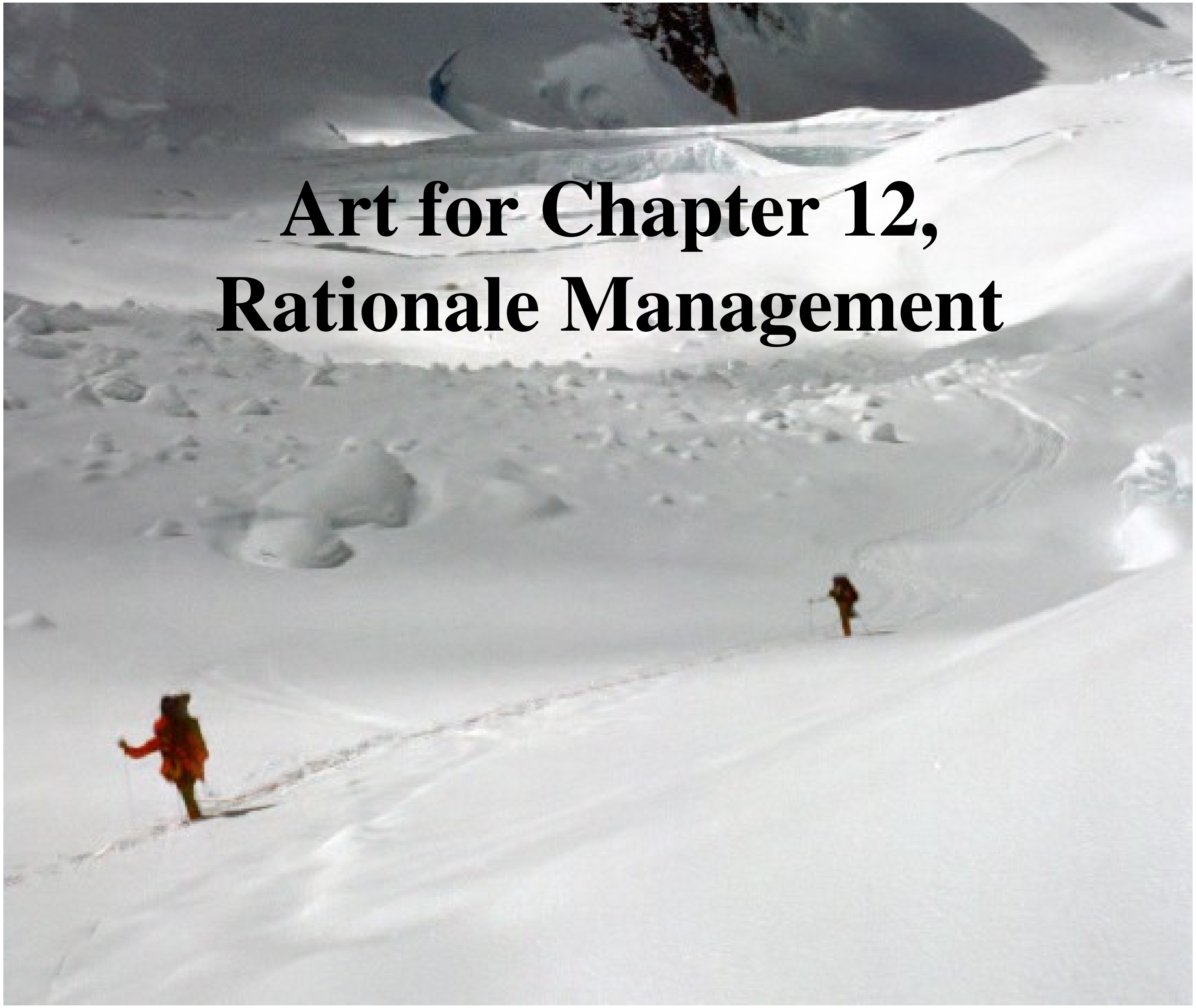
Object-Oriented Software Engineering
Using UML, Patterns, and Java



Art for Chapter 12, Rationale Management

The problem

Rationale is the justification of decisions.

The models we have described until now represent the system. Rationale models represent the reasoning that leads to the system, including its functionality and its implementation. Rationale is critical in two areas: it supports the decision making, and it supports the capture of knowledge. Rationale includes:

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

The problem_2

In the context of decision making, the rationale improves the quality of decisions by making decision elements—such as criteria, priorities, and arguments—explicit.

In the context of knowledge capture, the rationale is the most important information in the development process when changing the system. For example:

- when functionality is added to the system, the rationale enables developers to track which decisions should be revisited and which alternatives have already been evaluated.

The problem_3

- 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 that developers generate, and thus, is the most difficult to maintain and update.

Capturing rationale represents an up-front investment with long-term returns.

Rationale models enable developers to deal with change. Capturing the justification of decisions effectively models the dependencies between starting assumptions and decisions. When assumptions change, decisions can be revisited.

Overview

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

- Issue. To each decision corresponds an issue to be solved so that development can proceed. An important part of the rationale is a description of the specific issue that is being solved. Issues are usually phrased as questions: How should a ham be cooked? How should years be represented?

- Alternatives. Are possible solutions that could address the issue under consideration. These include alternatives that were explored but discarded because they did not satisfy one or more criteria. Representing years with a binary 16-bit number requires too much processing.

Overview_2

- **Criteria**. Criteria are desirable qualities that the selected solution should satisfy. For example:
  - Developers in the 1960s minimized memory foot prints.
- 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 tradeoffs (e.g., timely delivery versus 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 discussion that a consensus is reached or a decision imposed.

Overview_3

- **Decisions**. The resolution of an issue representing the selected alternative according to the criteria that were used for evaluation and the justification of the selection. 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.

Overview_4

We make decisions throughout the development process, and 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 together 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.

Overview_5

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 facilitates mitigation if these risks become problems.

Overview_6

- **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 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.

Overview_7

We distinguish four levels of rationale capture:

1. **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 E-mail messages, memos, and faxes.

2. **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.

3. **Rationale capture**. Major effort is spent in capturing rationale as decisions are made.

Overview_8

Rationale information is documented as a separate model and cross-referenced with other models. For example, issue models represent rationale with a graph of nodes, each representing an issue, an alternative, or an evaluation criteria. The rationale behind the requirements model can then be captured by attaching issue models to use cases.

4. **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.

Figure 12-1, An example of a CTC track section display (simplified for this example).

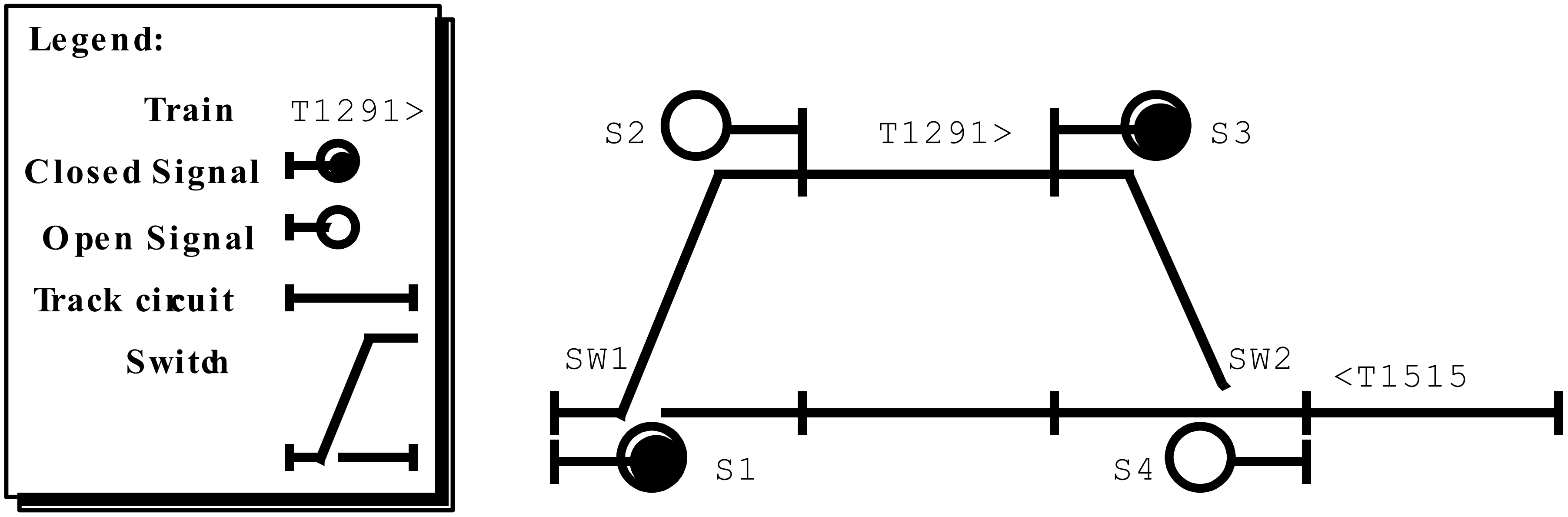


Figure 12-2, An example of issues.

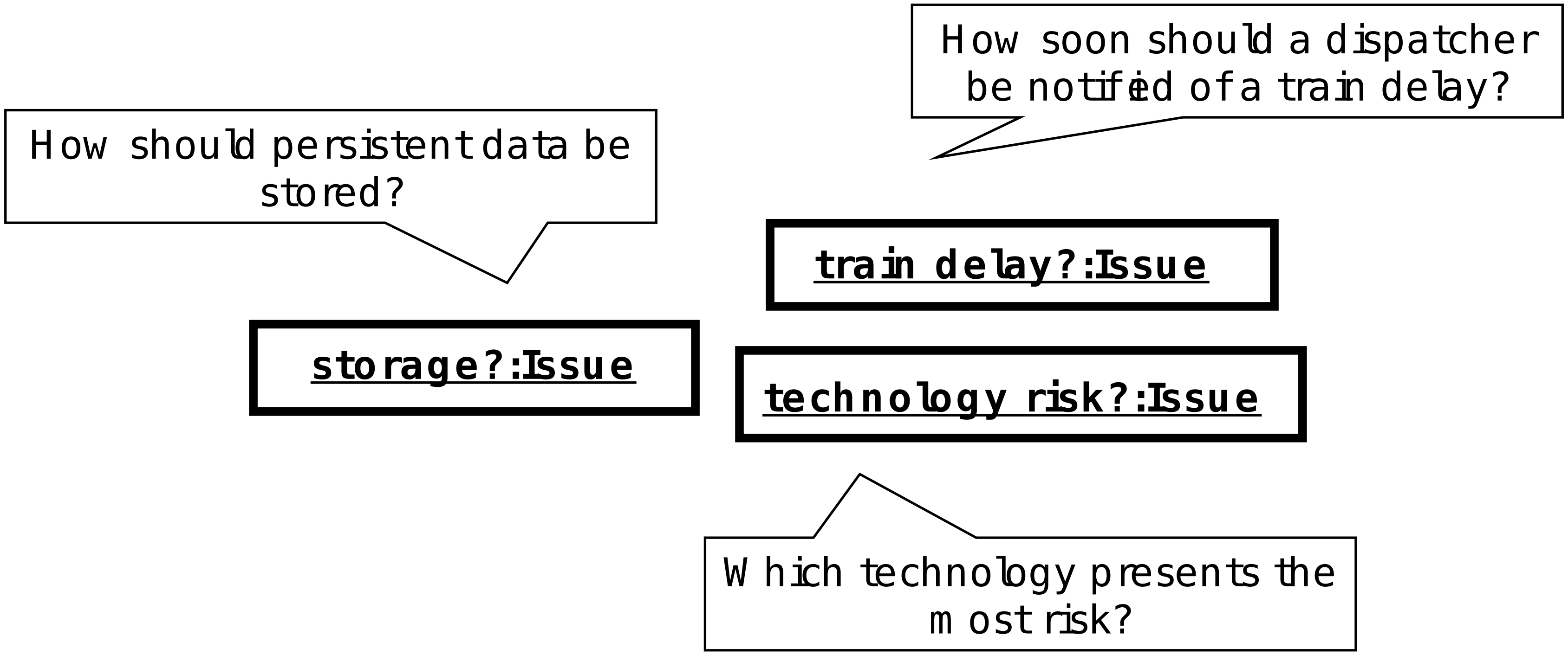

How soon should a dispatcher be notified of a train delay?
How should persistent data be stored?
train delay?:Issue
storage?:Issue
technology risk?:Issue
Which technology presents the most risk?

Figure 12-3, CTC interface issues.

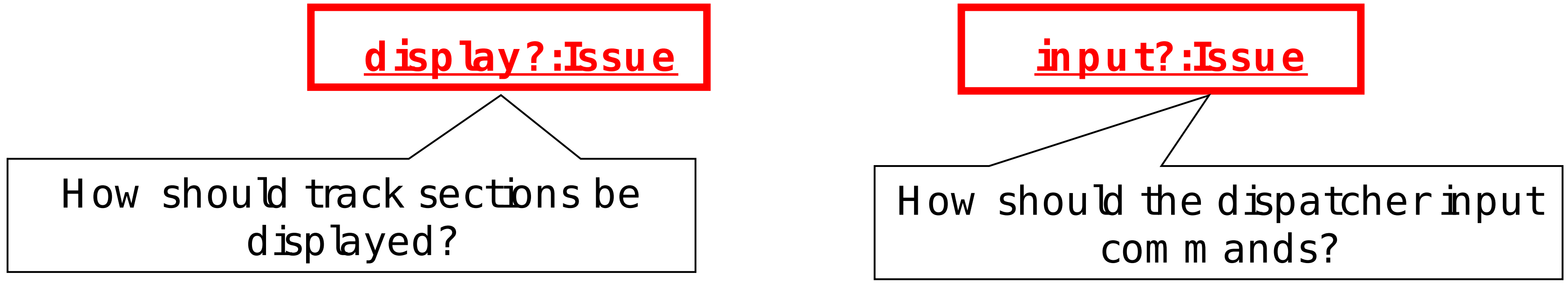
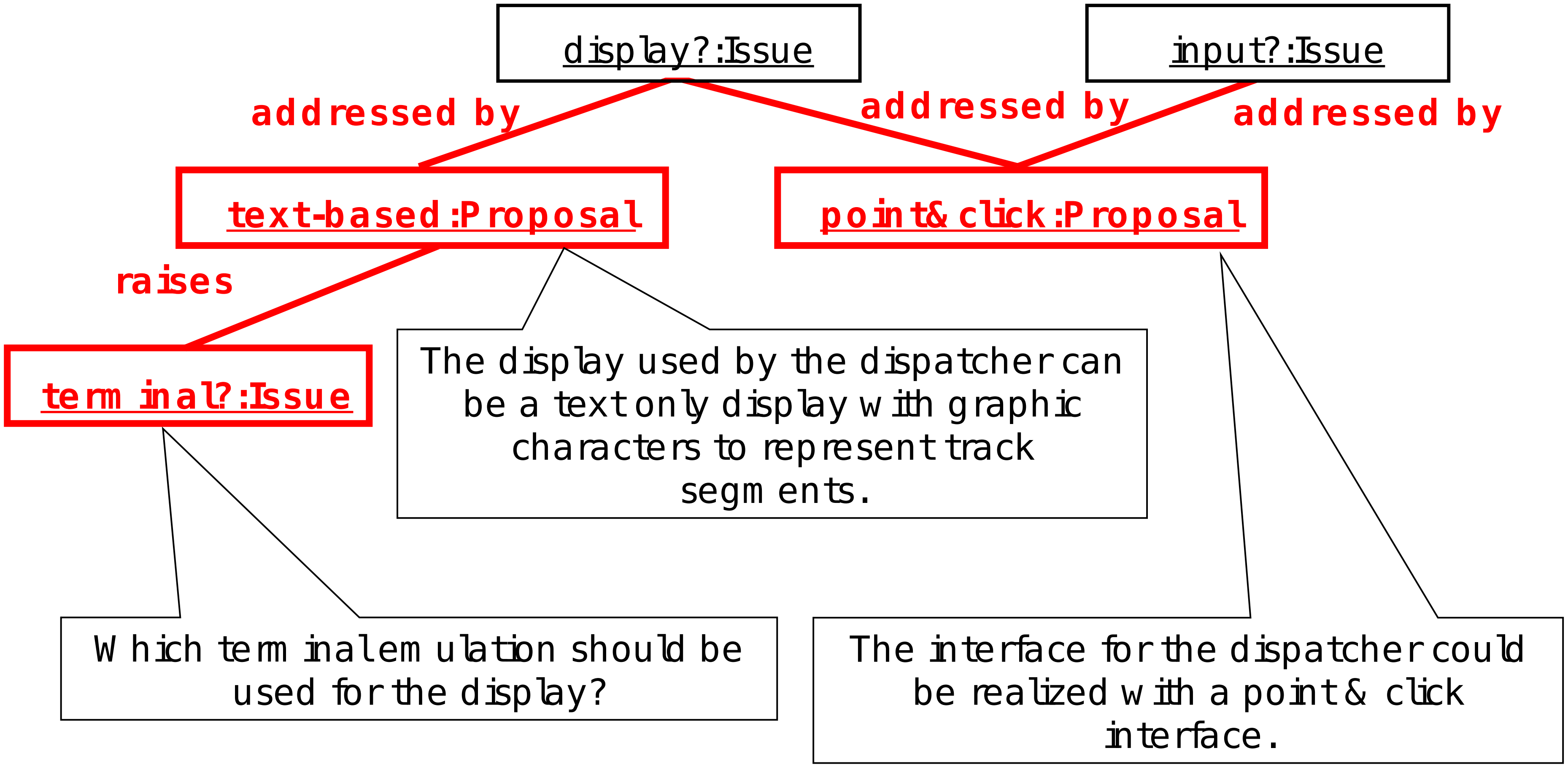

display?:Issue
How should track sections be displayed?
input?:Issue
How should the dispatcher input commands?

Figure 12-4, An example of proposals and consequent issue



The display used by the dispatcher can be a text only display with graphic characters to represent track segments.

Which terminal emulation should be used for the display?

The interface for the dispatcher could be realized with a point & click interface.

Figure 12-5, An example of criteria and assessments

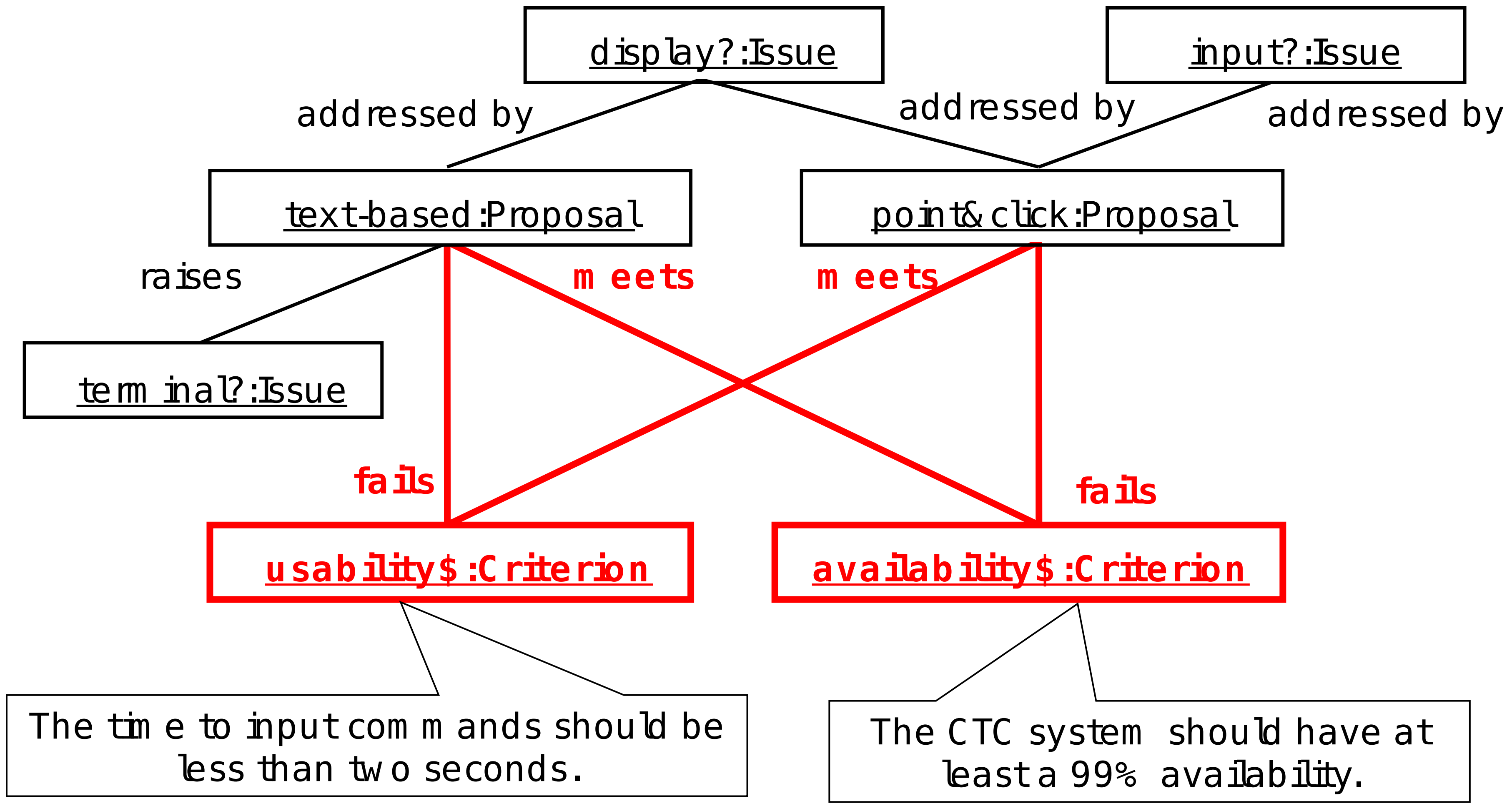
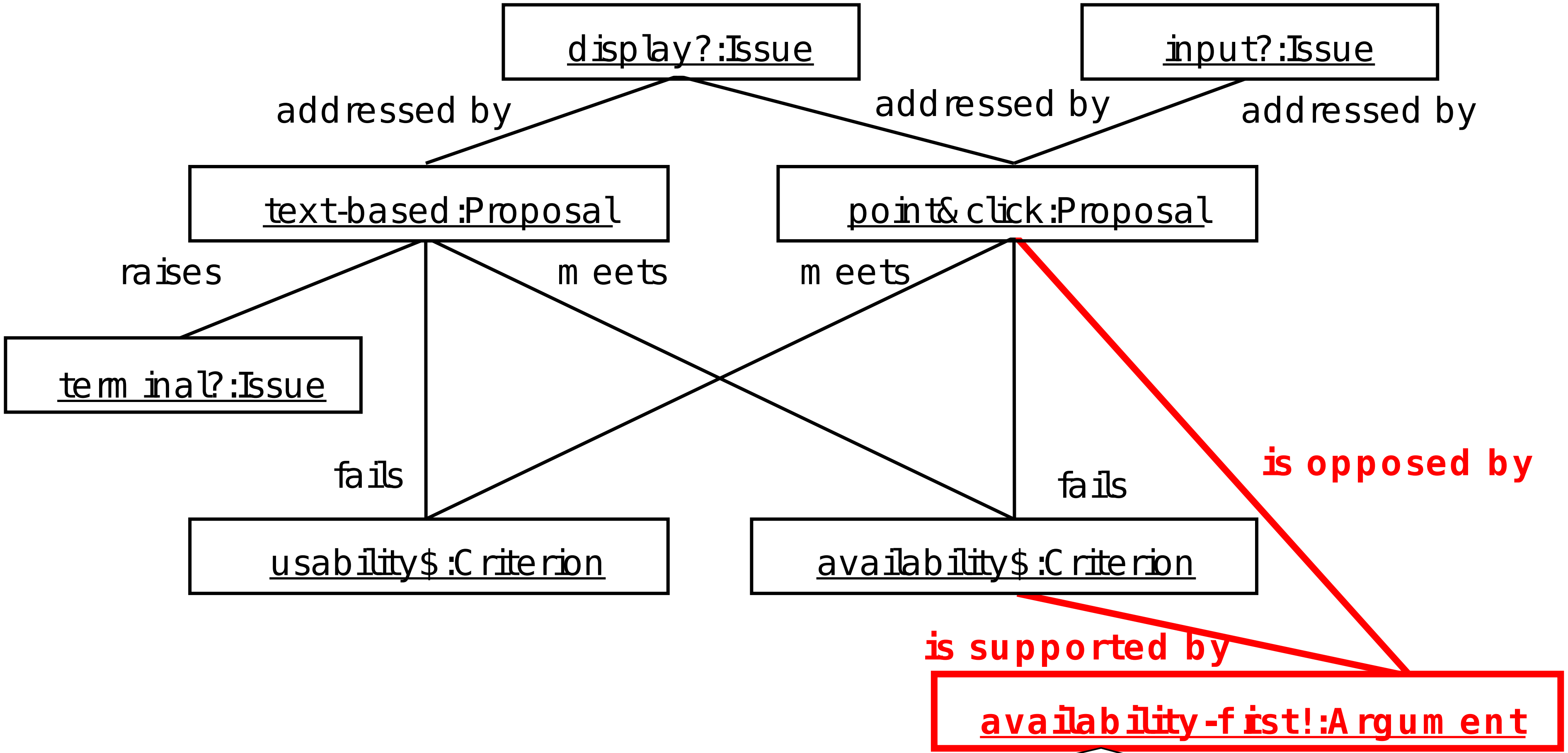


Figure 12-6, An example of an argument



Point&click interfaces are more complex to implement than text-based interfaces. Hence, they are also more difficult to test. The point&click interface risks introducing fatal errors in the system that would offset any usability benefit the interface would provide.

Figure 12-7, An example of closed issue.

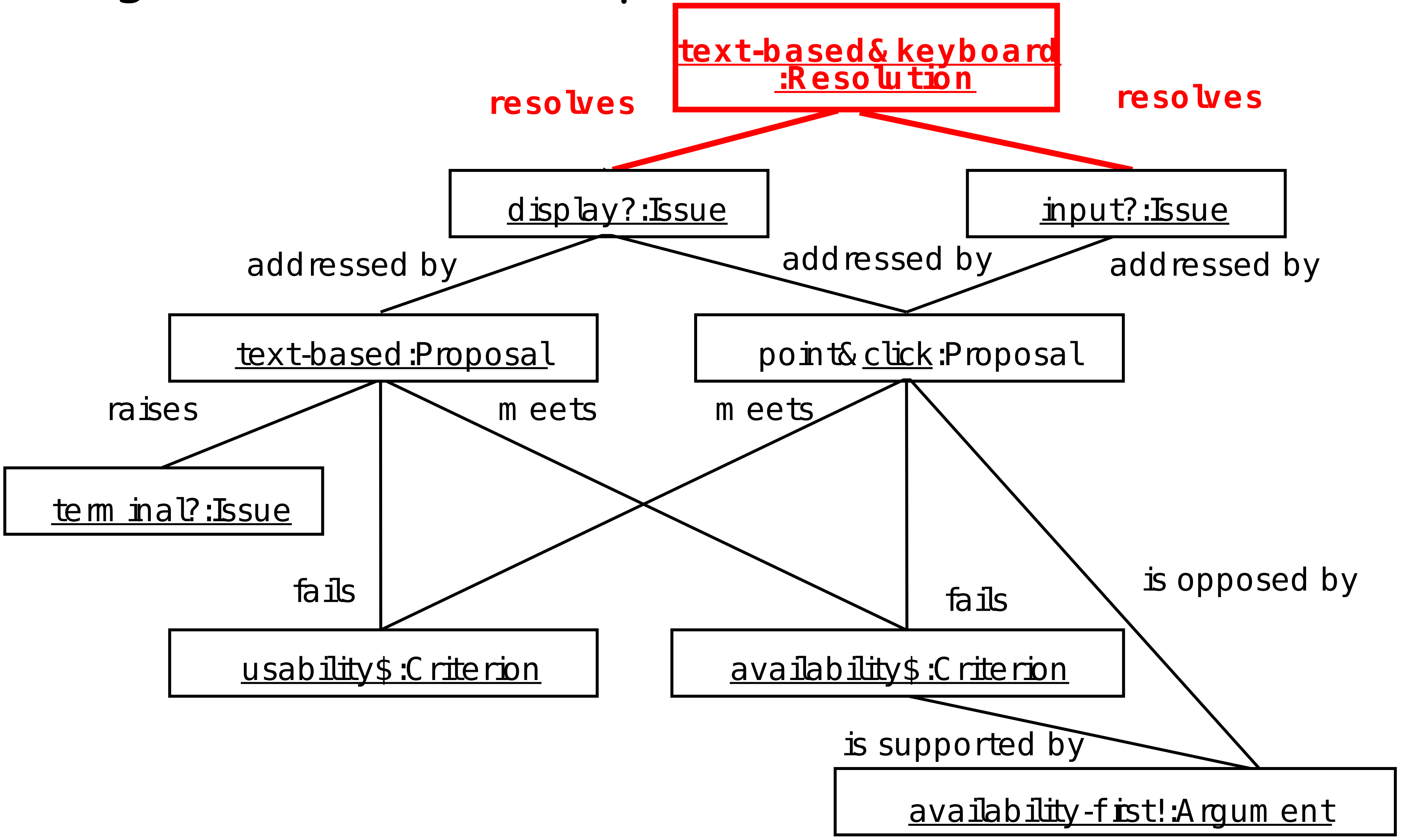


Figure 12-8, An example of implementation of a resolution.

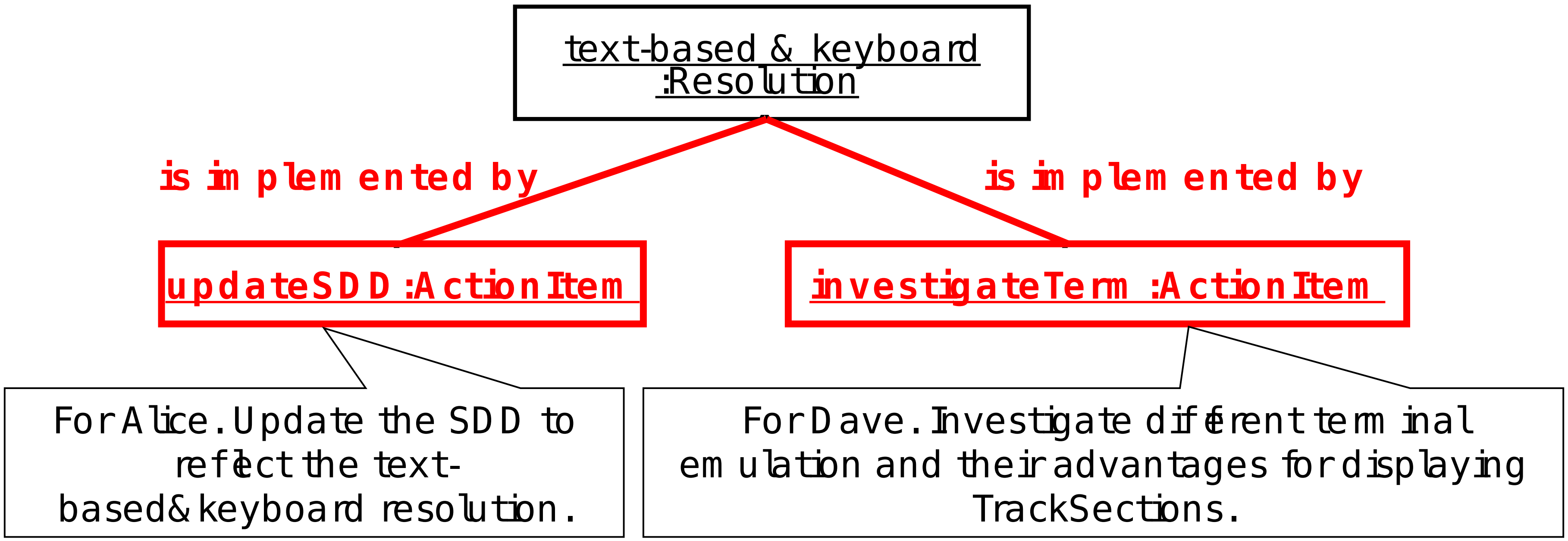

text-based & keyboard :Resolution
is implemented by
is implemented by
updateSDD :ActionItem
investigateTerm :ActionItem
For Alice. Update the SDD to reflect the text-based & keyboard resolution.
For Dave. Investigate different terminal emulation and their advantages for displaying TrackSections.

Figure 12-9, The IBIS model.

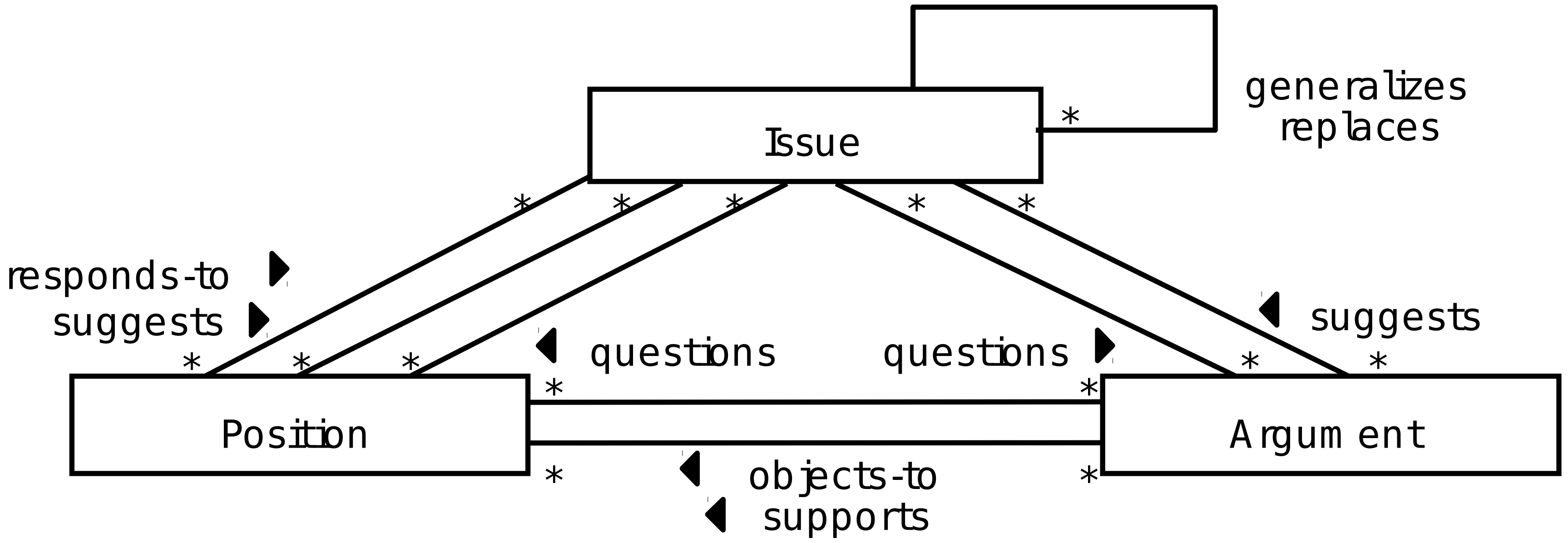


Figure 12-10, Decision Representation Language.

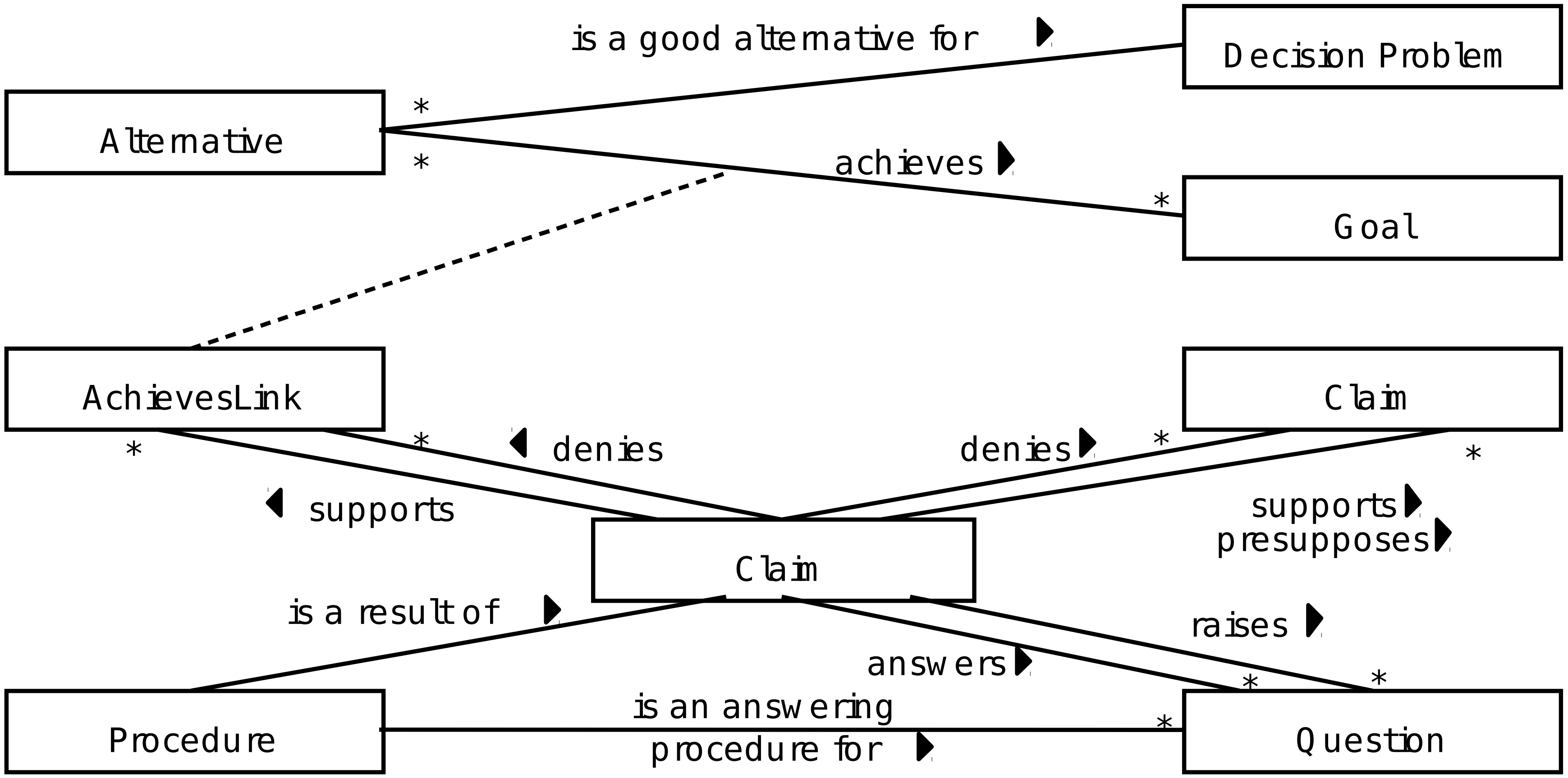


Figure 12-11, Questions, Options, Criteria model.

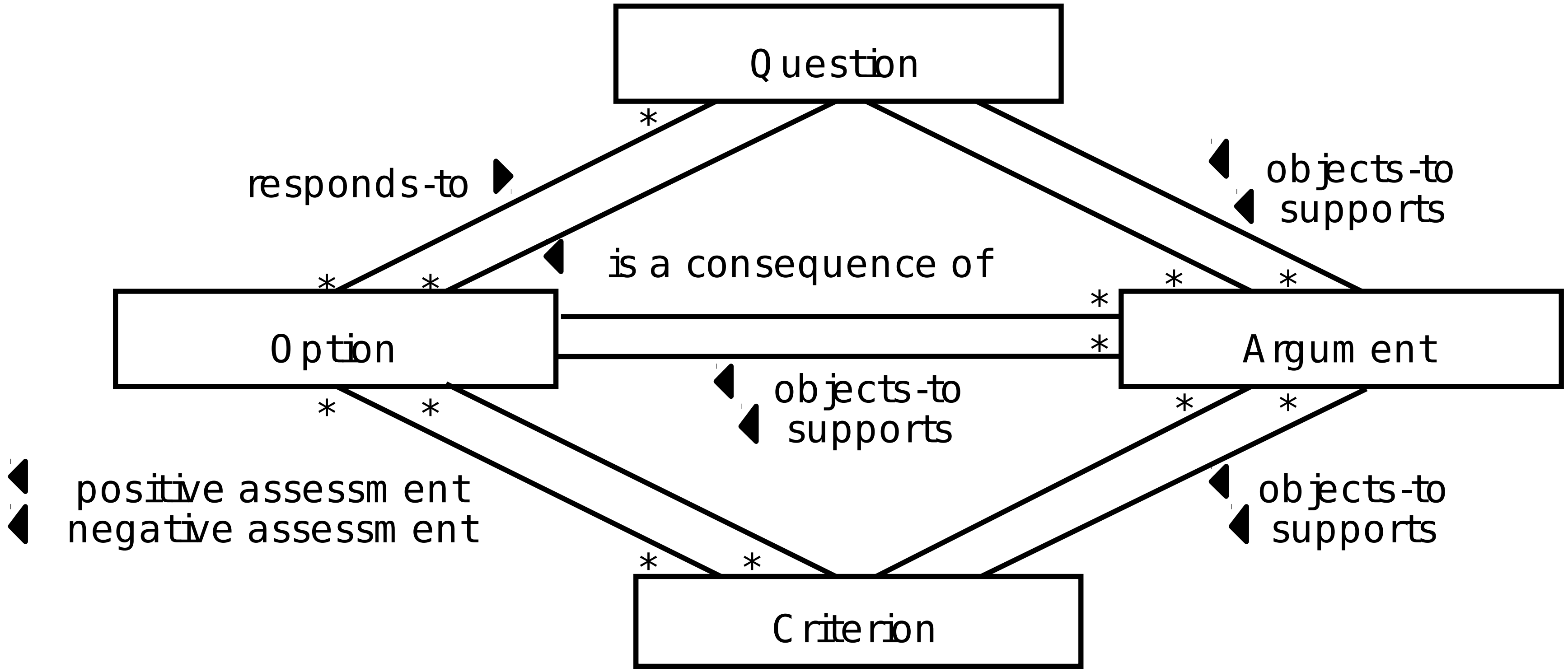


Figure 12-12, An example of goal refinement using the NFR Framework for the ATM authentication mechanism.

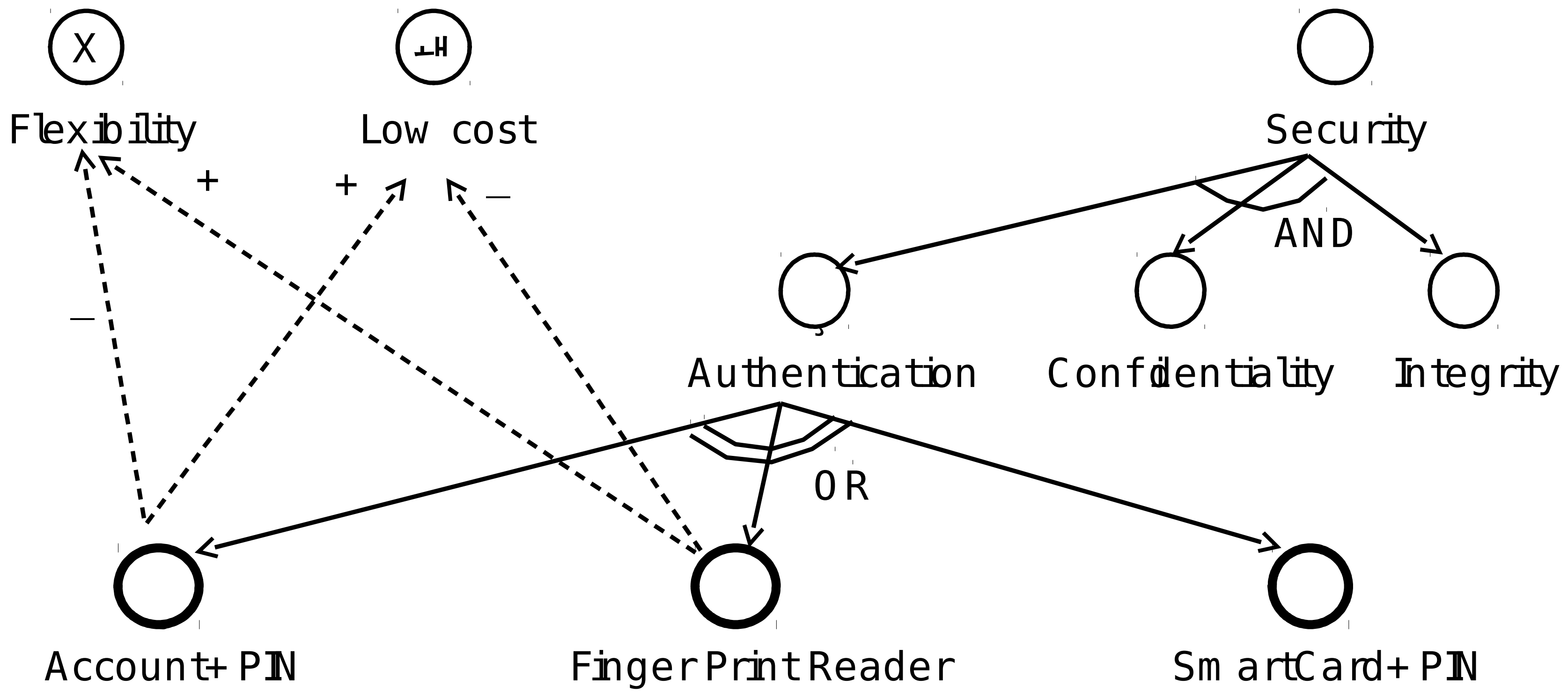


Figure 12-13, Subsystem decomposition for CTC.

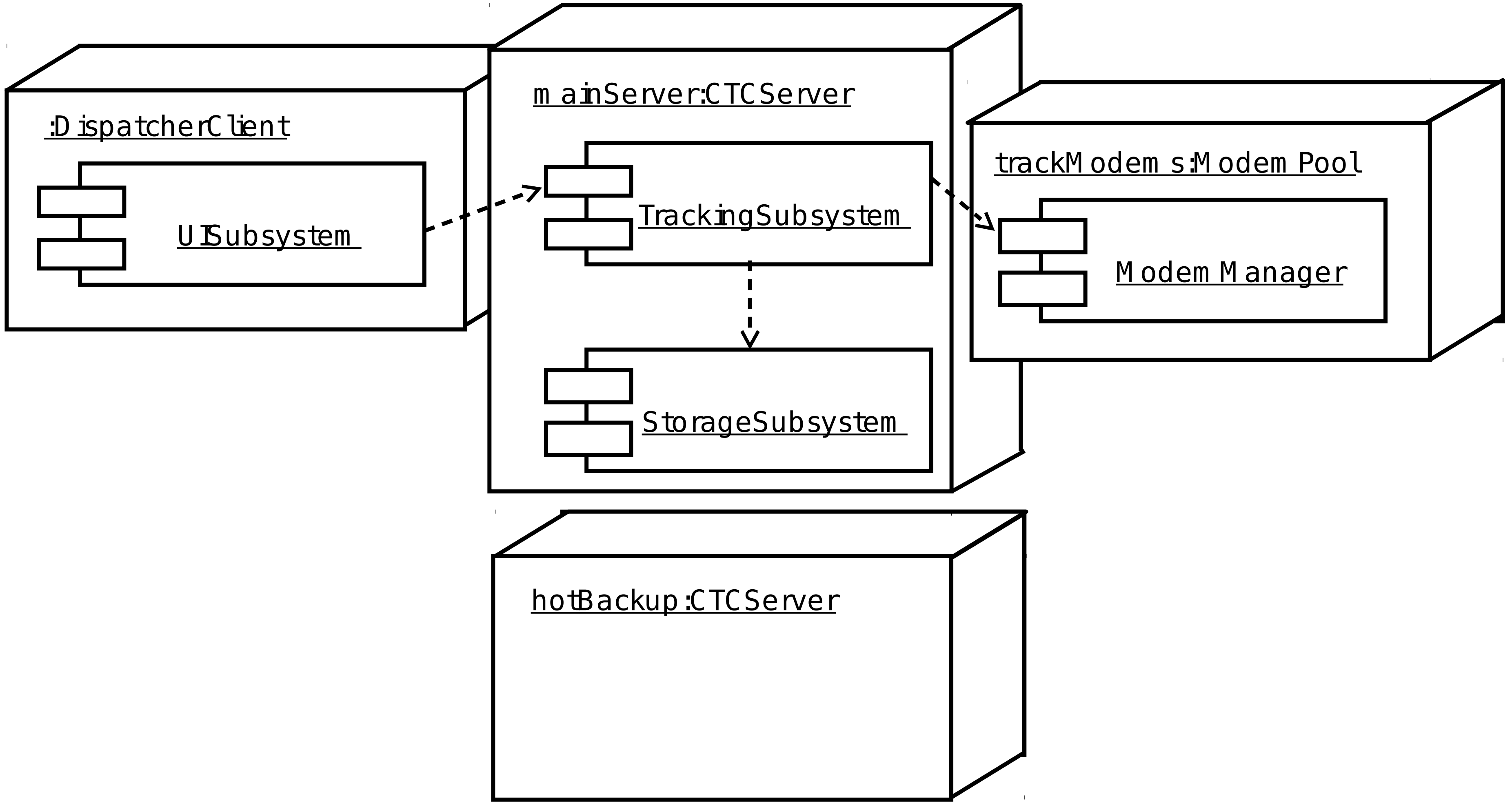


Figure 12-14, Agenda for the access control discussion of CTC

AGENDA: Integration of access control and notification

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, such as 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 12-15, Proposal P[1]: The access is controlled by the TrackSection object with an access list.

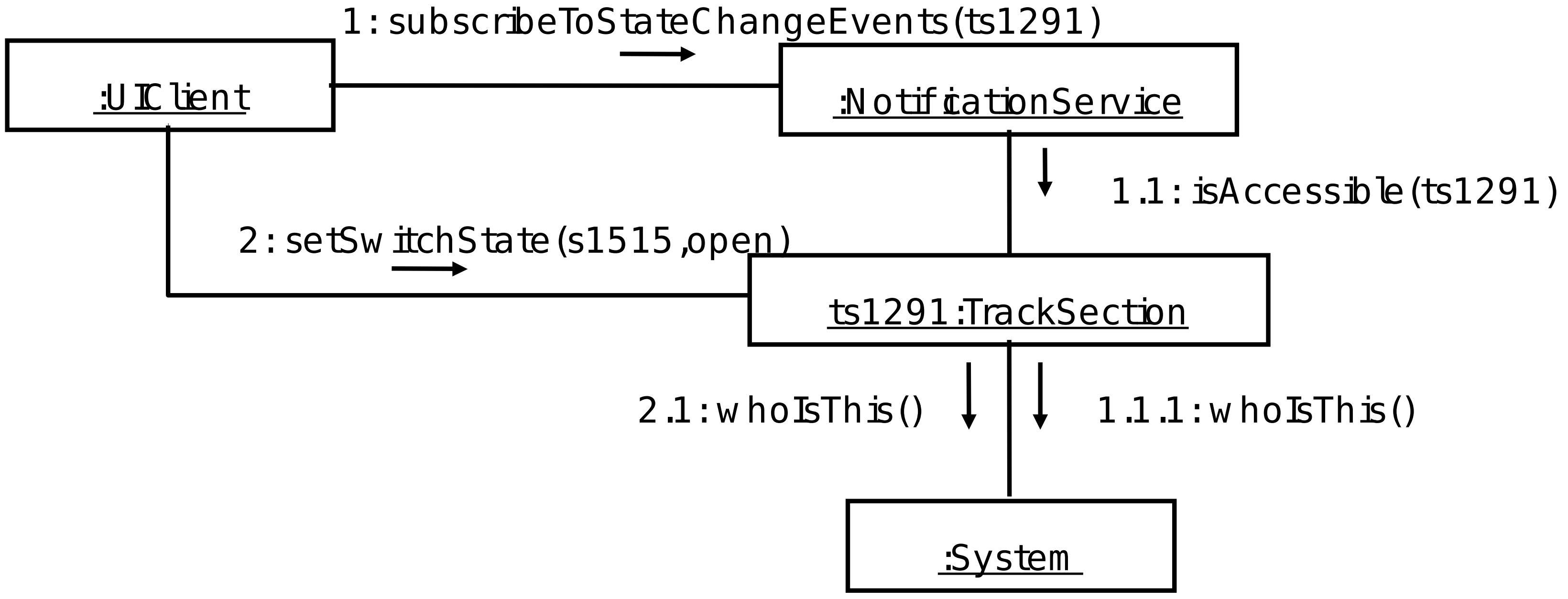

1: subscribeToStateChangeEvents(ts1291)
:UIClient
:NotificationService
1.1: isAccessible(ts1291)
2: setSwitchState(s1515, open)
ts1291:TrackSection
2.1: whoIsThis()
1.1.1: whoIsThis()
:System

Figure 12-16, Proposal P[2]: The UIClient subscribes to track section events via the subscribeToEvents() operation on the TrackSection.

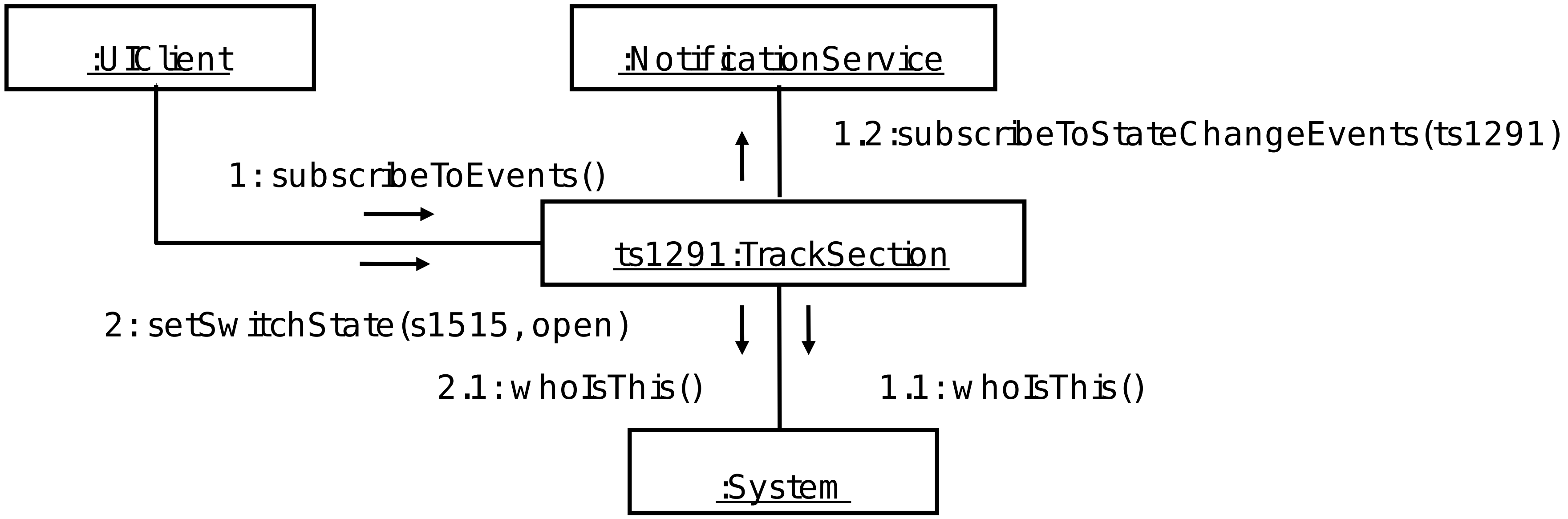


Figure 12-17, Proposal P[3]: The access to operations that modify TrackSections is controlled by the TrackSection object with an access list.

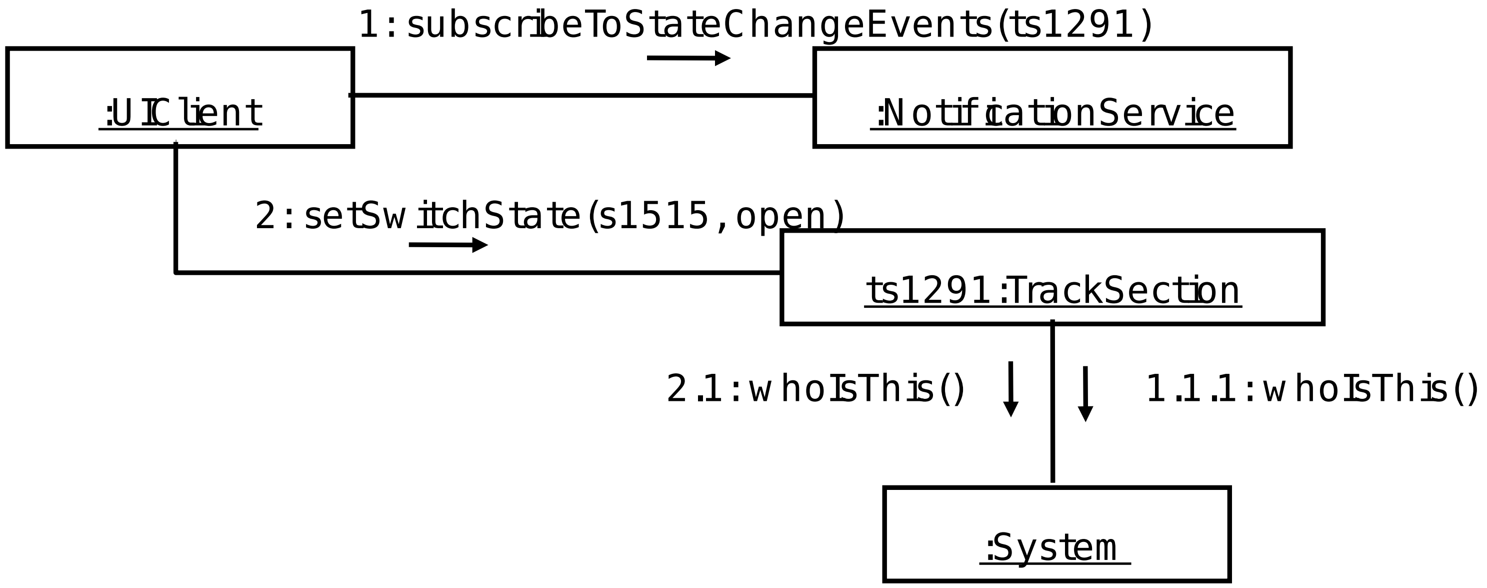


Figure 12-18, Chronological minutes for the access control discussion of CTC.

CHRONOLOGICAL MINUTES: Integration of access control and notification

4. Discussion

...

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 who 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 calls 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 not used for changing the `TrackSection` state, there is 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.

...

Figure 12-19, Structured minutes for the access control discussion of CTC.

STRUCTURED MINUTES: Integration of access control and notification

4. Discussion

...

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 a single 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.

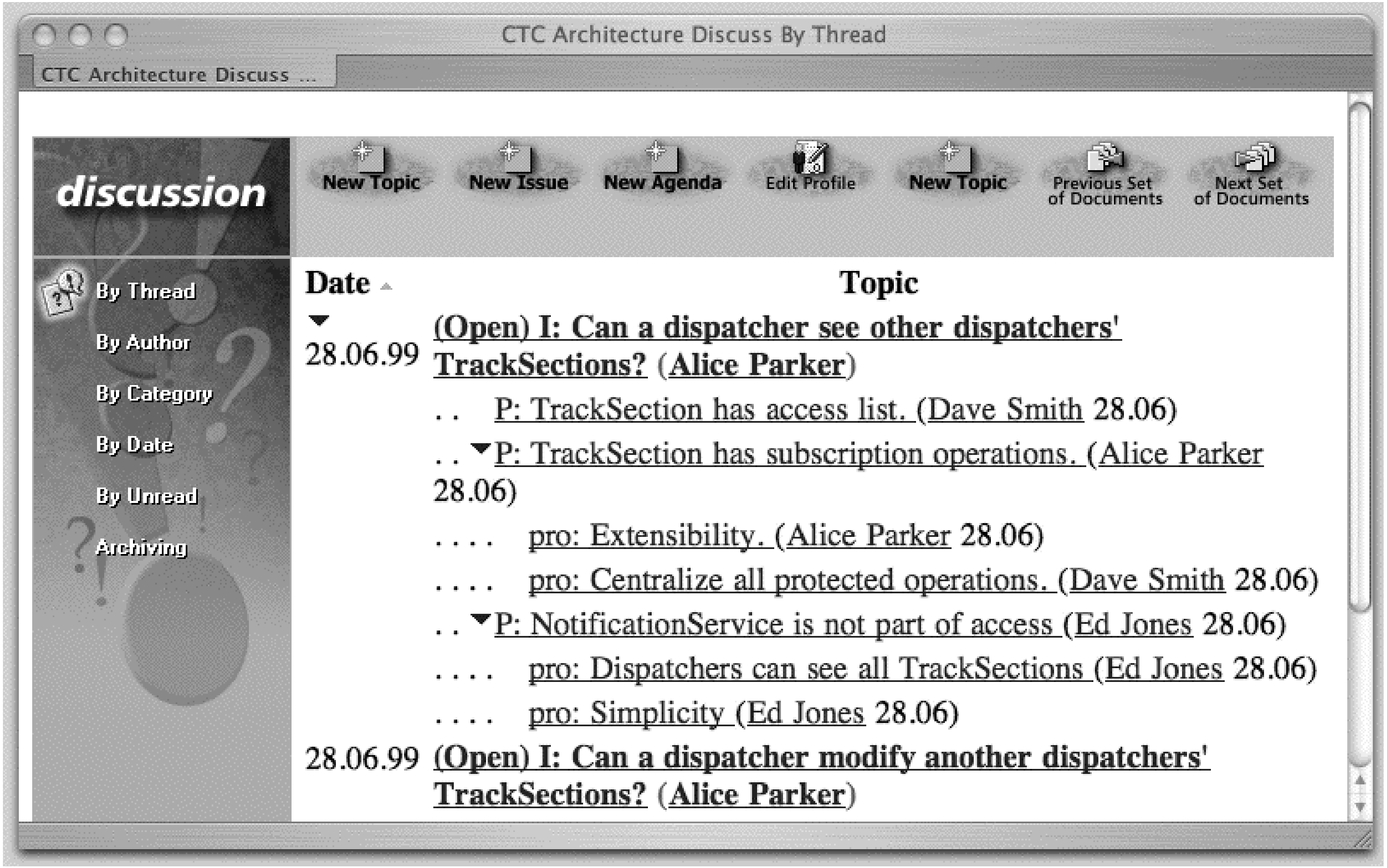
A[3.2] for P[3.3] Dispatchers can see the state of any TrackSections (see R[1]).

A[3.3] for P[3.3]: Simplicity.

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

...

Figure 12-21, An example of issue database.



Date | Topic

28.06.99 (Open) I: Can a dispatcher see other dispatchers' TrackSections? (Alice Parker)

. . P: TrackSection has access list. (Dave Smith 28.06)

. . P: TrackSection has subscription operations. (Alice Parker 28.06)

. . . . pro: Extensibility. (Alice Parker 28.06)

. . . . pro: Centralize all protected operations. (Dave Smith 28.06)

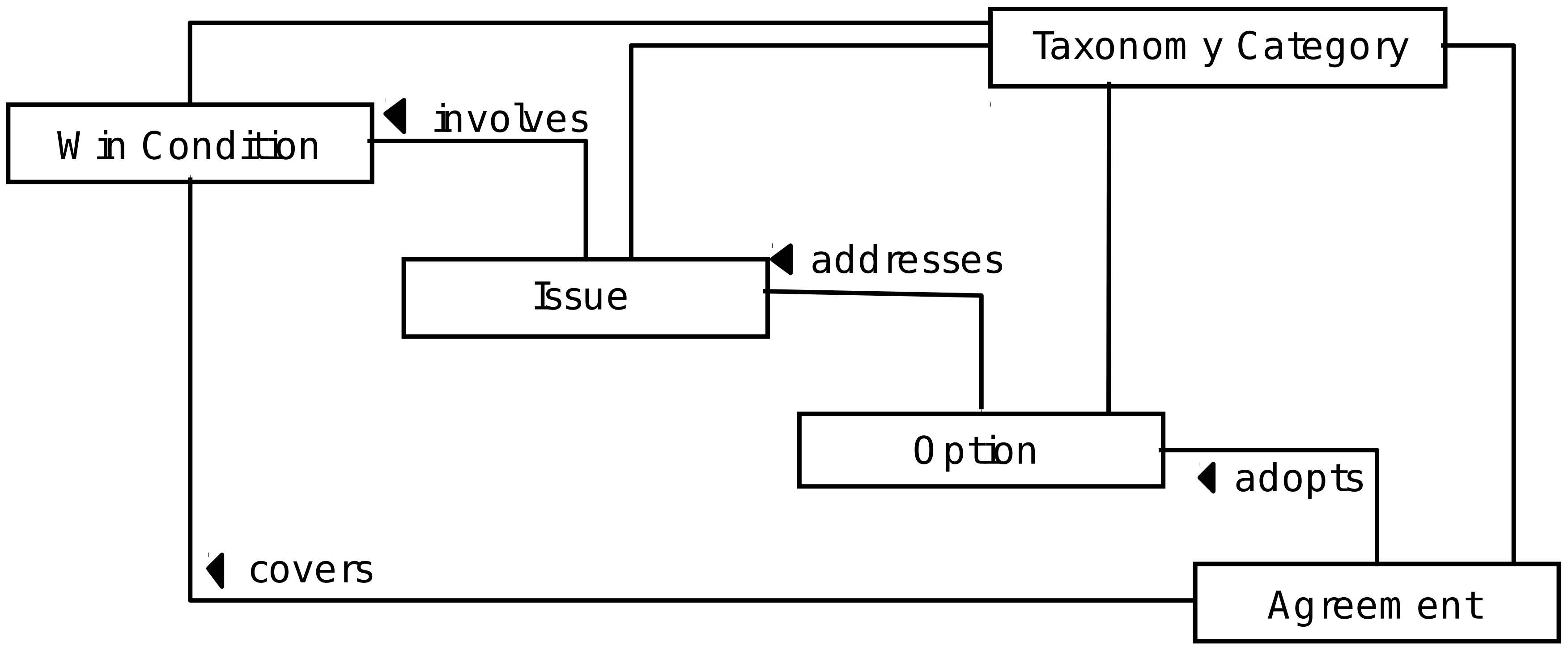
. . P: NotificationService is not part of access (Ed Jones 28.06)

. . . . pro: Dispatchers can see all TrackSections (Ed Jones 28.06)

. . . . pro: Simplicity (Ed Jones 28.06)

28.06.99 (Open) I: Can a dispatcher modify another dispatchers' TrackSections? (Alice Parker)

Figure 12-27, WinWin Issue Model


Taxonomy Category
Win Condition
involves
Issue
addresses
Option
adopts
covers
Agreement