meeting critique.

FIGURE 9-13. Agenda for the access control discussion of CTC.

During the meeting, we review the action item ***(AI[1]:*** *Investigate access control model by middleware)* generated in the previous architecture meeting. The middleware provides basic blocks for authentication and encryption, but does not introduce any other constraints on the access model. Issues **I[1]** and **I[2]** are resolved quickly with domain knowledge: a dispatcher can see all **TrackSections** but can only manipulate the devices of her **TrackSection**. Issue **I[3]**, however, *(How should access control be integrated with* ***TrackSections*** *and* ***NotificationService****?)* is more difficult and sparks a debate.

Dave, the developer responsible for the **NotificationService**, proposes to integrate the access control with the **TrackSection** (see Figure 9-14). The **TrackSection** would maintain an access list of the **Dispatchers** who can examine or modify the given **TrackSection**. Events would also be organized by **TrackSections**. To be notified about events in a **TrackSection**, a subsystem would need to subscribe to a **TrackSection** via the **NotificationService**. The **NotificationService** would then check with the given **TrackSection** if the current dispatcher had at least read access.



NotificationService	The **NotificationService** broadcasts changes of state of a **TrackSection**. To receive notices from the **NotificationService**, a subsystem needs to subscribe to a **TrackSection**. Only subsystems who have access to a given **TrackSection** can subscribe to the events generated by a **TrackSection**. The access is determined by invoking the **isAccessible()** operation on the **TrackSection**.

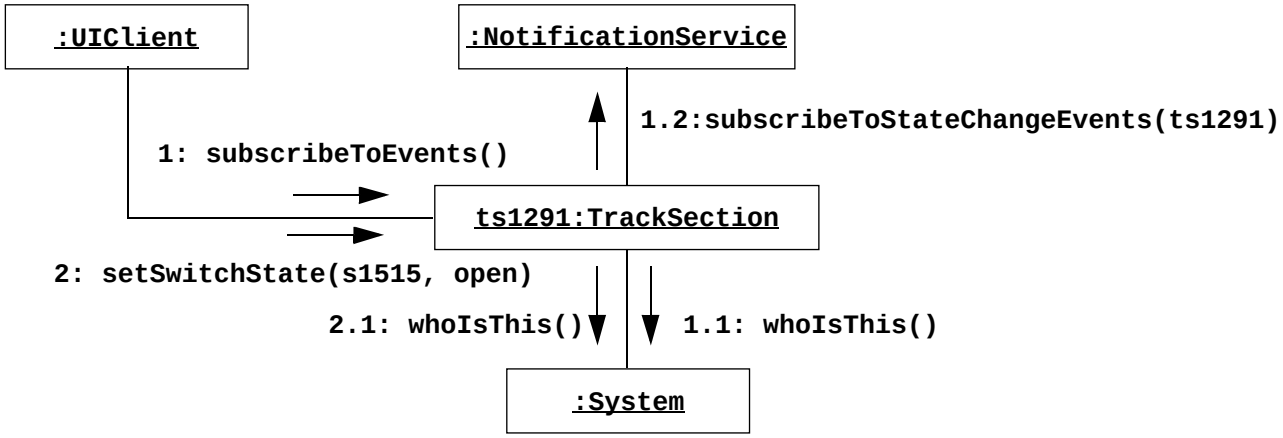
FIGURE 9-14. Proposal **P[1]**: The access is controlled by the **TrackSection** object with an access list. The **NotificationService** queries the **TrackSection** to determine if a subsystem can receive notices about a given **TrackSection**. (UML collaboration diagram.)

System	The **System** is responsible for tracking securely who is the current dispatcher based on information provided by the **UIClient**. The **TrackSection** examines the credentials of the current dispatcher with the **whoIsThis()** operation.
TrackSection	A **TrackSection** consists of a set of contiguous **TrackCircuits** and their associated **Devices**. Access is controlled at the **TrackSection** level with an access list.
UIClient	The **UIClient** is responsible for displaying **TrackSections** and inputting commands for changing **TrackSection** state.

FIGURE 9-14. Proposal **P[1]**: The access is controlled by the **TrackSection** object with an access list. The **NotificationService** queries the **TrackSection** to determine if a subsystem can receive notices about a given **TrackSection**. (UML collaboration diagram.)

Alice, the developer responsible for the **TrackSubsystem** which includes the **TrackSection** class, proposes to reverse the dependency between the **TrackSection** and the **NotificationService** (see Figure 9-15). In this proposal, the **UIClient** would interact only with the **TrackSection** class, including for subscribing to events. The **UIClient** would invoke the **subscribeToEvents()** method on the **TrackSection**, which would perform the access control checks and then invoke the **subscribeToStateChangeEvents()** on the **NotificationService**. The **UIClient** would then not have direct access to the **NotificationService**. This has the advantage to centralize all the protected operations

into one class and centralize the access control checks. Moreover, the **TrackSection** would then also be able to unsubscribe **UIClients** when the access list is modified.

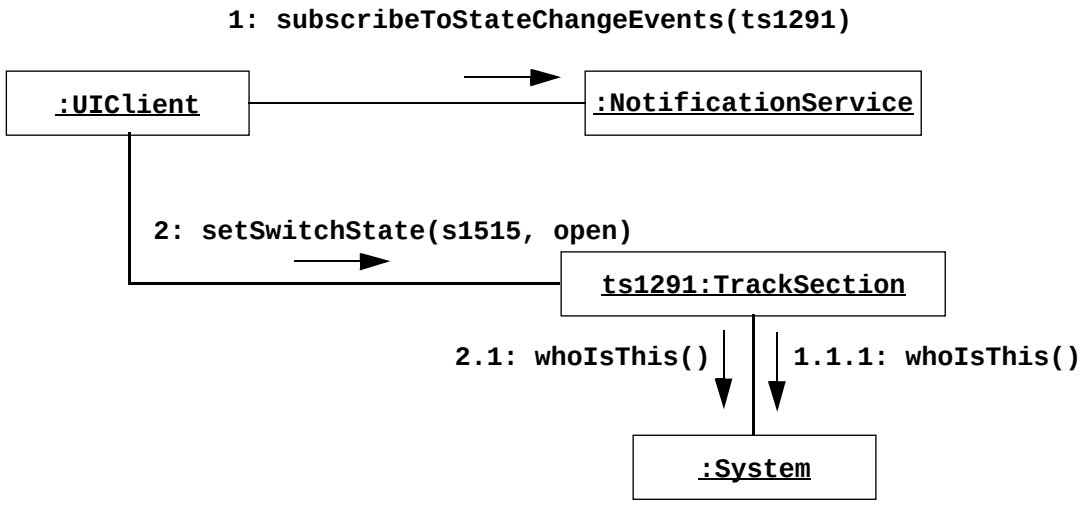


NotificationService	The **NotificationService** broadcasts changes of state of a **TrackSection**. To receive notices from the **NotificationService**, a subsystem needs to subscribe to a **TrackSection**. *The **NotificationService** is only accessible to the **TrackSection** class which controls access.*
TrackSection	A **TrackSection** consists of a set of contiguous **TrackCircuits** and their associated **Devices**. Access is controlled at the **TrackSection** level. *To receive notices of state changes, a subsystem needs to invoke the **subscribeToEvents()** operation of the **TrackSection** class. The **TrackSection** checks access before invoking the **subscribeToTrackSectionEvents()** method on **NotificaitonService**.*

FIGURE 9-15. Proposal **P[2]**: The **UIClient** subscribes to track section events via the **subscribeToEvents()** operation on the **TrackSection**. The **TrackSection** checks access and then invokes the **subscribeToTrackSectionEvents()** operation on the **NotificationService**. The **NotificationService** is not accessible to the **UIClient** class. (UML collaboration diagram, differences from Figure 9-14 highlighted in *italics*.)

Ed notes that every dispatcher is allowed to see other dispatcher's **TrackSections**, only modification of state needs to be controlled. Assuming that all modifications are done via method invocation and that the **NotificationService** is only used for broadcasting

changes, the **NotificationService** need not be integrated with access control. In this case, a refinement of Dave's initial proposal could be used.



NotificationService — The **NotificationService** broadcasts changes of state of a **TrackSection**. To receive notices from the **NotificationService**, a subsystem needs to subscribe to a **TrackSection**. ~~*Only subsystems who have access to a given* ***TrackSection*** *can subscribe to the events generated by a* ***TrackSection***. *The access is determine by invoking the* ***isAccessible()*** *operation on the* ***TrackSection***.~~

FIGURE 9-16. Proposal **P[3]**: The access to operations that modify **TrackSections** is controlled by the **TrackSection** object with an access list. The **NotificationService** need not be part of the access control since every dispatcher can see changes of state. (UML collaboration diagram, differences from Figure 9-14 highlighted in *italics*.)

The architecture team decides to use Ed's proposal based on its simplicity. Ed produces the chronological meeting minutes depicted in Figure 9-17.

CHRONOLOGICAL MINUTES: Integration of access control and notification

When and Where		**Role**
Date:	9/13/1998	Primary Facilitator: alice
Start:	4:30pm	Timekeeper: dave
End:	6:00pm	Minute Taker: ed
Building:	Train Hall	
Room:	3420	

1. Purpose

The first revisions of the hardware/software mapping and the persistent storage design have been completed. The access control model needs to be defined and its integration with the current subsystems, in particular, **`NotificationService`** and **`TrackingSubsystem`**, needs to be defined.

2. Desired outcome

Resolve issues about the integration of access control with notification

3. Information sharing

`AI[1]:` Dave: Investigate the access control model provided by the middleware.
Status: The middleware supports strong authentication and encryption. It does not introduce any constraints on the access model. Any access policy can be implemented on the server side.

4. Discussion

`I[1]:` Can a dispatcher see other dispatchers' **`TrackSections`**?
`Zoe`: Yes.
`Ed:` In CTC specification

`I[2]:` Can a dispatcher modify another dispatchers' **`TrackSections`**?
`Zoe:` No. Only the dispatcher assigned to the **`TrackSection`** can manipulate the devices of the section. Note that the dispatcher can be re-assigned dynamically.
`Ed:` Also in CTC specification.

`I[3]:` How should access control be integrated with **`TrackSections`** and **`NotificationService`**?

`Dave:` The **`TrackSection`** maintains an access list. The notification service asks the **`TrackSection`** about how has access.

`Alice:` We should probably reverse the dependency between **`TrackSection`** and **`NotificationService`**. Instead, the **`UIClient`** requests subscriptions from the **`TrackSection`** which checks for access and then call the **`NotificationService`**. This way, all protected methods are in one place.

`Dave:` This way the **`TrackSection`** can also more easily unsubscribe dispatchers when their access is revoked.

Ed: Hey, no need for access control in **NotificationService**: dispatchers can see all **TrackSections**. As long as the **NotificationService** is used not used for changing the **TrackSection** state, no need to restrict subscriptions.

Alice: But thinking about the access control on notification would be more general.

Ed: But more complex. Let's just separate access control and notification at this point and revisit
the issue if the requirements change.

Alice: Ok. I'll take care of revising the **TrackingSubsystem** API.

5. Wrap up

AI[2]: Alice: design access control for the **TrackingSubsystem** based on authentication and encryption provided by the middleware.

FIGURE 9-17. Chronological minutes for the access control discussion of CTC.

Ed produces the minutes of Figure 9-17 by inserting in the agenda the discussion relevant to the different issues. The discussion, however, is recorded as a chronological list of statements made by the participants. Most of these statements mix the presentation of an alternative with the argumentation against another alternative. In order to clarify the minutes, Ed restructures the minutes after the meeting with issue models (Figure 9-18)

STRUCTURED MINUTES: Integration of access control and notification

When and Where		**Role**
Date:	9/13/1998	Primary Facilitator: alice
Start:	4:30pm	Timekeeper: dave
End:	6:00pm	Minute Taker: ed
Building:	Train Hall	
Room:	3420	

1. Purpose

The first revisions of the hardware/software mapping and the persistent storage design have been completed. The access control model needs to be defined and its integration with the current subsystems, in particular, **NotificationService** and **TrackingSubsystem**, needs to be defined.

2. Desired outcome

Resolve issues about the integration of access control with notification

3. Information sharing

AI[1]: Dave: Investigate the access control model provided by the middleware.
Status: The middleware supports strong authentication and encryption. It does not introduce any constraints on the access model. Any access policy can be implemented on the server side.

4. Discussion

I[1]: Can a dispatcher see other dispatchers' **TrackSections**?
R[1]: Yes (from CTC specification and confirmed by Zoe, a test user).

I[2]: Can a dispatcher modify another dispatchers' **TrackSections**?
R[2]: No. Only the dispatcher assigned to the **TrackSection** can manipulate the devices of the section. Note that the dispatcher can be re-assigned dynamically (from CTC specification and confirmed by Zoe).

I[3]: How should access control be integrated with **TrackSections** and **NotificationService**?
P[3.1]: **TrackSections** maintain an access list of who can examine or modify the state of the **TrackSection**. To subscribe to events, a subsystem sends a request to the **NotificationService** which in turns sends a request to the corresponding **TrackSection** to check access.
P[3.2]: **TrackSections** host all protected operations. The **UIClient** requests subscription to **TrackSection** events by sending a request to the TrackSection, which checks access and sends a request to the **NotificationService**.
A[3.1] for **P[3.2]**: Access control and protected operations are centralized into one class.
P[3.3]: There is no need to restrict the access to the event subscription. The **UIClient** requests subscriptions directly from the **NotificationService**. The **NotificationService** need not check access.
A[3.2] for **P[3.3]** Dispatchers can examine the state of any **TrackSections** (see **R[1]**).
A[3.3] for **P[3.3]**: Simplicity.
R[3]: P[3.3]. See action item **AI[2]**.

5. Wrap up

AI[2]: Alice: design access control for the **TrackingSubsystem** based on authentication and encryption provided by the middleware and on resolution **R[3]** discussed in these minutes.

FIGURE 9-18. Structured minutes for the access control discussion of CTC.

The important results of the access control meeting are:

- dispatchers can see all **TrackSections** but modify only the ones they are assigned to,
- an access list associated with **TrackSections** is used for access control,
- **NotificationService** is not integrated with access control, since state changes can be seen by any dispatchers.

By focusing on the issue-model, we have also captured that:

- integrating the **`NotificationService`** with access control was investigated,
- centralizing all protected methods into the **`TrackSection`** class was an accepted principle.

The last two pieces of information are rationale information and would usually be considered unimportant. However, this is the type of information that is captured by the minute taker and structured for facilitating future changes.

9.4.3. Capturing rationale asynchronously

Meeting discussions rely on context information. When the meeting starts, most participants already have a substantial amount of information about the system, its intended purpose and its design. The facilitator of the meeting usually focuses on a small set of issues which need to be resolved. For example, in the meeting we presented in the previous section, all participants knew the purpose and functionality of the CTC system, its design goals, and current subsystem decomposition. The minutes of this meeting only records the issues under discussion and, therefore, does not contain much or any of the background information. Unfortunately, this information is lost over time and meeting minutes become obsolete quickly.

We can use issue modeling to address this problem. In Chapter 5, *Project Communication*, we described the use of groupware, such as newsgroups or Lotus Notes, for supporting asynchronous communication. By integrating the preparation and recording of the meeting with the asynchronous communication, we can capture more contextual information.

In the CTC example, assume Mary, the developer responsible for the **`UISubsystem`**, was not able to attend the access control meeting. She reads the agenda and the meeting minutes which were posted on the newsgroup dedicated to the architecture team. Although she understands the outcome of the meeting, the discussion about the **`NotificationService`** requires clarification: argument **`A[3.2]`** for proposal **`P[3.3]`** claims that, since dispatchers can see every **`TrackSection`**, all events can be visible and, hence, there is no need to control the access to the events.This implies that the **`NotificationService`** is used only for notifying other subsystems of state changes. In other words, the **`TrackSection`** does not change its state as a consequence of events generated by other subsystems. Mary wants to confirm that this assumption is correct and, consequently, posts an issue on the newsgroup

(Figure 9-19). She also proposes to disallow the **`TrackingService`** from subscribing to any events, in order to ensure proper access control.

```
Newsgroup:  ctc.architecture.discuss
Subject:                                                    Date:
I[1]: Can a dispatcher see other dispatchers' TrackSections?   9/14/1998
I[2]: Can a dispatcher modify another dispatchers'
   TrackSections?                                           9/14/1998
I[3]: How should access control be implemented?             9/14/1998
   P[3.1]: TrackSection has access list                     9/14/1998
   P[3.2]: TrackSection has subscription operations         9/14/1998
      +A[3.1]: Extensibility.                               9/14/1998
      +A[3.2]: Centralize all protected operations.         9/14/1998
   P[3.3]: NotificationService is not part of access        9/14/1998
      +A[3.3]: Dispatchers can see all TrackSections        9/14/1998
      +A[3.4]: Simplicity.                                  9/14/1998
---
From: Mary
Newsgroups: ctc.architecture.discuss
Subject: Consequent Issue: Should notification not be used for requests?
Date: Thu, 15 Sep 1998 13:12:48 -0400

I[4] responding to A[3.3]: for access lists against capabilities
> Dispatchers can see all TrackSections and, thus, should be able
> to see all events.

This assumes that the TrackSection does not rely on events to change its
state and that events are only used for informing other subsystems of
state changes. For the purpose of robustness, should we disallow the
TrackingService to subscribe to any events?
```

FIGURE 9-19. Example of a consequent issue posted asynchronously (newsgroup post). Mary, a developer who did not attend the meeting, requests clarification. This leads to the post of an additional issue and the capture more of the rationale.

Follow up on meeting minutes enable developers to capture more of the context surrounding the design. As a consequence, more rationale, and clearer information is captured. By using the same issue model for both meetings and on-line discussions allows us to integrate all rationale information. Although this can be done with minimal technology, such as newsgroups, the representation of the issue-model, the meeting agendas and minutes, and related messages can be integrated into a groupware tool, such as a

custom Lotus Notes database or a multi-user issue-base hosted on a web site (see example in Figure 9-20).

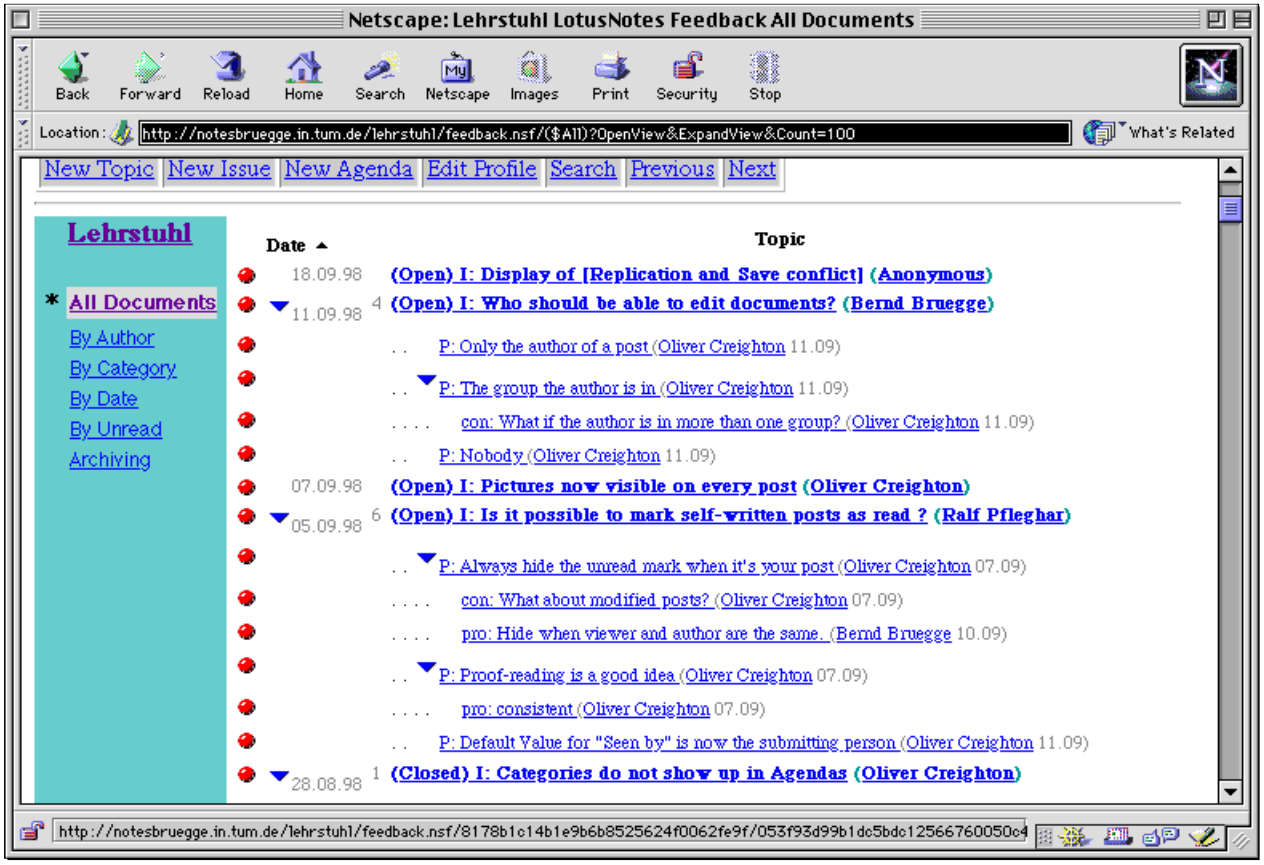


FIGURE 9-20. An example of web-based issue database (Lotus Notes / Domino). Developers posts issues, proposals, arguments, and resolutions with web forms and browse the issue model. (xxx place holder xxx)

Once we institute procedures for organizing and recording rationale in meetings, and expanding it with groupware, we are able to capture a great deal of rationale. The next challenge is to keep this information up-to-date as changes occur.

9.4.4. Capturing rationale when discussing change

Rationale models help us deal with change. Unfortunately, rationale is itself subject to change when we revise decisions and needs to be updated. When we design a solution in

response to a requirements change, for example, we look at past rationale to assess which decisions need to be revised, and design a change. Not only do we need to capture the rationale for the change and its solution, we also need to relate it with past rationale.

For example, in the CTC system, assume the requirements on the access control changed. Before, dispatchers were allowed to see all **`TrackSections`**. The client informed us that, unlike previously specified, dispatchers should be able to only see the neighboring **`TrackSections`**. In response to this change, we need to modify the design of the access control and organize a meeting with the architecture team. In particular, we need to search past rationale associated with access control Alice, the primary facilitator of the architecture team, posts the agenda depicted in Figure 9-21.

AGENDA: Revision of access control, dispatchers can only access neighboring tracks.

When and Where		**Role**
Date:	10/13/1998	Primary Facilitator: alice
Start:	4:30pm	Timekeeper: dave
End:	5:30pm	Minute Taker: ed
Building:	Signal Hall	
Room:	2300	

1. Purpose

The client requested that dispatchers be able to access neighboring **`TrackSections`**.

2. Desired outcome

Resolve access control issues related to this change of requirement.

3. Information sharing [Allocated time: 15 minutes]

`AI[1]:` Dave: Recover rationale for access control.

4. Discussion [Allocated time: 35 minutes]

`I[1]:` How should access control be revised based on the neighboring track requirement?

5. Wrap up [Allocated Time: 5 minutes]

Review and assign new action items.
Meeting critique.

FIGURE 9-21. Agenda for the access control revision of CTC.

During the meeting, Dave presents the rationale discussed in previous meetings and on the architecture newsgroup. The architecture team notices that the assumption that all subsystems can see events is not valid anymore: the dispatcher should be allowed to see only the events related to neighboring **`TrackSections`**. Proposal P[2, 9/14/98] (see

Figure 9-15) seems to be the better solution under the new requirements, as all protected operations could be centralized in the **`TrackSection`** class. Unfortunately, the implementation has already progressed and the developers want to minimize changes to the code. Instead, Alice proposes to select proposal P[1, 9/14/98] (see Figure 9-14): the current **`UIClient`** stays unchanged as the interfaces to the **`TrackSection`** and **`NotificationService`** classes need not change. Only the **`NotificationService`** needs to change such that it sends requests to the **`TrackSection`** to check the access of the current dispatcher. To revoke dispatcher privileges when an access list is changed, the **`TrackSection`** sends a request to the **`NotificationService`** to unsubscribe dispatchers. This introduces a circular dependency between **`TrackSection`** and **`NotificationService`** but minimizes modifications to existing code.

This solution is selected by the architecture team. Ed produces the structured minutes depicted in Figure 9-22 (chronological minutes not displayed for brevity).

STRUCTURED MINUTES: Revision of access control, dispatchers can only access neighboring tracks.

When and Where		**Role**
Date:	10/13/1998	Primary Facilitator: alice
Start:	4:30pm	Timekeeper: dave
End:	5:30pm	Minute Taker: ed
Building:	Signal Hall	
Room:	2300	

1. Purpose

The client requested that dispatchers be able to access neighboring **`TrackSections`**.

2. Desired outcome

Resolve access control issues related to this change of requirement.

3. Information sharing

`AI[1]:` Dave: Recover rationale for access control.
Result: issues **`I[1, 9/13/98]`** and **`I[2, 9/15/98]`** recovered:

`I[1, 9/13/1998]:` *How should access control be integrated with* ***`TrackSections`*** *and* ***`NotificationService`****? (Minutes from 9/14)*

> ***`P[3.1]: TrackSections`*** *maintain an access list of who can examine or modify the state of the* ***`TrackSection`****. To subscribe to events, a subsystem sends a request to the* ***`NotificationService`*** *which in turns sends a request to the corresponding* ***`TrackSection`*** *to check access.*

P[3.2]: TrackSections *host all protected operations. The* ***UIClient*** *requests subscription to* ***TrackSection*** *events by sending a request to the TrackSection, which checks access and sends a request to the* ***NotificationService****.*
A[3.1] *for* ***P[3.2]****: Extensibility.*
A[3.2] *for* ***P[3.2]****: Access control and protected operations are centralized into one class.*

P[3.3]: *There is no need to restrict the access to the event subscription. The* ***UIClient*** *requests subscriptions directly from the* ***NotificationService****. The* ***NotificationService*** *need not check access.*
A[3.3] *for* ***P[3.3]*** *Dispatchers can examine the state of any* ***TrackSections*** *(see* ***R[1]****).*
A[3.4] *for* ***P[3.3]****: Simplicity.*

R[3]: P[3.3]*. See action item* ***AI[2]****.*

I[2,9/15/1998]: *Should notification not be used for requests? (from Mary's news post 9/15)*
R[2]: *Notification should be used only for informing of state changes.* ***TrackSections*** *and, more generally,* ***TrackingSubsystem*** *should not change their state based on events.*

4. Discussion

I[1]: How should access control be revised based on the neighboring track requirement?
P[1.1]: Protected operations, including subscription, centralized in **TrackSection**, as in **P[3.2, 9/13/1998]**.
AI[1.1] against **P[1.1]:** This requires all subsystems subscribing to notification events to be modified, since the subscription operation is moved from the **NotificationService** to the **TrackSection**.
P[1.2]: **NotificationService** sends requests to **TrackSections** to check access. **TrackSection** sends request to **NotificationService** to unsubscribe dispatchers whose access has been revoked. **P[3.1, 9/13/1998]**
A[1.2] for **P[1.2]**: Minimal change to existing implementation.
A[1.3] against **P[1.2]**: Circular dependencies.
R[1]: **P[1.2]**, see **AI[2]** and **AI[3]**.

5. Wrap up

AI[2]: Alice: change the **TrackSection** to unsubscribe dispatchers when their rights are revoked.
AI[3]: Dave: modify **NotificationService** to check access with **TrackSection** when subscribing a new subsystem.

FIGURE 9-22. Structured minutes for the access control revision of CTC.

The minutes depicted in Figure 9-22 serve two purposes: to record the rationale for the new change and to relate it to past rationale. This is done by quoting the past rationale that was used to revisit the access control decision. Furthermore, these new minutes are posted on the architecture newsgroup and discussed by other developers who could not attend the meeting, thus completing the cycle of recording and clarifying rationale information. In the

case groupware is used, the new rationale can be related to the past rationale with a hyperlink, making it easier for developers to navigate to the related information.

Note that even when an issue base is used to maintain and track open issues, this information base can grow quickly into a large unstructured chaos. Moreover, some issues are not recorded as not all issues are discussed in meetings. Many issues are discussed and resolved informally in hallway conversions. It is necessary, therefore, to reconstruct the missing rationale of the system and integrate them with past rationale. We discuss this in the next section.

9.4.5. Reconstructing rationale

Reconstructing rationale is a different method for capturing the rationale of the system. Instead of capturing decisions and their justifications as they occur, rationale is systematically reconstructed from the system model, the communication record, and developers' memories. With this method, rationale is captured and structured more systematically. Fewer resources are invested during the early phases of the process, thus enabling developers to come faster to a solution. Also, separating the design activity from the rationale capture enables developers to step back and critique their design more objectively. Reconstructing rationale, however, focuses on the selected solution and fails to capture discarded alternatives and their discussion.

For example, assume we did not capture the rationale of the integration between notification and access control in CTC and that the only information we had was the system design model in Figure 9-23.

[...]
4. Access control
Access in CTC is controlled at the level of **`TrackSections`**: the **`Dispatcher`** who is assigned to a **`TrackSection`** can modify its state, that is, open and close signals and switches and modify other devices. Moreover, the **`Dispatcher`** can examine the state of neighboring **`TrackSections`** without modifying their state. This is necessary for the **`Dispatcher`** to observe the **`Trains`** that are about to enter the controlled **`TrackSection`**.
Access control is implemented with an access list maintained by the **`TrackSection`**. The access list contains the identity of the **`Dispatcher`** who can modify the **`TrackSection`** (i.e., writers) and the identify of the **`Dispatcher`** who can examine the state of the track section (i.e., readers). For the sake of generality, the access list is implemented such that it can include multiple readers and multiple writers.
The **`TrackSection`** checks the access list for every operation that modifies or queries the state of the **`TrackSection`**.
When subsystems subscribe to events, the **`NotificationService`** sends a request to the **`TrackSection`** to check access. The **`TrackSection`** sends a request to the **`NotificationService`** to unsubscribe dispatchers whose access is revoked.
The collaboration diagram of Figure 9-14 depicts this solution.
[...]

FIGURE 9-23. Excerpt from system design document, access control section.

We want to recover the rationale of the system design for review and documentation. We decide to organize each issue as a table with two columns, the left column for the proposals and the right column for their corresponding arguments. In Figure 9-24, we recover the rationale for the integration of access control with notification. We identify two possible solutions: **`P[1]`** in which the **`TrackSection`** class exports all operations whose access is controlled including subscription to notifications, and **`P[2]`** in which the **`NotificationService`** delegates the access control check to the **`TrackSection`**. We then enumerate the advantages and disadvantages of each solution in the right column and summarize the justification of the decision as a resolution at the bottom of the table.

`I[1]`: How should access control of TrackSections be integrated with notification?
Access in CTC is controlled at the level of **`TrackSections`**: the **`Dispatcher`** who is assigned to a **`TrackSection`** can modify its state, that is, open and close signals and switches and modify other devices. Moreover, the **`Dispatcher`** can examine the state of neighboring **`TrackSections`** without modifying their state. This is necessary for the **`Dispatcher`** to observe the **`Trains`** that are about to enter the controlled **`TrackSection`**.

FIGURE 9-24. Reconstructed rationale for the notification access control issue of CTC.

P[1]: TrackSection class controls all state modification, and notification subscription access. Access control is implemented as an access list in **TrackSection**. The **TrackSection** class checks access of the caller for every operation that examines or modifies state. In particular, the caller subscribes to notification events by invoking methods on **TrackSection**, which in turns forwards the request to the **NotificationService** if access is granted. This solution is illustrated in Figure 9-15.	**For:** • Central solution: all protected methods related to the **TrackSection** are in one place.
P[2]: TrackSection class controls state modification, NotificationService controls subscription. As **P[1]**, except that the caller requests subscriptions to events directly from the **NotificationService**. The **NotificationService** checks access with the **TrackSection** before granting the subscription. This solution is illustrated in Figure 9-14.	**For:** • Access independent interface: the interfaces of **NotificationService** and **TrackSection** are the same as if there was no access control (legacy argument). **Against:** • Circular dependency between **NotificationService** and **TrackSection**: The **TrackSection** invokes operations on the **NotificationService** to generate events, the **NotificationService** subscription methods invoke operations on the **TrackSection** to check access.

R[1]
P[2]. **P[1]** would have been a better solution, however, access control did not apply to notification. To minimize code and design rework, **P[2]** was selected.

FIGURE 9-24. Reconstructed rationale for the notification access control issue of CTC.

A reconstructed rationale, such as the one in Figure 9-24, costs less to capture than the activities we described previously. It is more difficult, however, to capture the discarded alternatives and the reasons of such choices, especially when decisions are revised over time. In Figure 9-24, the resolution states that we did not select the better proposal, and we were able to remember the reasons for this non optimal decision (i.e., that substantial code had be completed prior to this decision and we wanted to minimize code rework). Alternatively, reconstructing rationale is an effective tool for review, for identifying decisions that are inconsistent with the design goals of the project. Moreover, even if the reviewed decisions cannot be revised at a late stage in the project, this knowledge can benefit new developers assigned to the project or developers revising the system in later iterations.

The balance between rationale capture, maintenance, and reconstruction differs for each project and needs to be carefully managed. It is relatively frequent to see rationale capture efforts accumulate enormous amounts of information that are either useless or not easily accessible to developers who should benefit from such information. We focus on management issues next.

9.5. Managing rationale

In this section, we describe issues related to managing rationale activities. Recording justifications for design decisions is often seen as an intrusion from management into the work of developers, and thus, rationale techniques encounter resistance from developers and often degenerate into a bureaucratic process. Rationale techniques need to be carefully managed to be useful. In this section, we describe how to:

- write documents about rationale (Section 9.5.1),
- assign responsibilities for capturing and maintaining rationale models (Section 9.5.2),
- communicate about rationale models (Section 9.5.3),
- use issues to negotiate (Section 9.5.4), and
- resolve conflicts (Section 9.5.5).

As before, we continue focusing on the system design activity. Note, however, that these techniques can be applied uniformly throughout the development.

9.5.1. Documenting rationale

When rationale is explicitly captured and documented, tit is best described in documents that are separate from system model documents. For example, the rationale behind requirements analysis decisions is documented in the *Requirements Analysis Rationale Document* (RARD) which complements the *Requirements Analysis Document* (RAD). Similarly, the rationale behind system design decisions is documented in the *System Design Rationale Document* (SDRD), which complements the *System Design Document* (SDD). In this section, we focus on the SDRD, as system design is the activity which benefits most from capturing rationale. Figure 9-25 is an example of template for the SDRD.

The audience for the SDRD is the same as for the SDD. They are used by developers when revising decisions, by reviewers when reviewing the system, and by new developers when assigned to the project. The specific activities for which the rationale document is intended are described in the first section of the document. A document focusing on justifying the system for reviewers might only contain proposals and arguments relevant to the selected resolutions. A document capturing as much of the design context as possible might contain in addition all the discarded alternatives and their evaluations. The first section also repeats the design goals that were selected at the beginning of system design. These represent the criteria that developers used to evaluate alternative solutions.

The next two sections are composed of a list of issues, formatted in the same way as the access control issue we described earlier in Figure 9-24. The list of issue can constitute the systematic justification of the design or be simply a collection of issues captured in the

course of system design. Issues may be related to each other with subissue and consequent issue references. Finally, pointers to issues in this document can be inserted in the SDD in the relevant sections for ease of navigation.

System Design Rationale Document (SDRD)

Revision history

1. Introduction
    1.1 Purpose of the document
    1.2 Design goals
    1.3 Definitions, acronyms, and abbreviations
    1.4 References
    1.5 Overview

2. Rationale for current software architecture

3. Rationale for proposed software architecture
    3.1 Overview
    3.2 Rationale for subsystem decomposition
    3.3 Rationale for hardware/software mapping
    3.4 Rationale for persistent data management
    3.5 Rationale for access control and security
    3.6 Rationale for global software control
    3.7 Rationale for boundary conditions

Glossary

Appendixes

Index

FIGURE 9-25. An example of a template for the *System Design Rationale Document*.

The second section of the SDRD describes the rationale for the system being replaced. If there is no previous system, this section can be replaced by rationale for similar systems. The purpose of this section is to make explicit prior alternatives that have been explored and issues that developers should watch for.

The third section of the SDRD describes the rationale for the new system. Paralleling the structure of the SDD, this section is divided into seven subsections:

- *Overview* presents an bird view of this section, including a summary of the most critical issues that were dealt during system design.

- *Rationale for subsystem decomposition* justifies the selected system decomposition. How does it minimize coupling? How does it increases coherence? How does it satisfy the design goals set in the first section? More generally, any issue which impacted system decomposition is listed here.
- *Rationale for hardware/software mapping* justifies the selected hardware configuration, the assignment of subsystems to nodes, and legacy code issues.
- *Rationale for persistent data management* justifies the selection of data storage mechanism. Why were flat files / relational database / object-oriented database selected? Which design criteria drove this decision? Which other issues should the storage component deal with?
- *Rationale for access control and security* justifies the selected access control implementation. Why were access lists / capabilities chosen? How is the access control integrated with other subsystems distributing information, such as the middleware or the notification subsystem? The access control issue for CTC would be included in this section.
- *Rationale for global software control* justifies the selected control mechanism. Which legacy component constrained the software control? Were there incompatible legacy components? How was this dealt with?
- *Rationale for boundary conditions* justifies each boundary condition and its handling. Why does the system check the consistency of the database at start-up? Why does the system give a ten minute warning to the dispatchers before shutdown (as opposed to 20 minutes or 2 hours)?

The SDRD should be written at the same time as the SDD and revised whenever the SDD is revised. This ensures that both documents are consistent and encourages developers to make rationale explicit. The SDRD should be revised not only when the design is changed, but also when missing rationale is found (e.g., during reviews).

9.5.2. Assigning responsibilities

Assigning responsibilities for capturing and maintaining rationale is the most critical management decision in making rationale models useful. Maintaining rationale can easily be perceived as an intruding bureaucratic process through which developers need to justify all decisions. Instead, rational models should be maintained by a small number of people who have access to all developers information, such as drafts of design documents and developer newsgroups. This small group of people, becoming historians of the system design, become useful to the other developers when providing rationale information, and thus, create an incentive to developers to provide them with information. Below are the main roles related to rationale model maintenance:

- The ***minute taker*** records rationale in meetings. This includes recording chronological statements during the meeting and restructuring them with issues after the meeting (see Section 9.4.2).
- The ***rationale editor*** collects and organizes information related to rationale. This includes obtaining the meeting minutes from the minute taker, prototype and technology evaluations reports from developers, and drafts of all system models and design documents from the technical writers. The rationale editor imposes minimal overhead on the developers and the writers by performing the structuring and indexing role. The developers need not provide information structured as issue models, however, the rationale editor constructs index all information as issues.
- The ***reviewer*** examines the rationale captured by the rationale editor and identifies holes to be reconstructed. The reviewer then collects the relevant information from communication records and, if necessary, from the developers. This role should not be a management role or a quality assurance role: the reviewer must directly benefit the developers in order to be able to collect valid information. This role can be combined with the rationale editor role.

The size of the project determines the number of minute takers, rationale editors, and reviewers. The following heuristics can be used for assigning these roles:

- *One minute taker per team.* Meetings are usually organized by subsystem team or by cross functional team. A developer of each team can function as a minute taker, thus distributing this time consuming role across the project.
- *One rationale editor per project.* The role of rationale editor for a project is a full time role. Unlike the role of minute taker which can be rotated, the role of rationale editor requires consistency and should be assigned to a single person. In small projects, this role can be assigned to the system architect (see Chapter 8, *System Design*).
- *Increase the number of reviewers after delivery*. When the system is delivered and the number of developers directly need for the project decreases, some developers should be assigned the reviewer role for salvaging and organizing as much information as possible. Rationale information is still recoverable from developers memories but disappears quickly as developers move to other projects.

9.5.3. Heuristics for communicating about rationale

A large part of communication *is* rationale information, given that argumentation is, by definition, rationale (see Section 9.2). Developers argue about design goals, whether a given issue is relevant or not, the benefit of several solutions, and their evaluation. Rationale constitutes a large and complex body of information, usually larger than the system itself. Argumentation, moreover, occurs most often in small forums, for example, in a team meeting or a conversation at the coffee machine. The challenge of communicating about

rationale is to make this information accessible to all concerned parties without causing an information overload. In this chapter, we focused on techniques for capturing and structuring rationale, such as using issue models in minutes, follow-up conversations, and rationale documentation. In addition, the following heuristics can be used to increase the structure of rationale and facilitate its navigation:

- *Name issues consistently*. Issues should be consistently and uniquely named across minutes, newsgroups, email messages, and documents. Issues can have a number (e.g., 1291) and a short name (e.g., "the access / notification issue") for ease of reference.
- *Centralize issues.* Although issues will be discussed in a variety of context, encourage one context (e.g., a newsgroup or an issue base) to be a central repository of issues. This issue-base should be maintained by the rationale editor but could be used and extended by any developer. This enables developers to search for information quickly.
- *Cross reference issues and system elements.* Most issues apply to a specific element in the system models (e.g., a use case, an object, a subsystem). Finding which model element a specific issue applies to is straightforward. However, finding which issues apply to a specific model element is a much more difficult problem. To facilitate this type of query, issues should be attached to the applying model element when issues are raised.
- *Manage change.* Rationale evolves as system models do. Thus, configuration management should be applied consistently to rationale and documents as it is applied to system models.

Capturing and structuring rationale not only improves communication about rationale, but also facilitates communication about the system models. Integrating both rationale and system information enables developers to better maintain both types of information.

9.5.4. Issue modeling and negotiation

Most important decisions in development are the result of negotiation. Different parties representing different, and often conflicting, interests come to a consensus on some aspect of the system: Requirements analysis includes the negotiation of functionality with a client. System design includes the negotiation of subsystem interfaces among developers. Integration includes the resolution of conflicts between developers. We use issue-modeling to represent the information exchanged during these negotiations in order to capture rationale. We can also use issue-modeling to facilitate negotiations.

Traditional negotiation, which consists of bargaining over positions, is often time consuming and inefficient, especially when the negotiating parties hold incompatible positions. Effort is spent in defending one's position, citing all its advantages, while the

opposing part spends effort in denigrating the other's position, citing all its disadvantages. The negotiation either progresses in small steps towards a consensus or is ended by an arbitrary solution to the negotiated issue. Furthermore, this can occur even when negotiating parties have compatible interests: when defending positions, people have greater trouble evolving or changing their position without losing credibility. The Harvard method of negotiation [Fischer et. al., 1991] addresses these points by taking the focus away from positions. We rephrase several important points of the Harvard method in terms of issue modeling:

- *Separate developers from proposals.* Developers can spent a lot of resources developing a specific proposal (i.e., a position), to the point that a criticism of the proposal is taken as a personal criticism of the developer. Developers and proposals should be separated in order to make it easier to evolve or discard a proposal. This can be done by having multiple developers work on the same proposal or have all concerned parties participate in the development of all proposals. Separating the design and the implementation work can further facilitate this distinction. By ensuring that negotiation comes before implementation and before substantial resources are committed, developers are able to evolve proposals into ones that all can live with.
- *Focus on criteria, not on proposals.* As stated before, most arguments can be tracked to the criteria, often implicit, used for evaluation. Once an accepted set of criteria is in place, evaluating and selecting proposals is far less controversial. Furthermore, criteria are much less subject to change than other factors in the project. Agreeing on criteria early also facilitates revisions to decisions.
- *Take into account all criteria instead of maximizing a single one.* Different criteria reflect interests of different parties. Performance criteria are usually motivated by usability concerns. Modifiability criteria are motivated by maintenance concerns. Even if some criteria are considered higher priority that other ones, optimizing only these high priority criteria risks leaving out of the negotiation one or more parties.

Viewing development as a negotiation acknowledges the social aspects of development. Developers are persons who, in addition to technical opinions, can have an emotional perspective on different solutions. This can influence (and sometimes interfere) with their relationships to other developers as conflicts arise. Using issue-modeling to capture rationale and drive decisions can integrate and improve both the technical and social aspects of development.

9.5.5. Conflict resolution strategies

Occasionally, project participants fail to come to a consensus through negotiation. In such cases, it is critical that conflict resolution strategies are already in place to deal with the situation. The worst design decisions are those which are not taken because of the lack of

consensus or the absence of conflict resolution strategies. This delays critical decision until late in the development, resulting in high redesign and recording costs.

Many different conflict resolution strategies are possible. For example, consider the five following strategies:

- *Majority wins*. In case of conflict, a majority vote could remove the deadlock and resolve the decision. Several collaboration tools enable users to attach weights to different arguments in the issue model, and thus, to compute which proposal should be selected with an arithmetic formula [Purvis et al, 1996]. This assumes that the opinion of each participants matters equally and that, statistically, the group makes the right decisions.
- *Owner has last word.* In this strategy, the owner of an issue (the person who raised it) is responsible for deciding the outcome. This assumes that the owner has the largest stake in the issue.
- *Management is always right*. An alternative strategy is to fall back on the organizational hierarchy. If a group is unable to reach consensus, the manager of the group imposes a decision based on the argument. This assumes the manager is able of understanding the argument and making the right trade-offs.
- *Expert is always right*. In this strategy, an external expert, foreign to the debate, assess the situation and advises the best course of action. For example, during requirements analysis, a test user can be interviewed to evaluate the different proposals of an issue. Unfortunately, such an expert has limited knowledge of other system decisions or more generally of the design context.
- *Time decides.* As an issue is left unresolved, time becomes a pressure and forces a decision. Also, controversial issues may become easier to resolve as other decisions are made and other aspects of the system defined. The danger with this strategy is that it leads to decisions that optimize short term criteria (such as ease of implementation) and disregard long term criteria (such as modifiability and maintainability).

The *Majority wins* and the *Owner has last word* strategies do not work well. They both result in inconsistent results (multiplicity of decision makers) and in decisions that are not well supported by the rest of the participants. The *Management is always right* and *Expert is always right* strategies lead to better technical decision and better consensus when the Manager and the Expert are sufficiently knowledgeable. The *Time decides* strategy is a fallback, albeit one that may result in costly rework.

In practice, we first attempt to reach consensus, and, in case of lack of consensus, fallback on an expert or management strategy. If the expert or manager strategy fails, we let time decide or take a binding majority vote.

9.6. Exercises

1. Below is an excerpt from a system design document for an accident management system. It is a natural language description of the rationale for a relational database for permanent storage. Model this rationale with issues, proposals, arguments, criteria, and resolutions, as defined in Section 9.3.

> "One fundamental issue in database design was database engine realization. The initial non-functional requirements on the database subsystem insisted on the use of an object-oriented database for the underlying engine. Other possible options included using a relational database, a file system, or a combination of the other options. An object-oriented database has the advantages of being able to handle complex data relationships and is fully buzz word compliant. On the other hand, OO databases may be too sluggish for large volumes of data or high frequency accesses. Furthermore, existing products do not integrate well with CORBA since that protocol does not support specific programming language features such as Java associations. Using a relational database offers a more robust engine with higher performance characteristics and a large pool of experience and tools to draw upon. Furthermore, the relational data model integrates nicely with CORBA. On the down side, this model does not easily support complex data relationships. The third option was proposed to handle specific types of data which are written once and read infrequently. This type of data (including sensor readings and control outputs) has few relationships with little complexity and must be archived for extended periods of time. Files offer an easy archival solution and can handle large amounts of data. Conversely, any code would need to be written from scratch including serialization of access. We decided to use only a relational database, based on the requirement to use CORBA and in light of the relative simplicity of the relationships between the system's persistent data."

2. In Section 9.3, we examined an issue related to access control and notification in the CTC system. Select a similar issue that could occur in the development and CTC and populate it with relevant proposals, criteria, arguments, and justify a resolution. Example of such issues include:

- How should consistency between the `mainServer` and its `hotBackup`?
- How should failure of the `mainServer` be detected and the subsequent switch to the `hotBackup` implemented?

3. You are developing a CASE tool using UML as its primary notation. You are considering the integration of rationale into the tool. Describe how a developer could attach issues to different model elements. Draw a class diagram of the issue model and its association to model elements.

4. Considering the same CASE tool as in Exercise 3. You are considering the generation of rationale documents from the model. Describe the mapping between classes, issues, and the sections rationale document.

5. You are integrating a bug reporting system with a configuration management tool to track bug reports, bug fixes, feature requests, and enhancements. You are considering an issue model for integrating these tools. Draw a class diagram of the issue model,

the corresponding discussion, configuration management, and bug reporting elements.

9.7. References

[Buckingham Shum & Hammond, 1994] S. Buckingham Shum & N. Hammond, "Argumentation-based design rationale: what use at what cost?," International Journal of Human-Computer Studies, 40, 603-52, 1994.

[Conklin & Burgess-Yakemovic, 1991] J. Conklin & KC Burgess Yakemovic, "A Process-oriented Approach to Design Rationale," Human-Computer Interaction, 6 (3 & 4): 357-91, 1991.

[Dutoit et al.; 1996] A.H. Dutoit, B. Bruegge, & R.F. Coyne, "The use of an issue-based model in a team-based software engineering course." Conference proceedings of Software Engineering: Education and Practice (SEEP'96). Dunedin, New Zealand. January 1996.

[Fischer et. al., 1991] R. Fisher, W. Ury, & B. Patton. Getting to Yes: Negotiating Agreement Without Giving In (Second edition). Penguin Books. 1991.

[Kunz & Rittel, 1970] W. Kunz & H. Rittel, Issues as elements of information systems (Working Paper No. 131). Institut fuer Grundlagen der Plannung, Universitaet Stuttgart, 1970.

[Lee, 1990] J. Lee, "A qualitative decision management system," In P.H Winston & S. Shellard (Eds.), ARTIFICIAL Intelligence at MIT: Expanding Frontiers. Cambridge, MA, MIT Press. Vol 1, 104-33, 1990.

[Lee, 1997] J. Lee, "Design Rationale Systems: Understanding the Issues.", IEEE Expert, May / June 1997.

[MacLean et. al, 1991] A. MacLean, R.M. Young, V. Bellotti, & T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," Human-Computer Interaction, 6 (3&4): 201-50, 1996.

[Moran & Carroll, 1996] T.P. Moran & J.M. Carroll (Eds.), Design Rationale: Concepts, Techniques, and Use, Lawrence Erlbaum Associates, NJ, 1996.

[Potts & Bruns, 1988] C. Potts & G. Bruns, "Recording the Reasons for Design Decisions," In Proceedings of the 10th International Conference on Software Engineering, 418-27, 1988.

[Potts, 1996] C. Potts, "Supporting Software Design: Integrating Design Methods and Design Rationale," In T.P. Moran & J.M. Carroll (Eds.) Design Rationale: Concepts, Techniques, and Use. Lawrence Erlbaum Associates, Mahwah, N.J., 1996.

[Purvis et al, 1996] M. Purvis, M. Purvis, & P. Jones. "A Group Collaboration Tool for Software Engineering Projects," Conference proceedings of Software Engineering: Education and Practice (SEEP'96). Dunedin, New Zealand. January 1996.

[Shipman & McCall, 1997] F.M. Shipman III & R.J. McCall, "Integrating Different Perspectives on Design Rationale: Supporting the Emergence of Design Rationale from Design Communication," Artificial Intelligence in Engineering Design, Analysis, and Manufacturing, Vol. 11, No. 2, 1997.