ISAC
User Requirements Document
Software Requirements Document
Architectural Design Document
Software Design Document
Use Cases, Test Cases

The *ISAC*-Team
`isac@ist.tugraz.at`
`www.ist.tugraz.at/projects/isac`

Revision

Contents

List of Figures

Part I

User Requirements Document

This document describes the user requirements for the $\mathcal{ISAC}$-system.

By now the document captures at least those requirements in detail, which are a prerequisite for the *first phase of development, i.e. the development of the kernel of the math knowledgebase, of the indispensible tools for interaction on the knowledge, and of the tools for authoring the example collection.*

The design will try to meet these requirements while accepting the structure of the $\mathcal{ISAC}$-math-engine, which is defined by mathematical reasons, and offers new functionality (calculations are done stepwise, the learner can input a formula or a tactic and receive feedback from the engine, the math-engine usually 'knows' the next step).

The User Requirements Document is structured along the different kinds of users envisaged. The Software Requirements Document, on the other hand, will be structured along the modules implementing the functionality. In order to establish comfortable tracing, the $m : n$ relation between user requirements and software requirements will be accurrately recorded in a double-linked way.

Terms marked by $\rightarrow$ are briefly described in appendix D.

Chapter 1

Kinds of $\mathcal{ISAC}$ users

There are several kinds of $\mathcal{ISAC}$ users which set the respective sections of requirements:

visitor (Besucher): occasionally drops into an $\mathcal{ISAC}$-site and browses the respective math knowledge base and the example collection. May try to calculate some examples. Or some scientific content provider (wiki, scientific branding in an institution or an enterprise) includes occasional $\mathcal{ISAC}$-services, in particular the possibility to interactively explore some specific calculations.

learner (Lernender): uses $\mathcal{ISAC}$ for learning and exercising, i.e. primarily calculates examples in the example collection by use of the math knowledge base. As a member of courses the learner is called a student.

math author (Mathematik-Autor): is an expert in computer mathematics who adapts and extends the mathematics knowledge base.

dialog author (Dialog-Autor): is an expert in learning theory who adapts and extends the dialog guide.

course designer (Kurs-Designer): adapts and extends the example collection which can be solved by a given math knowledge base, and adds explanations to items in the knowledge base. These tasks do not require special knowledge in computer mathematics.

course admin (Kurs-Administrator): is a person administering the use of $\mathcal{ISAC}$ for learning within a group of learners. This person is also regarded as a legal representative of the institution consuming $\mathcal{ISAC}$ services.

administrator (Administrator): this róle combines the system administrator installing the software, and the person who implements the overall design of an $\mathcal{ISAC}$-site (introductory pages etc.). This person is also regarded as a legal representative of the institution hosting $\mathcal{ISAC}$ services.

These kinds of users are distinguished by the respective access-rights.

Table 1.1: Survey on the roles of $\mathcal{ISAC}$ users

task	math author	dialog author	course designer	course admin
determine interaction in an expl.	by theories problems methods	by impl. of dialog patterns	by impl. of examples, of explanations to knowl.items	by setting content delivery, assessments

Chapter 2

General requirements

This section describes the requirements common to all users.

2.1 Users of *ISAC*

UR 2.1.1 ***ISAC is a multi-user system.***

UR 2.1.2 ***The users access ISAC via internet.***
The computing resources needed to run such a complex application exceed the computing-resources presently available to the average user. Moreover, organisation of centralised courses and curricula suggests separation of application and user-interface. An additional requirement is to keep expenses and effort for the average user at a minimum, in terms of computing power needed and installation effort. (Ideally, a standard web browser would suffice, but interaction required for the worksheet cannot be managed by a browser yet.)

UR 2.1.3 ***ISAC's data storage supports simultaneous access***
Data storage has to support locking and versioning.

UR 2.1.4 ***Users access ISAC with different roles***
Several possible roles when accessing *ISAC* dictate different rights and access to different modules of *ISAC* see p.13.

UR 2.1.5 ***Learners can be grouped into courses.***
There are groups of learners in order to support the adminstration of courses. The membership w.r.t. these groups determines the selections of examples in the example collecton (see UR.8.7.1), the selection of explanations in the knowledge (see UR.8.6.3)) and the initial setting of the dialog as captured in UR.4.4.11 and UR.4.2.8.

UR 2.1.6 ***One learner may be member in different groups***
One learner may be member in different groups but the settings for a session depend on exactly one group. This implies that multiple memberships have to be resolved at login time.

2.2 Calculations and Data Involved

UR 2.2.1 ***ISAC's math engine is given as an already-implemented module.***
ISACs mathematics engine(ME) is already given. Thus several requirements, looking like software requirements, are listed here. A calculation (see terms in the ISAC-project [iT02b]) undergoes three phases: the modelling phase, the specification phase and the solving phase, where the latter may contain these phases recursively (see the demonstration example in the usecases document [iT02c]).

Calculations are created interactively in steps. A step is initiated by the input of a learner: input of a tactic, of a formula or of a 'go-on' command. A tactic is applied to a formula and generates another (the derived) formula. The tactics specified so far are listed in appendix E An input formula during modelling phase completes a model, and during the solving phase an input formula is considered a derivation of the previous formula.

UR 2.2.2 ***ISAC stores 4 kinds of Mathematical Knowledge***

Theories

Problems

Methods

Examples

See [Neu01] for a detailed explanation.

UR 2.2.3 ***A Calculation undergoes 2 Phases***
A calculation undergoes two phases: the Specification Phase and the Solving Phase, where the latter may contain these phases recursively (see the demonstration example in appendix A).

UR 2.2.4 ***Specifying constructs a Model and a Specification.***
See UR.2.2.5 and UR.2.2.6.

UR 2.2.5 ***A Model consists of fields and items***
A Model consists of the fields 'given' containing the input-items, 'where' containing the pre-condition on the input-items, 'find' containing the output-items and 'relate' containing parts of the post-condition.

UR 2.2.6 ***A Specification consists of theory, Problem, Method***
Theory, Problem and Method provide for 3 pointers into the knowledge base (see Fig.29.1 on p.186) referencing the knowledge required to solve the example specified by a model.

UR 2.2.7 ***Solutions are calculated interactively in steps.***
Stepwise calculation is done in the Solving Phase. A step is initiated by the input of a learner: input of a tactic, of a formula or of a request that *ISAC* take over the calculation. A tactic is applied to a formula and generates another (the derived) formula.

UR 2.2.8 ***In the Solving Phase, every formula is justified by a tactic.***

UR 2.2.9 *ISAC* ***uses tactics as listed in appendix E***

UR 2.2.10 ***A calculation has a tree-like structure***

UR 2.2.11 ***A Tactic may contain Error Schemes***

UR 2.2.12 ***There are fill-in patterns for Tactics.*** In addition to a Tactic being applied manually by the user or automatically by *ISAC*, a Tactic can be presented with parts left blank to be filled in by the user.

UR 2.2.13 ***There are fill-in patterns for items of the Model.***

UR 2.2.14 *ISAC* ***guarantees correct results.***
ISAC's display of a Calculation reflects the state of the Math Engine. The Math Engine does not produce any inconsistent state of calculation. Thus everything displayed by *ISAC* (apart from lines being edited by the user right now) is proven to be consistent with the Specification and the Knowledge Base. User input will be accepted only if it can be proven to be correct by a check with the underlying Math Engine.

UR 2.2.15 ***The the mode of the ME*** is *one* of the following:

> step is applicable, result guaranteed
> step is applicable, result is not guaranteed
> (because the learner has input some strange step previously)
> the ME is helpless, i.e. it cannot propose a next step
> the step is not applicable, plus an error message, why not applicable.

UR 2.2.16 ***Consistent "look & feel" for all users.***
As long as the deductive part is left to Isabelle, the same is with the theory browser. *ISAC* will develop the "look & feel" of its own, and thus violate uniformity w.r.t. the theory browser.

2.3 Miscellaneous

UR 2.3.1 *ISAC* ***supports internationalization.***
The mother language should be used by the student. The language of math formulas is independent from other languages. Thus the names used in the knowledge base can simply be changed for each language (scriptures with different structure like Japanese will not be considered here). But there are other texts delivered by the system, like error messages, or the fields 'given,...' in a model; these should be implemented multilinqual.

UR 2.3.2 ***ISAC* supports cross-linking calculations and knowledge**
Calculations may contain links into the Knowledge Base, e.g. to the definitions or proofs of tactics applied in the course of calculation or into underlying theories. The Knowledge Base may contain links to examples illustrating theoretical concepts. More specific requirements are in sect.4.4.

UR 2.3.3 ***ISAC* is open to data exchange with other tools.**
To facilitate interfacing with other tools, *ISAC*'s objects support being exported to and imported from open data formats. XML formats are preferred.

UR 2.3.4 ***Several users can watch the progress of one calculation.***
Several users may watch the progress of a calculation, but there is exactly one user controlling the calculation and taking actions. This can be useful for instruction situations, especially teleteaching.

UR 2.3.5 ***A calculation can be edited with other tools*** Editing in *ISAC* is limited by UR.2.3.6. For publishing purposes, *ISAC*'s calculations can be exported for editing with standard publishing tools.

Guarantee of correct results is given by *ISAC*s calc-state in the ME. This guarantee can only be given, when each editing on the worksheet is mirrored in the calc-tree (eventually cutting certain branches, if an intermediate step is redone). Sometimes the learner might want to edit a calulation (shorten the calculation by cutting intermediate steps, etc.) without affecting the calc-tree any more.

UR 2.3.6 ***ISAC* guarantees correct results.** In this case the worksheet reflects the calc-state, which shall be indicated on the screen and on the printout; in the latter case this indication should be certificate which cannot be manipulated.

Chapter 3

Requirements of the visitor

Each user approaching an *ISAC*-site first time wants to know about the purpose of *ISAC* and about the contents of the site.

3.1 Browsers: overview – detail:

The contents of the knowledgebase and of the example collection are expected to become very large. Thus there need to be facilities to switch from an overview to a detail and vice versa. These facilities should be handled similarly for all the browsers. Primarily, there is no direct interaction between the visitor and the ME. Information for the visitor should be provided in statical HTML-Pages.

Visitors should get an overview over *all* the knowledge available at an *ISAC*-site, and *all* the explanations, and *all* the examples. Therefore, access to additional information should be gained if available.

UR 3.1.1 ***There are 4 kinds of data to be browsed: theories, problems, methods, examples.***

UR 3.1.2 ***All 4 kinds of data need a table of contents of variable detail.***

UR 3.1.3 ***Browse through statical HTML-Pages*** without e.g. matching a model with a problem.

3.2 Do some examples:

Visitors should get an overview over *all* the features available at an *ISAC*-site, and the most exciting feature is stepwise interactive calculating and reasoning guided by the system. This should be demonstrated by some examples, which also the visitor can execute.

However, the possibility for any visitor to start an interactive calculation, i.e. a Worksheet from a browser would have resulted in a too complicated and unstable system. A respective architecture see [Gri03]. Now a visitor has to choices:

UR 3.2.1 ***A visitor can view a calculation.***
These calculations are displayed a a whole, with the possibility of folding and unfolding the hierarchical structure. There is no possibility of interactive stepwise construction of the calculation as described in sect.4.

UR 3.2.2 ***A visitor can download and run an interactive Worksheet.***
After downloading the launched front-end (containing the Worksheet) there is a possibility to connect to a server running the $\mathcal{ISAC}$ back-end performing the calculation and user guidance.

Chapter 4

Requirements of the learner

[1]

4.1 Start a calculation

There are two different ways for users to approach *ISAC*s facilities for learning: (a) the user may browse the knowledge base, and eventually calculate an example (illustrating a definition, a problem, etc.) and (b) the user calculates examples from the example collection, and while asking for justifications of steps in the calculation enters knowledge browsers.

UR 4.1.1 ***A calculation can be started for a pre-defined example from an example collection***

UR 4.1.2 ***A calculation can be started for a pre-defined example from the Knowledge Base*** This is to illustrate knowledge from the Knowledge Base by typical examples stored with the knowledge.

UR 4.1.3 ***A calculation can be started from scratch.*** In this case, the calculation must start with specifying, and *ISAC*'s support is limited as described in UR.4.2.2.

UR 4.1.4 ***A calculation can start with specifying or solving.*** In order to emphasize exercising specifying *or* solving, which are very different tasks.

UR 4.1.5 ***A calculation can be done like in an algebra system*** by simple input of a function call like `solve`, `simplify`, `integrate` or the like. In this case the specifying phase is skipped.

---

[1] Begin of copy from [Kre05] p.36-38

4.2 User Guidance

UR 4.2.1 ***ISAC provides User Guidance if the problem to be solved has been specified.*** A problem known to the system implies that the system knows a method to solve the problem. Thus *ISAC* can solve the problem automatically or propose the next step to be done.

UR 4.2.2 ***ISAC can assist in calculating examples unknown to the system*** Without knowing the Problem, *ISAC* cannot propose steps or solve the Problem automatically. Still, *ISAC* can apply Tactics chosen by the user to a formula. With a theory specified, *ISAC* can check formulas input by the user for correctness and consistency with previous steps in the calculation. At all times, *ISAC* ensures the calculation displayed is consistent and error-free as detailed in 2.3.6.

UR 4.2.3 ***ISAC can offer a list of actions meaningful in the current state*** of the calculation. 'Meaningful' is weaker than 'applicable', see UR.4.2.4

UR 4.2.4 ***ISAC can offer a list of actions applicable to the current state*** of the calculation.

UR 4.2.5 ***ISAC can propose the next action to be taken.***

UR 4.2.6 ***ISAC can do one or more steps automatically.***

The flow of interaction shall be adapted to the learner. The learner might be bored because the system offers too little challenge (by using less active dialog atoms – see UR.6.2.1, or by proceeding in too little steps of calculation) – or, in contrary, the user might be frustrated by too high challenges (in activity and/or stepwidth).

UR 4.2.7 ***The 'activity' of the dialog atoms adapts to the learner.***

UR 4.2.8 ***Varying stepwidth with tactics, rule sets and subproblems.***

The dialog regards presettings of the course admin (see Sect.7 below), when guiding the flow of interaction, and the dialog regards the ongoing interaction with the student.

UR 4.2.9 ***The system records examples done/not done by a learner*** regardless which course the learner is logged in.

UR 4.2.10 ***The dialog regards the performance*** in calculations done by the learner in the current session. The performance is measured by response times, errors, difficulty of examples done, requests into the knowledge base, active-passive behaviour. [2]

---

[2] The completion of this list is up to a future phase of development, and the evaluation of these data as well.

UR 4.2.11 ***The dialog regards the knowledge*** touched by the learner in the current session.

UR 4.2.12 ***The dialog regards the history*** of performance and knowledge touched in previous sessions.

UR 4.2.13 ***The amount of user guidance is configurable.*** The amount can be set by the user according to his preferences or by a course designer to match requirements of the course. For exam purposes, the amount of user guidance can be limited.

UR 4.2.14 ***The learner can override the Dialog behaviour chosen by the system.*** This includes the settings for dialog activity, stepwidth and display filtering rules. The conflicts between this UR and UR.4.2.13 need to be resolved by an indepth design lateron.

4.3 Help in the specify phase

According to UR.2.2.3 this phase comprises modeling (i.e. translating an example into a formal representation, the so-called Model) and specifying (i.e. relating the model to the mathematics knowledge available — that means, identify an appropriate theory, a problem and a method).

In the model phase the learner generally has to input the →items of a →model (see the example for reference in chapter A.1 p.219): input and output items can be 'correct', 'incomplete', 'missing', 'superfluous' or have a 'syntax error'; preconditions can be 'correct' or 'false' (see UR.A.3) — these properties should be clearly indicated. This help is only possible, if the learner choose an example (with a hidden →formalization and →specification) from an →example collection, or if he had specified a problem.

If users want to calculate an example *unknown* to the system they have two choices:

1. specify a problem in order to get help from the system. Again, there are two possibilities:
    (a) first specify the appropriate problem (which contains the →item-descriptions), and then input the items. The item-descriptions help the user to provide the appropriate item-data.
    (b) first input the items (the item-descriptions are to be looked up in a specific theory), and then specify the problem — already assisted by the system (see UR.4.3.2 and UR.4.3.3).
2. do the calculation independently and without relying on assistance by the system (i.e. without specifying a problem, thus the system cannot check for completeness, and cannot assign a →method). In this case the input of items in the 'given'-field is sufficient; the users can calculate the example by applying tactics (after manual input).

UR 4.3.1 ***Visualization of the feedback on input to a model.*** The user gets immediate feedback on data entered into the Model by means of an item-status. The item-status can be: 'correct', 'incomplete', 'missing', 'superfluous', 'syntax error' or 'false'.

Retrieve a matching problem: Given a model initiating a calculation, or as a subproblem within a calculation, the learner has to determine a problem matching this model. This involves information retrieval from a large problem-hierarchy. There the learner may get lost, and thus he should get help: make the math-engine find a matching problem; visualize the problem found within the hierarchy.

In case of an exam, the user is forced to find the correct problem with no or limited help. Therefore it is nessesary to adjust the amount of help given by the browsers (e.g. only "match" or "nomatch" instead of a detailed listing of all conditions)

UR 4.3.2 *ISAC* ***can retrieve and match a model to a problem.***

UR 4.3.3 *ISAC* ***can help by automated refinement of a problem.***

4.4 Views into the knowledge

ISAC is a 'transparent system' by providing access to all the knowledge the system requires for (automated) problem solving. The knowledge is structured within a 3-dimension universe as stated in UR.2.2.2 on p.16. Needless to say that the knowledge has a highly complex structure, and that it is a demanding challenge to help the student not to get lost in this structure.

Principally, there are two ways to acces the knowledge, (1) access along the structure inherent to the respective knowledge, and (2) access from a concrete calculation, where the knowledge is required for solving the problem. These two kinds of access are separated below.

4.4.1 Surveys on related knowledge

As example collections (respective requirements see sect.8.2, sect.8.3 and sect.8.4) are very close to the other three parts of knowledge, theories, problems and methods, we include them in this subsection.

UR 4.4.1 ***Each element of the knowledge belongs to either theories, problems or a methods, or to the example collection.*** The elements of belonging to theories are

- theorems
- rule sets, i.e. sets of theorems or other rule sets, which are applied as long as they can be applied to a certain formula.

- rewrite orders which are required to terminate rule sets which contain theorems on commutativity etc. TODO
- computations involving sml-code (the only exception to rewriting). TODO

UR 4.4.2 ***Examples, theories, problems and methods all have a hierarchical structure*** and each element of the knowledge base has a unique position in this hierarchy.

UR 4.4.3 ***Each element of the knowledge base is displayed in the related browser.*** This requirement has to be met in particular, if such an element is referenced by a link from another browser: where-ever such a link is located, the element is displayed in the browser it belongs to, a theory-element in the theory-browser etc.

UR 4.4.4 ***Links can go from any element to any element.*** That means, an explanation for a problem can have a link to a method solving it, or to a theorem (in a theory) important for this problem; an explanation for a theorem can have a link to a problem, which uses the theorem in a certain way, etc.

UR 4.4.5 ***There are specific links which start an example.*** Such a link may be located in any part of the knowledge; if the link is activated, the respective example is displayed in the example-browser (UR.4.4.3) together with the examples location in the example hierarchy (UR.4.4.7) **and** a worksheet is opened for this example.

UR 4.4.6 ***Links outside ISAC's knowledge base open the standard browser.*** i.e. links within the knowledge base may point anywhere, and if the destination is outside ISAC's knowledge, it is displayed outside ISAC, too.

UR 4.4.7 ***An element is always displayed togehter with the respective location in the hierarchy.***. This requirement has to be met in particular, if the element is displayed following a link (indenpendently, from which browser).

UR 4.4.8 ***Each element has a certain position in the respective hierarchy.*** The hierarchy is the means for systematic search by the user. This does *not* mean, that this position can serve as a unique identification over time, see UR.8.6.4 !

UR 4.4.9 ***With each element in the knowledge base, explanations can be stored.*** Every element in the Knowledge Base can provide explanations illustrating its meaning, giving theoretical background information, referencing related topics or giving examples of use. See sect.7.1 p.35 and also UR.8.6.2.

4.4.2 Context-related views

The context is given by the state of a calculation (the so-called 'calc-state') on a Worksheet. UR.4.3.2 and UR.4.3.3 is one case, the other is displaying a method with the tactic marked which just has been applied in a calculation (this is well-known from debuggers). And there are other dynamic views into the knowledge base.

Context to all parts of the math-knowledge

UR 4.4.10 ***ISAC's use of math knowledge can be watched by the user.*** The user can look up the elements of the Knowledge Base currently used by *ISAC* to do a certain step in a calculation. This includes showing

- Tactics used in the calculation in their context in the Knowledge Base; particularly interesting are rewrite-tactics. The context to the calculation allows to display the rewrite, the assumptions generated, the rewrite-order used etc.
- Methods being applied to solve the current problem with indication of the tactic being currently applied, of the method's guard etc.
- Problems currently being solved in their context in the hierarchy of problems. This particularily helpful just before and after having *ISAC* refine a problem.

UR 4.4.11 ***On request, ISAC provides additional information on parts of the calculation.***
Additional information in a calculation can be provided at any time on request of the learner. This feature comprises more detailed views onto the calc-state, as well as explanations according to the following table:

`'detail'` on	element	yields
whole formula	//	intermediate steps
whole formula	//	tactic applied or applicable
whole formula	//	applicable tactics
whole formula	//	associated assumptions
whole formula	//	accumulated assumptions
formula	function-constant	definition in the theory
formula	floatingpoint-no	precision of this no
evaluated predicate	//	derivation
evaluated assumption	//	derivation
tactic	theorem	theorem instantiated
theorem instantiated	//	animation of matching
`'detail'` on	calchead	yields
specification	theory	file of the respective theory
specification	problem	model instant. this problem
specification	problem	inst. problem in the hierarchy
specification	method	guard instantiated by the model
specification	method	script in the hierarchy

UR 4.4.12 ***The context is a certain formula in a certain calculation*** as displayed on the respective 'worksheet'.

###################### WN060704 end update! ######################
[3]

---

[3] Begin of work extracted from [Kom07].

UR 4.4.13 ***A context is the current position on the worksheet last touched.*** The formula at this position is highlighted.

UR 4.4.14 ***Usually there is a context to any of the 3 parts of the math-knowledge for a position. See also UR.4.4.15 and UR.4.4.16***

UR 4.4.15 ***If there is no Worksheet open, then there is no context.***

UR 4.4.16 ***There may be NO context for an element of the math-knowledge.*** The reasons are specific to theories, problems and methods.

UR 4.4.17 ***The default for <Context on/off> is <on>*** - as soon as a Worksheet has been opened.

UR 4.4.18 ***If the KnowledgeBrowser is opened the first time and there is no context, it displays the element at the root of the respective hierarchy.***.

UR 4.4.19 ***Any formula in a calculation has a context (with different details to different parts of the math-knowledge).***

UR 4.4.20 ***The user can switch the context to a calculation on or off.***

UR 4.4.21 ***The context of an element is displayed in the respective browser-window.*** This is analoguous to UR.4.4.1

UR 4.4.22 ***A context is displayed as soon as*** the window is activated or the respective button <theory> <problem> <method> is pushed – if the user has NOT decided for .

UR 4.4.23 ***The contents of the hierarchy can be filtered (due to a UserModel).***

UR 4.4.24 ***The content of the hierarchy remains unchanged during a session (SR: thus the hierarchy is loaded*** **once** ***at the beginning of a session)***

Context to examples

UR 4.4.25 ***The context of an example is the respective worksheet.*** If <Context on> is set and a worksheet is brought to top, the respective example (i.e. the description of the example) is displayed in the example browser.

UR 4.4.26 ***A worksheet started by <New> has no context.*** It is handled according to UR.4.4.18

UR 4.4.27 ***<Context on/off> is the only choice for examples.*** This choice is always available. It is never changed by the system.

UR 4.4.28 ***A example may be displayed and*** **not** ***be allowed to calculate (according to the UserModel).***

Context to elements of theories

UR 4.4.29 ***The context comes exactly from the current position on the worksheet.***

UR 4.4.30 ***<Context on/off> is always available.***

UR 4.4.31 ***<To Worksheet> is available if:***

1. <Context on> is selected *and*
2. the worksheet on top is in the specify phase (and a CalcheadPanel is open)

UR 4.4.32 ***<To Worksheet> is* not *available:***

1. if <Context off> is selected, as described in UR.4.4.36
2. if no worksheet is open
3. if the worksheet on top is in the solve phase (and *no* CalcheadPanel is open)

Context to problems

UR 4.4.33 ***The context of a formula to the problems is given by the headline of the calc-head on the next higher level in the calculation.***

UR 4.4.34 ***The context of problem concerns the model of the current position and the modelpattern of a problem.*** Thus there is always a context, only exception is UR.4.4.18

UR 4.4.35 ***<Context on/off> is always available.***

UR 4.4.36 ***If <Context off> is selected, then <Refine> and <To Worksheet> are not available.***

UR 4.4.37 ***<Refine> is available if <Context on> is selected.***

UR 4.4.38 ***<Refine> is* not *available:***

1. if <Context off> is selected, as described in UR.4.4.36
2. if no worksheet is open

UR 4.4.39 ***<To Worksheet> is available if:***

1. <Context on> is selected *and*
2. the worksheet on top is in the specify phase (and a CalcheadPanel is open)

UR 4.4.40 ***<To Worksheet> is* not *available:***

1. if <Context off> is selected, as described in UR.4.4.36
2. if no worksheet is open
3. if the worksheet on top is in the solve phase (and *no* CalcheadPanel is open)

Context to methods

UR 4.4.41 ***The context of a formula to the methods is given by the headline of the calc-head on the next higher level in the calculation.***

UR 4.4.42 ***The context of a method concerns the guard and the script.*** The script in context show the tactic which calculated the currents position; the guard is matched with the model of the problem at the current position.

UR 4.4.43 ***<Context on/off> is always available.***

UR 4.4.44 ***<To Worksheet> is available if:***

1. <Context on> is selected *and*
2. the worksheet on top is in the specify phase (and a CalcheadPanel is open)

UR 4.4.45 ***<To Worksheet> is* not *available:***

1. if <Context off> is selected, as described in UR.4.4.36
2. if no worksheet is open
3. if the worksheet on top is in the solve phase (and *no* CalcheadPanel is open)

[4]

---

[4] End of work extracted from [Kom07].

Chapter 5

Requirements of the math author

All the authoring of math knowledge is still be done on the SML toplevel, i.e. immediately on the datastructures holding the knowledge. This part of the task is described in the 'interfaces for authors of math knowledge' [iT02d].

UR 5.0.46 ***Remote access to the sml-kernel*** is required. However, knowledge modfied need not yet imported to the production (i.e. tutoring) system.

Another result of authoring math knowledge, however, will be tools for the visitor and the learner to view the knowledge generated. This part of the task is described here.

UR 5.0.47 ***Automatic linking-tool*** supports setting links to other occurrences of an item: the system suggests links, the author accepts or rejects. Automated linking is done between the following items:

item	in ...	linked to occurrence in
predicate	theorem	definition in theory
	pre/postcondition	definition in theory
theorem	tactic	definition in theory
TODO [1]		

UR 5.0.48 ***Exchange data with other ISAC sites.***
TODO

UR 5.0.49 ***Exchange data with other knowledge bases.***
TODO

Copyright is important as *ISAC*'s development will depend on the efforts of many, many authors.

UR 5.0.50 ***Copyright on any substantial item or part of math knowledge.*** Such items are

- a theory, and within theories
    - a rule set
- a problem
- method

See UR.8.1.3.

UR 5.0.51 ***An item can have more than author*** according to the *ISAC*-charta.

Authoring theories

The generation of the theory browser is already implemented by Isabelle. Within phase 1 of development, *ISAC* will take this component without any change.

Authoring problems

TODO

Authoring methods

TODO

Chapter 6

Requirements of the dialog author

The dialog author focuses on learning theory; administrative aspects of dialogs are discussed in chap.7 for the course admin.
[1]

6.1 User Profiling

UR 6.1.1 ***ISAC records examples done by the user*** ISAC keeps a per-user record of examples done and the user's performance in doing the example. The record is independent of the course the user has been logged into when doing the example.

UR 6.1.2 ***ISAC records items in the Knowledge Base viewed by the user.*** This information can be used to base the Dialog Guide's behaviour on information supposedly known to the user.

UR 6.1.3 ***ISAC records the user's success and errors.*** This extends to application of single Tactics, fill-in patters and error-patterns as well as whole examples or courses.

UR 6.1.4 ***ISAC records the user's time performance.*** In the future, assumptions about the user's familiarity with certain topics could be derived from these data.

UR 6.1.5 ***ISAC records the user's activity.*** In this context, activity means the ratio of steps done by the user to the steps the user had done by ISAC.

---

[1] Begin of copy from [Kre05] p.38-40

6.2 Flexible Dialog Behaviour

UR 6.2.1 ***ISAC's Dialog behaviour is constructed from Dialog Atoms*** We hope that it is possible to develop a language which allows to define Dialog Patterns as combinations of Dialog Atoms already implemented and Dialog Strategies sequencing these atoms. By means of such a language learning strategies could be described, and this description could be interpreted in reaction to a dynamic dialog state and according to a knowledge profile.

To do such 'dialog programming' is considered a comprehensive task, which in general exceeds the knowledge of a course designer or a course admin. On the other hand, a course admin can be expected to associate courses with dialog profiles, and a course designer can be expected to select Dialog Strategies within process of time in a course.

UR 6.2.2 ***The Dialog's behaviour can be configured.*** The Dialog's behaviour in terms of Dialog Atoms to be used can be preset by the course designer and the user to match the requirements of the situation in style and complexity. The probability of asking the user a question is an example of such a preset.

UR 6.2.3 ***The number of calculation steps taken at a time can be configured.*** This extends to taking several steps at a time and doing whole rulesets or subproblems in one step.

UR 6.2.4 ***The amount of information displayed can be configured.*** As with the Dialog's behaviour, the amount of information can be preset to meet the requirements of the learning situation. This could mean displaying or not displaying the Tactics used in the calculation or hiding specific steps in the calculation considered too complex or too trivial.

6.3 Adaptation to Individual Users

UR 6.3.1 ***The activity of the Dialog Guide adapts to the learner.*** In a learning situation, active participation of the student is one key to acquiring and consolidating knowledge and skills. The Dialog Guide will support this by asking questions or letting the user decide what to do in the next step. On the other hand, with growing expertise, once thrilling questions become trivial and boring. The Dialog Guide adapts his strategy by dropping challenges the user has alredy mastered a few times. Whenever possible, the dialog adapts its behaviour, i.e. the choice of Dialog Atoms, to challenge the user without frustrating him. The choice is based on the present and past actions of the individual user.

UR 6.3.2 ***The Dialog adapts the amount of information displayed to the learning situation.*** As with the Dialog's behaviour, the amount of information displayed adapts to the requirements of the current situation.

UR 6.3.3 ***The Dialog regards the performance of the user.*** The performance is measured by response times, errors, difficulty of examples done, requests into the Knowledge Base and Dialog Activity.

UR 6.3.4 ***The Dialog regards the knowledge touched by the user in the current session.***

UR 6.3.5 ***The Dialog regards the history of the user.*** In addition to the of performance and knowledge touched in the current session, the history of the user's previous sessions is regarded as well.
[2]

---

[2] End of copy from [Kre05] p.38-40.

Chapter 7

Requirements of the course admin

Authoring in $\mathcal{ISAC}$ comprises various tasks: authoring mathematics knowledge and authoring the dialog have been described in Chapt.5 and Chapt.6; these both tasks require more special knowledge than the others. Chapt..8 describes a kind of knowledge ('explanations') which may change from one course to the other, and wich does not require special knowledge on computer mathematics or dialog design.

The latter part of authoring is done by the course designer preparing as course in advance (see Chapt.8). Some of the knowledge prepared in advance underlies time constraints, which are managed by the course admin. The requirements of the course admin are separated in this section (despite the fact that the course designer and the course admin are one and the same person, the lecturer or teacher of the course).

7.1 Groups of learners

There are groups of learners in order to support the adminstration of courses. The membership w.r.t. these groups determines the selections of examples in the example collecton (see UR.8.7.1), the selection of explanations in the knowledge (see UR.8.6.3)) and the initial setting of the dialog as captured in UR.6.2.4 and UR.4.2.8.

UR 7.1.1 ***Learners can be grouped into courses.*** There are groups of learners in order to support the adminstration of courses. The membership in these groups determines the selections of examples in the example collecton (see UR.8.7.1), the selection of explanations in the knowledge (see UR.8.6.3)) and the initial setting of the dialog as captured in UR.4.4.11 and UR.4.2.8.

UR 7.1.2 ***One learner may belong to different groups*** but only to *one* group within a session. For instance, a student of TUG can be member of

Analysis semester 1, of Signal Processing semester 5, Analysis semester 2.

UR 7.1.3 ***Administrative information for groups*** is part of any login, and should be available to the course admin for review at any time during a session. [1]

[2]

7.2 Restrictions

Restrictions will apply when using $\mathcal{ISAC}$ as a learning or tutoring tool, especially during exams. When being used as a calculation tool, restrictions are relaxed to gain access to the full power of the Math Engine.

UR 7.2.1 ***Restrictions are individual to a user or group.***

UR 7.2.2 ***Groups of examples may be invisible.*** Parts of the example collection may be inaccessible for specific groups of learners.

UR 7.2.3 ***Access to items in the Knowledge Base can be restricted.***

UR 7.2.4 ***The amount of User Guidance may be restricted.***

UR 7.2.5 ***The use of Dialog Patterns may be restricted.***

UR 7.2.6 ***Restrictions may be overridden.*** Depending on the settings provided by the course-designer and the user's access rights, some restriction may be overridden by the user. Overriding e.g. allows to look at explanations and examples for other courses.

UR 7.2.7 ***Restrictions may apply within time limits.*** Some restrictions may apply only within certain time limits, e.g. during an exam or during class. Time limits can be given by start and finish or by their duration.

UR 7.2.8 ***Restrictions may depend on the user's learning progress.*** Some restrictions may apply until certain examples have been solved or the user has mastered certain aspects of knowledge.

[3]

Time limits for delivering course material are a general requirement for educational systems, and it applies for example collections as well. This requirement provides the course admin to distribute workload over time and to focus the attention of the learners. Sometimes it may be desirable to have some examples finished within a certain time limit. Also examples (for instance prepared for an examination) may be invisible for learners.

---

[1] End of copy from [Kre05] p.38-40

[2] Begin of copy from [Kre05] p.41

[3] End of copy from [Kre05] p.41

UR 7.2.9 ***There are 2 kinds of time limits:*** (1) given by start and finish, and (2) given by a duration (where start and finish are recorded with the user).

UR 7.2.10 ***Groups of examples may be locked*** for groups of learners for certain time limits. (SR: attention with links from the KB !)

UR 7.2.11 ***Groups of examples may be invisible*** for all users except the example author for certain time limits. (SR: examples have an author !)

UR 7.2.12 ***Restricted help for the learners during a exam*** i.e. particular links into the knowledge base are restricted for a particular group for a limited time.

UR 7.2.13 ***Access-rights during an exam-session*** are particularily restricted (a user might get unwanted information by opening another session as a member of another group).

UR 7.2.14 ***Restrictions may be overridden or not*** depending on the setting by the course-designer. Overriding e.g. allows to look at explanations and examples for other courses.

7.3 Survey on the progress of the learners

i.e. of the students in a course. Such surveys are indispensable requirements for a course admin. Thus all the surveys below are related to a specific course.

UR 7.3.1 ***There is a statistics-tool*** for the learners progress

UR 7.3.2 ***Statistics are drawn from*** the following data: TODO
Examples for statistics:

- Which examples have been done by whom with which performance
- Sorts w.r.t. the examples: frequency of touched, not touched, solved, unsolved (, evaluation of performance, as soon as evaluation functions have been designed) over all students of a course
- Sort w.r.t. the students: likewise over all examples in specified groups
- Which lookups into the knowledge have been done
- Sorts w.r.t. the destination: likewise over students
- Sorts w.r.t. the students: likewise over the knowledge.

For preliminarily checking the usage of the system (as well as for preliminary monitoring of the system – see sect.9.1) the following requirement is stated, until the respective requirements analyses are done.

UR 7.3.3 ***A text-file records the usage of the system.*** The records contain the following basic data for each user logging in during the life-time of the session-manager.

- at login
  - user name (if there are anonymous users, record the mail address)
  - time of login
- for each example called (no recording of access to the knowledge base !)
  - example-ID
  - time of starting the example (i.e. the respective worksheet)
  - time of removing the example (i.e. the respective worksheet)
- at logout
  - user name (if there are anonymous users, record the mail address)
  - number of examples
  - time of login and logout

Special requirements may be raised by field studies on learning mathematics. $\mathcal{ISAC}$ shall be open for such special research. I.e. queries over examples, lookups, students *and courses*. Anonymous evaluation has to be regarded.

UR 7.3.4 ***Anonymous evaluation of statistics.***

7.4 Written examinations

can be done due to requirements stated elsewhere: hidden examples UR.7.2.11, restricted access to knowledge UR.7.2.12, and adapted dialog UR.6.2.1.

UR 7.4.1 ***The course admin can force a student to exam-mode:*** That includes, that there is no way for this user, to open a session with a user-model which would undermine the exam-mode. A student, in spite of being a member of a group (or several groups), might be examined separated from other members of the group (in case of illness at the time of the group's exam etc.)

UR 7.4.2 ***The course admin can force a group of users to exam-mode:*** Sideconditions as in UR.7.4.1.

UR 7.4.3 ***The exam-mode is described as follows:*** Preliminarily, the buttons <next> and <auto> are not available, neither in the specification phase nor in the solve phase.

UR 7.4.4 ***The exam-mode can be quitted in several ways:***

1. by the learner any time

2. predifined by settings (given by the course admin) the system closes the exam-mode due to the time constraints set (and kindly warning the learner in time)
3. by the course admin at any time (in case of cheating, etc)

UR 7.4.5 ***The couse admin gets several views on the results of an exam:*** Required are views to individual results ("Prüfungseinsicht"), to groups, to several groups for comparison, etc. These views can be printed out ("Prüfungsliste" only with "Matrikelnummer" for public announcement, etc).

Chapter 8

Requirements of the course designer

8.1 General

A course desinger prepares the learning materials and exercises for a course according to the goals of that course, or according to the learners' level. If *ISAC* is being used for a course, specific explanations may be added to the knowledge [1] and examples will be prepared within the example collection.

UR 8.1.1 ***Explanations from other ISAC-sites can be imported.***

UR 8.1.2 ***Examples from other ISAC-sites can be imported.***

These requirements are important, because *ISAC* will be an open system in all respects. In particular, each *ISAC*-site shall go public with the knowledge and the examples available at this site — for information of individual learners as well as a starting point for exchange of course-content between educational institutions.

UR 8.1.3 ***Copyright for the author of explanations.*** See UR.5.0.50.

UR 8.1.4 ***Copyright for the author of examples.***

8.2 Appearance of example collections

One application of *ISAC* will be to mechanize tutoring on existing example collections, as found for instance in traditional textbooks. Thus the example collection must copy the structure already given (the enumerations, page breaks, assembly on a page etc.) in order to allow the learner to find a particular

---

[1] It is the task of a *mathematics author*, however, to edit the *contents* of a math knowledgebase !

example (e.g. given as homework). Traditional textbooks use arbitrary labels for their chapters and sections, the levels are nested arbitrarily deep, and there are arbitrary labels for the examples.

For copy-right reasons it also may happen, that the example itself (i.e. text, formulas, figures) is not displayed, only the respective label. In this case the label should be located exactly at the same position on a virtual 'page' on the screen as the original position in the page of the textbook.

UR 8.2.1 ***The labels of examples are defined by the author***

UR 8.2.2 ***The layout can model textsbooks.*** i.e. the structure of the underlying system must be strong enough to model the structure of textbooks (but also exceeds this structure, e.g. with links)

8.3 The structure of example collections

is a hierarchy of groups of examples. A group of examples consists of the part visible to the learner (which may be copied from a textbook) and meta data for *ISAC* to suggest examples of a level appropriate to an individual learner. There are already requirements concerning examples, UR.7.2.10 and UR.7.2.11, which apply to groups of examples for convenience.

UR 8.3.1 ***There is only one example collection in the system*** in analogy to *one* problem-hierarchy and *one* method hierarchy, see UR.4.4.2. This collection, however, even may comprise several textbooks.

UR 8.3.2 ***There are groups of examples*** with common properties, see UR.7.2.10 and UR.7.2.11.

UR 8.3.3 ***A group and/or an example are weighted*** w.r.t. properties to be defined in a later phase of development (at least the properties 'difficulty' and 'length').

UR 8.3.4 ***A tool for selecting examples*** w.r.t. the weights (and the usermodel etc.) is required. There may be also administrative concerns:

- A limit of the number of solved examples may be defined for a group; if the limit is touched, this group has been mastered successfully by a student within a certain course. An evaluation-function for the performance will be advantageous here in the future.
- There might be obligatory examples in a group; i.e. such an example must be solved by a student in order to have the group mastered successfully.

8.4 Edit examples in the example collection

An example can be described by verbal text, by formulas, and by figures (and eventual by movies). Additionally, each example contains data hidden from the visitor and the learner.

UR 8.4.1 ***Explanatory data can be embedded into examples*** An example can be described by verbal text, by formulas, by figures and possibly by movies. Additionally, each example contains data hidden from the visitor and the learner.

UR 8.4.2 ***A formalization is a list of formulas.***

UR 8.4.3 ***A specification is a triple*** of three pointers to a theory, a problem and a method respectively.

UR 8.4.4 ***Each example is combined with a list of pairs of*** a formalization and a specification respectively. The pairs of Formalization and Specification are used for user guidance while specifying a calculation and are remain to users in a learning situation.

UR 8.4.5 ***Formalizations and specifications are hidden*** from the learner.

UR 8.4.6 ***An example may contain Error Schemes*** An Error Scheme modelling typical errors and providing explanations and specific user guidance for resolving the errors may be stored with an example.An Error Scheme may be paired with an explanation (see UR.8.6.1…UR.8.6.2 below).

8.5 Checks for example collections

For each example or for groups of examples is necessary to run particular checks before delivery to the learners.

8.5.1 Check of format

Like the other items of *ISAC*s knowledge base, examples are stored in XML-format. The data in the XML-files are filtered and converted to HTML for presentation to the user.

UR 8.5.1 ***A tool for checks on the conversion from XML to HTML*** for groups of examples is required, and also for particular examples found to be buggy.

8.5.2 Check of solvability

The course designer addresses items of the mathematics knowledge (theories, problems, methods) prepared by math authors. It may be, that particular examples cannot be solved automatically by means of the knowledge addressed.

UR 8.5.2 ***A tool for batch-processing examples*** is required.

UR 8.5.3 ***Errors for batch-processing*** are: TODO

8.6 Edit explanations in the knowledge base

As already mentioned, explanations are multi-media add-ons to the math-contents of the knowledge base. The latter is generated automatically from the SML-representation in the SML-kernel (where the knowledge in the SML-kernel is due to authoring by math-authors, see chap.5.).

UR 8.6.1 ***An explanation may consist of*** text, formulas, figures, movies, and links into $\mathcal{ISAC}$'s knowledge base itself, into the web, to an example in the resident site. Most HTML-editors, as presently available, meet the requirements for editing explanations.

UR 8.6.2 ***Each item in a knowledge base can have an explanation.*** This should also be possible within formulas, e.g. an explanation to a predicate in a 'where'-field of a model etc. . See also UR.4.4.9.

UR 8.6.3 ***Explanations are course-specific.*** Each course might have different explanations according to the contents of the course, according to the specific example collection, and according to the learners' level.

UR 8.6.4 ***The location of an item in the hierarchy is not persistent over time.*** For instance, a problem like 'linear equation' (together with its children) can be shifted around in the hierarchy of equations. As the knowledge comes from 2 sources, (1) exported from the SML-kernel after authoring by math-authors and (2) manually addition of explanations by course-designers, the identification of a primary key requires particular attention.

UR 8.6.5 ***There are tools to shift hundreds of explanations together with their items.*** This may happen with the above example for 'linear equation' together with its children, and with explanations for several user groups.

UR 8.6.6 ***There are tools for searching the knowledge base.*** Not only the learner (see sect.4.4) but also the author depends on requirements for searching the knowledge; these requirements are different.

8.7 A knowledge profile

for each course results from the explanations following the requirements from above, plus from the detail knowledge is used in, from error schemes and from fill-in patterns for theorems.

UR 8.7.1 ***An example group and/or an example specifies the details*** as defined in UR.4.4.11.

UR 8.7.2 ***There are error schemes,*** eventually several for one theorem in a particular method, or for specific tactics in a particular method. An error scheme is paired with an explanation (UR.8.6.1. . . UR.8.6.2).

An Error Scheme modelling typical errors and providing explanations and specific User Guidance for resolving the errors

UR 8.7.3 ***There are fill-in patterns*** for theorems, eventually several for one theorem in a particular method. Such patterns are used for the dialog atoms UR.6.2.1, e.g. 2 or 5.

In addition to a tactic being applied manually by the user or automatically by $\mathcal{ISAC}$, a tactic can be presented with parts left blank to be filled in by the user.

8.8 A dialog profile

can be preset according to the students' level. These predefined setting can be overridden by the students in most cases, but not in all cases. The dialog profile will be more elaborated as soon the 'dialog programming' and the user model habe been clarified in a future development phase.

UR 8.8.1 ***A dialog mode restricts the dialog atoms*** (UR.6.2.1) to be used by the learner within certain time limits (e.g. during the time of a written examination).

Chapter 9

Requirements of the administrator

The administrator has to install the system and to monitor it. Moreover his duty is to implement the overall design of an *ISAC*-site. This includes introductory pages as well as an basic overall design (Corporate Design, links to “home”...)

9.1 Install and monitor the system

TODO
preliminary requirements:

UR 9.1.1 ***A text-file records the data exchange with the sml-kernel.*** This file serves as a primitive way to check if the sml-kernel is alive as well as for debugging.

UR 9.1.2 ***The system can be asked for the number of users logged in.***

UR 9.1.3 ***The system can be asked for the number of calculations under construction.***

9.2 Customize the appearance in the web

TODO
preliminary requirement:

UR 9.2.1 ***Basic parameters to adapt the HTML-outtput to a corporate design (e.g. colors, font, . . .)***

Part II

Software Requirements Document

This Software Requirements Document is structured along the modules abstracted from the functionality defined in the User Requirements Document. The User Requirements Document, however, is structured along the different kinds of users envisaged. In order to establish comfortable tracing, the $m : n$ relation between user requirements and software requirements will be accurrately recorded in a double-linked way.

Chapter 10

General requirements

10.1 Delopment environment,

standards and components, documentation and revision control are described in appendix C.

10.2 Decisions for underlying systems

have been done already. As this requirements document shows, *ISAC* is a large system: thus it cannot be done from scratch, rather is has to use as much as components already available. The theorem prover Isabelle provides for both, a comprehensive frontend for interactive deduction (i.e. for definitions, axioms and proving theorems), and a hughe mathematics knowledge base. Both are under rapid development, which *ISAC* shall take advantage of. The kernel of Isabelle, however is relatively stable already, and thus the interfaces to *ISAC*s math engine (ME) are sufficiently stable.

SR 10.2.1 ***The deductive part is left to Isabelle.***

SR 10.2.2 ***ISACs math engine closely cooperates with Isabelle***
Thus the math engine is already implemented in the same program language as Isabelle, SML.

10.3 The connection between Java and SML

has to be 'hand-made'. During the next few years there will be several changes at the interfaces between the *ISAC*-components belonging to these two language environments.

SR 10.3.1 ***The SML-kernel is started by a java thread*** which controls the time-out eventuall resulting from non-terminating loops in the knowledge interpreter.

SR 10.3.2 ***The SML standard-output channel is read by parsers,*** one for the SML-structures, and one for the mathematics formulas embedded in the SML-structures.

10.4 System requirements for the users:

Users (with exception of the math authors, which work directly on the SML-kernel) are expected to work on standard-browsers (UR D) and some additional software components.

SR 10.4.1 ***On the client-side a Java virtual machine version ....*** must reside on the local computer of the learners and authors.

SR 10.4.2 ***On the server there is a Linux or Unix*** operating system, Linux version ..., Unix ...

10.5 Communication in the distributed system

[1]

10.5.1 Choosing a Means of Communication

For integration of the various components across several machines an off-the-shelf solution was sought with the following criteria in mind:

Speed Many of the user's requests originating on the Worksheet require processing in the Math Engine. With *ISAC* being an interactive system, we need response times near the response times in normal human interaction.

Security With many of the features of *ISAC* depending on the identity of the user and the enforcement of access rights, basic security features are a necessity.

Mobility Users might want to use *ISAC* at work, in class or at home, even on computers they use only occasionally. This asks for communication robust against changing addresses of client and server computers, but still secure.

Conformity to Standards is especially important for the Worksheet, which runs on the user's machine. An ideal solution would build on standard resources available on the user's machine without the need of additional installation. If installation is required, it must be kept as simple as possible.

With the goal of being open to cooperation with the tools of other projects, a standard solution would ease integration of the different systems.

In addition to that, a standard solution is preferred with the limited resources of the *ISAC* project in mind.

---

[1] Begin of copy from [Kre05] p.58-61.

Several options were reviewed to find a suitable means of communication, among them Java-RMI [2], XML-RPC [3], SOAP [4] and Dinopolis [5].

While the Dinopolis approach looked most promising with respect to the aforementioned criteria, it was not yet available in a stable implementation. Java-RMI was chosen for the implementation of the *ISAC* prototype. The strongest arguments for Java-RMI were the seamless integration into the Java programming environment and the widespread use and accepted stability.

10.5.2 The Dinopolis Middleware project

Dinopolis [Sch02a] is an object management middleware system aiming at object and component retrieval, security and run-time polymorphism. While not part of the current implementation of *ISAC*, Dinopolis still played an important role during the design phase.

The main features of Dinopolis include:

Component and object management Extending the traditional data-and-operations view, Dinopolis objects have the following aspects, all handled by the middleware:

- **Content** is the traditional data content.
- **Meta-Data** can be used for administrative purposes which are not necessarily reflected by the content.
- **Interrelations** associate several objects, of equal or different types, in a m:n manner.
- **Operations** manipulate the content of an object.
- **Services** are GUI building blocks offered by an object to facilitate data manipulation by the user. Services can be viewed as the View and Controller parts of a MVC triple.

All of these aspects are handled transparently by the Dinopolis middleware according to the current run-time context, which not only takes burden from the programmer, but also allows for run-time dynamic typing as a key feature.

Objects and components are accessed through Globally Unique Handles (GUH), a mechanism allowing for unambiguous retrieval of objects independent of their physical locations [Sch02b].

Role-based security Dinopolis takes control over access to components, objects, their data and operations. Access control is based on the user and his role as opposed to the user's identity alone. With Dinopolis, the same user would have different access rights and therefore see different aspects

---

[2] http://java.sun.com/products/jdk/rmi/

[3] http://www.xmlrpc.com/spec

[4] http://www.w3.org/TR/SOAP

[5] http://www.dinopolis.org/

and different behaviour of objects depending on whether he accesses *ISAC* as administrator, teacher or student. All security-relevant operations are handled transparently by the Dinopolis middleware, so the only task left to *ISAC* is to identify the user. As far as objects are concerned, the transparent security system may be considered a special case of dynamic typing described above.

Ability to embed existing systems Existing systems can easily be integrated into Dinopolis by writing embedders for existing objects such as databases. This could ease integrating *ISAC* with other projects without the need to define common data structures. In addition to that, the SML part of *ISAC* could easily be integrated this way.

Platform-independency At present, Dinopolis is being implemented in C++ and Java, but the protocols and algorithms are open to arbitrary programming languages.

The features mentioned above make Dinopolis ideally suited for the needs of *ISAC*, for communication of the various distributed components as well as for security and role-dependent behaviour of the system. Unfortunately, Dinopolis is still being developed and was not finished in time to base the current prototype of *ISAC* on. The *ISAC* team still hope to use the features of Dinopolis in future versions.

10.5.3 Java-RMI

Java Remote Method Invocation provides for communication of processes running in different Virtual Machines, even if the VMs run on different computers. Basically, RMI provides invoking methods on remote objects and passing parameters to remote methods. RMI cannot be compared directly to Dinopolis, because RMI's purpose is communication between remote objects, but not object retrieval, system security or dynamic typing.

To invoke methods on a remote object, the remote object has to implement a public remote interface extending `java.rmi.Remote`. Every method in the remote interface has to declare to throw `RemoteException`. The remote object may implement other methods in addition to the remote interface, but only the methods in the remote interface can be invoked remotely.

Any objects implementing the `java.io.Serializable` interface can be passed as parameter. Unlike other distributed component systems, this is the only condition imposed on parameters.

Apart from the non-restrictive conditions listed, remote objects can be used like local objects once a connection is established.

This tight integration into the Java programming environment is the key advantage of Java-RMI and one of its few drawbacks, as it is limited to Java as a programming language.

[6]

---

[6] End of copy from [Kre05] p.58-61.

```
-------- Original Message --------
Subject: Re: RK noch eine Bitte
Date: Thu, 31 May 2007 18:56:28 +0200
From: Walther Neuper <neuper@ist.tugraz.at>
To: isac@ist.tugraz.at
References: <465DA1BD.506@ist.tugraz.at> <465DECBF.5030607@sbox.tugraz.at><465EE077.9000504

Robert Knighofer wrote:
> Zitat von Walther Neuper <neuper@ist.tugraz.at>:
>
>> _noch_ eine Frage zu Eurer, Deiner und Nelles, Arbeit an der HTL: laut
>>
>>     http://java.sun.com/developer/onlineTraining/rmi/RMI.html
>>
>> sollte RMI bei Firewallproblemen _automatisch_ auf http-tunneling
>> schalten (..it is blocked by the firewall. When this happens, the RMI
>> transport layer automatically retries by encapsulating the JRMP call
>> data within an HTTP POST request. etc..)
>>
>> Habt Ihr bei Eurer Arbeit davon bemerkt ?
>
> Nein, leider.
>
Ah, sehr interessant, dass das nicht so luft ...

> Die einleitenden Worte wrden schon irgendwie darauf hindeuten, dass das
> alles automatisch passiert. Davon haben wir allerdings leider nichts
> bemerkt. Vielleicht haben wir uns dieses feature auch unabsichtlich
> selbst deaktiviert, als wir anstelle der Default Sockets zur
> Kommunikation unsere eigenen Sockets mit fixierten Ports aktiviert haben.
>
?!?!?
> [...]
> "If a client is behind a firewall, it is important that you also set the
> system property http.proxyHost appropriately. Since almost all firewalls
> recognize the HTTP protocol, the specified proxy server should be able
> to forward the call directly to the port on which the remote server is
> listening on the outside."
>
... das gilt fr die Firewall auf der Serverseite; wir brauchen aber nur
die Firewall auf der Clientseite beachten, da der isacserver am IST
_vor_ der Firewall (in der dmz) luft !

> [...]
Vielleicht ist das Ganze einfacher, als wir denken !?!
```

```
>
> lg RK
>
lg Walther
--
------------------------------------------------------------------------
Walther Neuper                          Mailto: neuper@ist.tugraz.at
Institute for Software Technology          Tel: +43-(0)316/873-5728
TUG University of Technology,              Fax: +43-(0)316/873-5706
and HTL Ortweinschule, Graz, Austria      Home: www.ist.tugraz.at/neuper
------------------------------------------------------------------------



-------- Original Message --------
Subject: Re: RK noch eine Bitte
Date: Fri, 01 Jun 2007 12:35:28 +0200
From: Nebojsa Simic <nelle@sbox.tugraz.at>
To: Walther Neuper <neuper@ist.tugraz.at>
References: <465DA1BD.506@ist.tugraz.at> <465DECBF.5030607@sbox.tugraz.at><465EE077.9000504

Quoting Walther Neuper <neuper@ist.tugraz.at>:

> Robert Knighofer wrote:
>> Zitat von Walther Neuper <neuper@ist.tugraz.at>:
>>
>>> _noch_ eine Frage zu Eurer, Deiner und Nelles, Arbeit an der HTL: laut
>>>
>>>    http://java.sun.com/developer/onlineTraining/rmi/RMI.html
>>>
>>> sollte RMI bei Firewallproblemen _automatisch_ auf http-tunneling
>>> schalten (..it is blocked by the firewall. When this happens, the
>>> RMI transport layer automatically retries by encapsulating the
>>> JRMP call data within an HTTP POST request. etc..)
>>>
>>
> ... das gilt fr die Firewall auf der Serverseite; wir brauchen aber
> nur die Firewall auf der Clientseite beachten, da der isacserver am
> IST _vor_ der Firewall (in der dmz) luft !

Das grosste Problem ist dass ISAC Client gleichzeitig ein RMI Server ist ...
Um die Events (CalcChanged + doUIAction) die von Dialog->Worksheet
gehen, der ISAC Client registriert einen RMI Server auf der Client
Machine ... Und wenn der Server versucht, die Verbindung aufzubauen,
dann scheitert das ganze am Client Firewall ...
```

Das alles schaut grob so aus :

```
WindowApplication                 ISAC Server

RMI Client        --------------> RMI Server( 1040, 1050 , ... )
RMI Server (3113) <-------------- RMI Client
```

Wir haben das ganze so umgeschrieben dass die RMI Sockets umgekehrt funktionieren ... Das ist eigentlich schwer zu erklren, aber es schaut das ganze so aus :

```
RMI Client              --------------> RMI Server( 1040, 1050 , ... )
Quasi RMI Server (3113) --------------> Quasi RMI Client
```

Das Problem da ist, dass ganze sehr empfindlich ist ... Falls die Verbindung aufgebrochen wird, gibt es kein mechanismus um die wieder aufzubauen und das System wartet ewig darauf ...

Beim letzten versuch im HTL war die kommunikation zwischen Client und Server schon aufgebaut (wir konnten alle browser fenster aufmachen), aber es war, wie schon gesagt, sehr instabil bzw. hat das System oft abgesturzt ....

Meine vermutung ist dass im ReverseClientSocket und ReverseServerSocket ein bug liegt oder dass man diese klassen irgendwie umbauen muss ...

--
lG,
Nelle

Chapter 11

The worksheet

A worksheet is the protocol of a more or less interactive calculation of an example, and it offers access to all services necessary to calculate the result of the example.

A calculation mirrors the structure of the prooftree.

SR 11.0.1 ***The structure of a calculation is given by the ME.*** Consequently any editing in a calculation affects the parts depending on the edited formula or tactic. Edit the worksheet w.r.t. UR 2.3.5 for publication etc. is done outside $\mathcal{ISAC}$ after having exported the calculation.

SR 11.0.2 ***Export a calculation to a standard format*** preferably XML plus MathML.

Chapter 12

Views on Examples and Knowledge Items

[1]

This chapter describes the software requirements for the browsers and the browserdialogs controlling these browsers.

All browsers (Example-Browser and Knowledge-Browsers) present their output in a similar way. Textual descriptions have to be combined with images, formulas, formalisations, problems, ... and links to further informations. All items of information might be interlinked with each other.

12.1 General Requirements

SR 12.1.1 ***Unique identification by a GUH.*** Each item of the examples collection or the knowledge base is uniquely identified by a GUH (Global Unique Identifier). The GUH is a String starting with `'thy_'` for theory elements, `'pbl_'` for problem elements, `'met_'` for method elements and `'exp_'` for examples.

SR 12.1.2 ***Presentation in a standard browser and in the $\mathcal{ISAC}$-browsers.*** The knowledge elements and examples can be viewed in a standard browser as well as in the $\mathcal{ISAC}$-browsers. Thus, knowledge elements have to be available in HTML form in some way.

SR 12.1.3 ***GUHs as links.*** Links between elements of the knowledge base are defined by the GUH of the link target. To make the links work in a standard browser, some path information has to be added to the URL in the HMTL representation of the knowledge elements.

---

[1] Begin of [KÖ6] p....– p....

SR 12.1.4 ***Asynchronous load of content.*** The *ISAC*-browsers load the content to be displayed asynchronously. That means, that the user interface does not block until the page is loaded. The page is loaded in an extra thread instead.

SR 12.1.5 ***The hierarchy is displayed in a frame*** in order to have it visible all the time.

SR 12.1.6 ***The hierarchy has arbitrary levels.***

SR 12.1.7 ***The hierarchy shows the position*** of the related element displayed in the browser-window.

12.2 The Knowledge Browsers

The following enumerations do *not* show all items contained in the respective sml datastructure; rather it shows the 'most important' ones — a preliminary decision, which will be overlayed by filtering due to the dialog-guide.

SR 12.2.1 ***A problem page consists of:***

1. name of the problem or the 'CAS-command' (a short command similar to an algebra system; e.g. *solve*)
2. a model consisting of the fields 'given', 'where', 'find' and 'relate'
3. explanations
4. the authors
5. the position within the problem-hierarchy displayed in the hierarchy-frame)

SR 12.2.2 ***A method page consists of:***

1. the script
2. a name (only displayed in the hierarchy-frame)
3. a guard consisting of the fields 'given', 'where', 'find' and 'relate'
4. explanations
5. the authors
6. the position within the method-hierarchy displayed in the hierarchy-frame

SR 12.2.3 ***A theorem page (within theories) consists of:***

1. the name of the theorem
2. the formula of the theorem

3. a link to the proof of the theorem within Isabelle
4. explanations
5. the authors (math authors and course designers)

SR 12.2.4 ***A ruleset page (within theories) consists of:***

1. the identifier of the ruleset
2. the type of the ruleset (`Rls, Seq, Rrls`)
3. a list of rules and links to the rules. Rules can be theorems other rulesets or operations.
4. a rewrite order
5. explanations
6. the authors (math authors and course designers)
7. the position within the theory-hierarchy displayed in the hierarchy-frame

SR 12.2.5 ***A htmldata theory page consists of:***

1. explanations
2. the authors

SR 12.2.6 ***Similar representation for static and dynamic content.*** If elements of the knowledge base are shown in the $\mathcal{ISAC}$-browser, the content can be enriched with dynamically generated context dependent information. Any selected formula in the worksheet has a context to an element of the knowledge base. This context information is inserted into the HTML content displayed in the browser if the feature is activated.

SR 12.2.7 ***Data and representation separated.*** The knowledge data and its representation should be separated well.

SR 12.2.8 ***Easy generation of different representations.*** SR.12.2.7 The system must provide an easy way of generating different representations of the same knowledge data. SR.12.2.7 is a precondition to make this possible.

12.3 The example browser

In contrary to the knowledge browsers, the presentation of the contents of a browser-window is *not* generated automatically. UR 8.2.2 requests for a layout 'handmade' by the course designer; there are, however, a lot of attributes invisible for the learner to be added by the course designer, too.

SR 12.3.1 ***The headlines of the example-hierarchy:*** The hierarchy comprises the labels of the chapters, sections, subsections etc. plus the respective head line, and the blocks of examples with the respective labels – all defined by the user (see UR 8.2).

SR 12.3.2 ***Metadata for selecting examples:***

Attributes of examples and collections are numerous, and thus defaults help to safe space.

SR 12.3.3 ***Visibility of examples:*** Visibility of the examples is defined in two levels: (1) 'locked' displays the text of the example, but doesn't allow to calculate it; and (2) 'invisible' doesn't display the example at all.

SR 12.3.4 Only ***visible examples are checked for being locked.***
[2]

---

[2] End of [Kö6] p....– p....

Chapter 13

The dialog guide

As of 2012, the dialog is still a stub which just passes input from the learner on to the math engine, and results of calculation from the math engine back to the learner. Presently, replacement of Java code by a rule-based system in the dialog is under construction. A UserLogger is being re-engineered in order to serve dialog guidance planned for the future.

13.1 Components of the dialog guide

SR 13.1.1 ***A dialog consists of***

1. a *dialog profile* which initializes the dialog
2. a *dialog history* which records each step in problem solving
3. *dialog rules* which which determine the flow of interaction
4. a *dialog guide* which selects the *dialog rules*;
   these rules determine what is called a 'dialog mode'
5. a *dialog state* which comprises the data from the current session
6. a *learner model* which abstracts from the *dialog state*s during a course.

The above notions will be used without the initial 'dialog' in a respective context.

SR 13.1.2 ***A dialog profile determines initialisation*** of the dialog guide at the first registration and also in case of an 'exclusive' profile (in exams etc). A dialog profile can be 'exclusive' in order to feature assessments: within explicit time limits an exclusive profile inhibits all other sessions of this student.

SR 13.1.3 ***The dialog history records*** **dialog atoms** , i.e. steps of problem solving (an input by the learner together with the reaction of the math engine) or look-ups in the knowledge base.

SR 13.1.4 ***A set of dialog rules comprises*** **dialog patterns** which in turn are composed from *dialog atoms.*

SR 13.1.5 ***The dialog guide selects*** **dialog patterns** according to the *user model* from previous sessions, from the *dialog profile* at start of the current session and from the dialog state modified during the current session.

SR 13.1.6 ***The dialog state comprises current data***, i.e. the *dialog history* of the current session and the sets of *dialog rules* employed in the current session.

SR 13.1.7 ***The learner model abstracts over all sessions*** in a course such that the next session can be started adapting to the personal needs of the learner.

/ − − − − − − not revised since 1^{st} design phase in 2001 − − − − − − \

13.2 The dialogstate

is read and updated during *one* session. A dialog resumes the dialogstate from the previous session done as a member of the same student-group.

SR 13.2.1 ***The last dialogstate is stored*** at the end of a student's session for each group the student is a member of. When storing and replacing the previous dialogstate, this dialogstate is transferred to the history of the usermodel (eventually after compression).

SR 13.2.2 ***The dialogstate has the attributes***

begin	timestamp of begin of session
provided-end	e.g. for examinations
actual-end	empty, or timestamp of end of session
group	the user has started the session with
interactions, for each:	
timestamp	
label of example	empty if *ISAC* entered via KB
input	tactic, formula, command or label in KB
response	??? of which part of system ???
pattern	of dialog
activity	
stepwidt	
. . . TODO . . .	

The use of these fields is shown by use-case UC TODO.

13.3 The usermodel

consists of two parts: the settings of the personal preferences and the history of (condensed) dialogstates. The history is constructed from the dialogstates:

before a dialogstate is being replaced at the start of a new session, its data are restructured and appended to the history.

SR 13.3.1 ***The usermodel has the attributes***

settings	
patterns, for each:	of dialog
activity	
stepwidt	
. . . TODO . . .	
history, for each session:	
begin_end	2 timestamps
group	the user has started the session with
kb_touchs, for each:	
label of KB item	
timestamps	
examples, for each:	
label of example	
begin_end	2 timestamps
finished	yes/no
performance	from example.evaluation.TODO and from dialog.interactions

The use of these fields is shown by use-case UC TODO.

\ − − − − − − not revised since 1^{st} design phase in 2001 − − − − − − /

legend to the reader's marks:

Part III

Architectural Design Document

Chapter 14

Surveys

14.1 Survey on the components

Below there is a short description of the role of the modules shown in Fig.14.1.

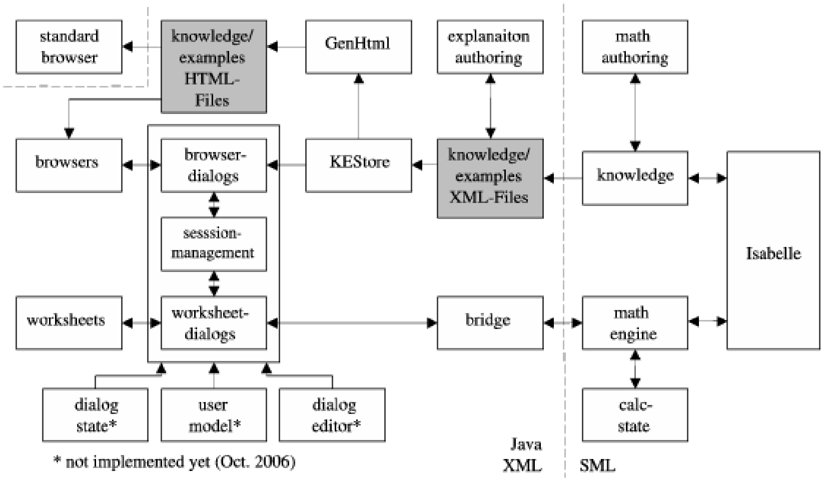


Figure 14.1: $\mathcal{ISAC}$s components

Browsers (Example Browser, Knowledge Browser) allow to browse through the content of $\mathcal{ISAC}$s knowledge base. The browsers also allow dynamic views onto the knowledge, where an actual example is the starting point to find appropriate (types of) problem(s), theorems in use, etc.

Worksheet is the learners main working place when calculating an example in interaction with $\mathcal{ISAC}$. A calculation on this worksheet is intended to be like traditional paper-and-pencil work.

Dialog-guide consists of two parts, the worksheet-dialog and the browser-dialog. The former is responsible for the interaction with the math-engine (appropriately detailed steps etc.), the latter for the views into the knowledge base (which might depend of which course the learner is member of, etc.)

Dialog editor allows to define dialog patterns and strategies, and to preset dialog states.

User-model holds settings for the dialog-state and a (compressed) history of dialog-states.

Dialog-settings are given for each learner in order to preset the dialog appropriately to the respective preferences (graphical representation etc.).

Explanation Editor is used by a course designer or a teacher (no specific exptertise in computermathematics is required !) (1) to prepare examples and (2) to extend *ISAC*s knowledge base with explanations.

Examples are provided with with hidden formalisations for automated solving as *ISAC*s prerequisite for user guidance. They can exactly be formatted like example collections in traditional textbooks.

Knowledge + explanations is what the learner sees during problem solving:

1. Theories with axioms, definitions and theorems (proven with Isabelle)
2. Problems like types of equations, or problems of applied math
3. Methods to solve the problems.

These three parts of knowledge have been imported from SML in a batch process, and this raw material can be augmented with multimedia explanations by the explanation editor, presumerably specific to courses.

Math authoring tools provide for the compilation of the knowledge, which *ISAC* requires for automated problem solving (as the prerequisite for user guidance). This task requires more or less expertise in computermathematics.

Knowledge comprises theories, problems and methods, and is held in the SML-core for efficient access by the math engine. This knowledge is exported to XML (in a batch process) for augmentation with explanations using the explanation editor.

Math engine is a fairly small knowledge interpreter, which depends on the knowledge and some of Isabelles services (matching, parsing, pretty-printing etc.). Thus the math engine is written in the same language as Isabelle, in SML.

Calc-state (short for state of a calculation) is held in a calc-tree for each calculation.

Isabelle is an interactive theoremprover which lays the deductive foundation of *ISAC*.

Bridge wraps the SML-process of the math-engine as a Java-object, i.e. it provides for data-exchange between SML and Java, for multi-user facilities and for distributing work-load to several instances of the math-engine.

[1]

14.2 Basic Concepts for Separable User Interfaces

Apart from the benefits of structuring complex systems, separable user interfaces provide adaptability of look-and-feel without the need of reworking the entire system. For our analysis, we will concentrate on the Seeheim and Model-View-Controller (MVC) basic architectures.

14.2.1 The Seeheim Model

The Seeheim Model [Pfa85] splits the entire system into three components as follows:

The Presentation Layer is responsible for translation of physical representations, such as images, sounds, key-presses or mouse events into the logical concepts of the system and vice versa. Typical tasks of the Presentation Layer include rendering data on the display and parsing user input.

The Dialogue Controller defines the structure of the interaction between user and system. Typical tasks of the Dialogue Controller include accepting events the user triggered on the Presentation Layer, routing events to appropriate destinations and making decisions whether and how to notify the user of changes in the state of the system. In other words, the Dialogue Controller defines (and enforces the use of) a language for the interaction between user and application.

The Application Interface is an abstraction of the application's data and procedures from the user interface's point of view. It maps objects and operations on the user interface to actual data objects and code in the application, thus representing the application's functionality in a concise and consistent way.

Note that in figure 14.2, the messages are named "notify" and "request" from the user's point of view. From the Dialogue Controller's point of view, the messages are distinguished by their direction in or out of the Dialogue Controller. Even more so, the Application and the Presentation Layer (representing the

---

[1] Begin of copy from [Kre05] p.53-57.

user) do not differ in a structural way. Both are merely objects generating events which might be of interest to other objects and have to be handled according to the Dialogue Controller's state and logic. It is the semantic in the Dialogue Controller's logic that makes a difference between user and application, if any.

14.2.2 The MVC Architecture

As opposed to the Seeheim Model, which structures the system as a whole, the MVC architecture [BMR^{+}96] is grouped around single data objects as follows:

The Model is any data object in the application requiring user interaction.

The View is an object providing a visual representation of the respective Model, thus enabling the Model to output its data.

The Controller is an object accepting user input and notifying the Model or the View accordingly, thus providing the user with a means of controlling the Model.

In a complex system, the link between application and user can contain several Model-View-Controller-triples, each grouped around a specific data item.

14.2.3 Comparing the approaches

- Without the need to pass the Dialogue Controller on every interaction, the MVC model tends to be faster in runtime. This is particularly important for giving the user immediate feedback about his current actions and options.
- Being built around smaller units of single objects, MVC is more flexible and easier to develop and extend.
- On the other hand, MVC lacks the clear separation between application and presentation layer, spreading the functionality across a multitude of interacting MVC triples. While this eases development, the resulting complex dependencies make it harder to understand or debug the system as a whole.

14.2.4 Implications for $\mathcal{ISAC}$

The design of $\mathcal{ISAC}$ is based on the Seeheim Model, for the following reasons:

- A clear separation of Application, Dialogue Controller and Presentation layer is a design goal because:
  - At present, the Application itself is written in SML, whereas the rest of the system is being implemented in Java. The necessity to interface the different worlds of different programming languages implicitly separates Application from User Interface.

  - The already-implemented Math Engine with the Isabelle system in the background uses more computing resources than a typical consumer machine can provide at present. This and the goal to centralise mathematical knowledge and example collections for groups of users suggests running the Application on a dedicated server.
  - To minimise effort and expenses on the side of the user, the part of the system running on the user's machine should be kept as small as possible, a Java-capable web browser being the aimed-at minimum. An ideal design would leave only the Presentation Layer running on the user's machine.
  - The goal to adapt *ISAC*'s behaviour to the situation of the individual user's situation asks for a well-defined, configurable and exchangeable Dialogue Controller.
  - As it seems foreseeable that the design of *ISAC*'s Dialogue Controller will pose questions going well beyond the scope of a single master's thesis, a separable Dialogue Controller component will ease independent research. We expect the development of new approaches to dialogue description languages as practical experience with a *ISAC* prototype becomes available.

- The major drawback of the Seeheim Model, the difficulty to provide immediate feedback to the user, is not relevant to the design of *ISAC*, as the complex mathematical knowledge involved requires consultation of the Math Engine for even basic feedbacks. With the present speed of the Math Engine, any additional delays from the use of the Seeheim Model will not be perceivable by the user.

  Any operations not involving mathematical knowledge can be handled locally by the Presentation Layer, in a MVC manner, if desired.

- The aforementioned goal of adapting *ISAC*'s behaviour to the situation of the individual user can be easily adressed by augmenting a centralised Dialogue Controller with a User Model. The User Model would store preferences set by the course designer and the user himself. The Dialog Guide would report activities to the User Model, which would store these reports and process them into an abstraction of the user's preferences, experience and behaviour. The other way round, the User Model would be queried by the Dialog Guide for clues how to behave in a specific user-interaction situation.

Based on these considerations, the top level design of *ISAC* looks like this:

Math Engine or Kernel In terms of the Seeheim Model, this is the Application. This component is already implementetd in SML and is intended to run on a centralisad dedicated server. All mathematical knowledge resides in this component, all calculations are done here. The SML system communicates via the standard input and output text streams.

Dialog Guide and User Model In terms of the Seeheim Model, this is the Dialogue Conroller. This component is being implemented in Java. All user interaction is controlled by this component, and this is the only component aware of the individul user.

Worksheet In terms of the Seeheim Model, this is the Presentation Layer. This component is being implemented in Java, with the additional goal of running in standard environments encountered on a consumer PC installations, as this component is intended to run locally on the user's machine. The Worksheet is the only component aware of visual aspects of data, such as formatting, and the only component with direct user-interaction.

[2]

14.3 Survey on the architecture

[Hoc04] describes the architecture depictured in Fig. 14.5 on p.75 as follows.

The system architecture is designed as a *distributed system.* This means that the components described below are designed process independently and they can be executed on different computers concurrently. The thesis covers the design of the graphical user interface component and the interfaces that are necessary to connect to the other components of *ISAC*.

The components of *ISAC* are:

- The **backend** of *ISAC* is responsible for doing the calculation and holding the mathematical knowledge also known as knowledge base. The mathematical engine can only do one calculation at a time. This is a restriction that needs to be bypassed.

- Hence, the component called **Bridge** is designed as a *multiuser* component. Multiuser in this context means that the Bridge distributes the simultaneous user requests over several instances of the mathematical machine. A more detailed description can be found in [Gra04]. Note that in this context the name Bridge has nothing to do with the structural software pattern "Bridge", described in Design Patterns by the "Gang of Four". Rather, the component Bridge in the *ISAC* system architecture can be described as an "Adapter" in the meaning of a design pattern. It converts the interface of the mathematical engine into another interface that the other parts of *ISAC*, especially the *WorksheetDialog*, expect. Moreover, the Bridge also "converts" the functional software design paradigm of the mathematical engine into an object-oriented design paradigm the other parts of *ISAC* expect; therefore, the Bridge lets components work together that could not otherwise because of incompatible interfaces.

---

[2] End of copy from [Kre05] p.53-57.

- The **WorksheetDialog** acts as a middleman between the Worksheet and the Bridge. It is the central component while solving an example. The Worksheet provides the user with information during a calculation in cooperation with the WorksheetDialog. The WorksheetDialog decides how detailed the user should see the information, depending on the user history, experience of the user and role of the user. The WorksheetDialog can also restrict the access to parts of the knowledge to ensure that the user is not swamped with examples for which his experience level is not high enough and he is not meant to solve yet.

  The design of the WorksheetDialog at the moment concentrates on the solving part. User history, restricting access and different grades for showing information are scheduled in the design but not yet completely finished.

  Worksheet and WorksheetDialog stand in a 1:1 relation, which means that for each Worksheet a separate WorksheetDialog is necessary.

- The **BrowserDialog** is the component in charge as far as the knowledge base is concerned. Strictly speaking, the *InformationProcessor* provides information about the knowledge base. Each part of the knowledge base (problem, method and theory, see section **??**) can be accessed and also the example collection can be accessed. Another task of the InformationProcessor is the authorization of the user that wants to connect to the *ISAC* system. Currently a username and a password are used to authorize the user. Security considerations like encryption have not been included in the design process so far.

  The InformationProcessor provides a well designed interface that can be used by a client and is described in more detail in section 21.3.1.

  The *SessionDialog*, which is also part of the BrowserDialog, is responsible for the (re-) identification of the user. A user might connect and log in several times (e.g. with different applications), then the SessionDialog has to ensure that all WorksheetDialogs for this user work on the same data. A deeper look into the design of the SessionDialog can be found in [Gri03].

  There is only one SessionDialog per *ISAC* system.

- The **Graphical User Interface** is responsible for establishing a connection to *ISAC*. It consists of two components, namely, the *HierarchyBrowser* and the *Worksheet*. The requirements for these components have already been described in sections **??** and **??**. The following sections describe the design of the graphical user interface and the communication interfaces to the WorksheetDialog and BrowserDialog.

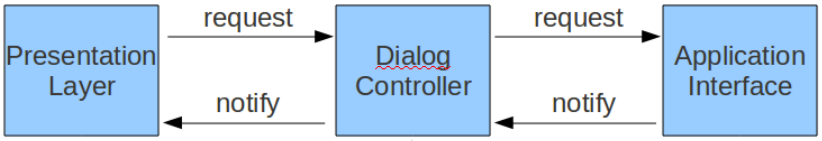


Figure 14.2: Interaction in the Seeheim Architecture

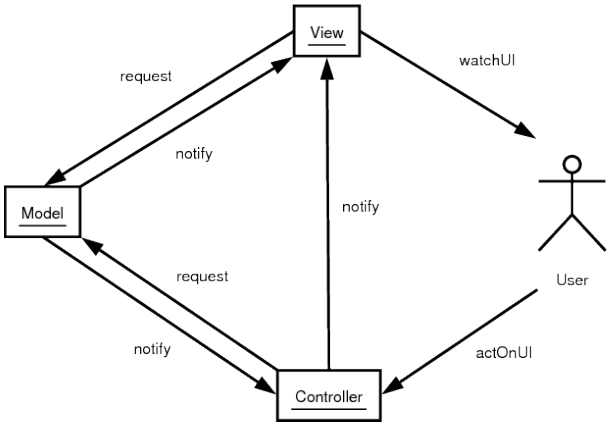


Figure 14.3: Interaction in the MVC Architecture

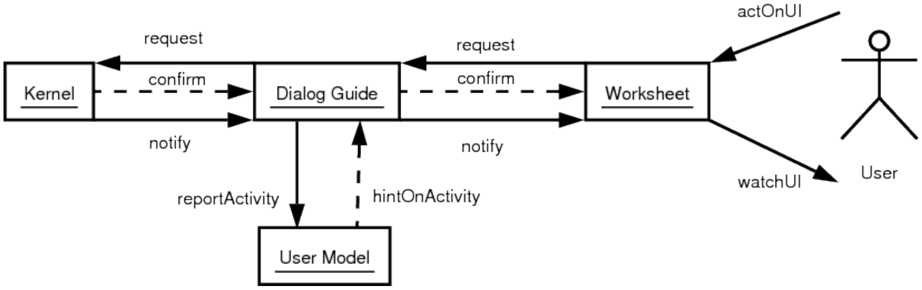


Figure 14.4: Basic $\mathcal{ISAC}$ architecture for calculations

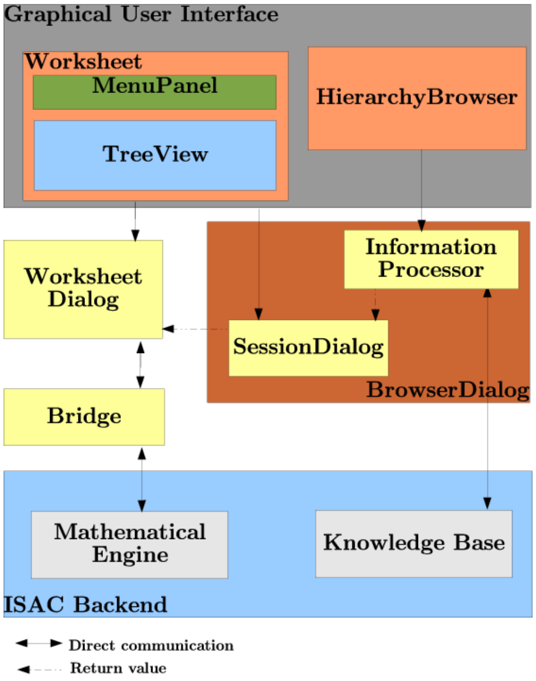


Figure 14.5: The current design of the *ISAC* system

Chapter 15

Session Management

15.1 The Dialogs

The *dialog* is the "heart" of the system which controls the behavior of the different modules. It is responsible to adjust the reaction of the system to fit the learners (and teachers) demands. Among others, these demands contain:

record a learners success by means of solved examples. This record can be used to create a set of proposed examples for a single user. Furthermore, the teacher can use this feedback to enhance the quality of his lecture (e.g if a single example causes problems for most students)

restrict access to parts of the knowledge The set of available examples can vary depending on the already solved examples or the progress of the lecture to ensure, that the student is not swamped with examples he is not meant to solve yet. Furthermore, the available information have to be restricted while the learner writes a test. This also implies, that the access for not authenticated users can be restricted as well (by IP). (The user may not access the public information)

communication between different applications $\mathcal{ISAC}$ uses different applications to access the knowledge and mathematical capabilities of the system. For example: from the users point of view, the worksheet calculates while the KnowledgeBrowsers are used to select problems or methods for use within the calculation. In this case, worksheet and KnowledgeBrowser are just two access-points for the same application. The dialog establishes and controls the connection of these access-points.

manage the connected users A user might be connected using several applications. For instance he can calculate an example and search a related *problem* using a *browser*. The dialog has to ensure, that these applications work on the same data. Therefore a *session* is used which is aware of the different connections of a user.

Because of the various duties of the dialog, it is split up into a number of modules. Applications have an own "peer" which is used to communicate which them. These peers act as a kind of "border" – informations the user is not meant to get, may not cross this border but are filtered before. This filtering goes along with an possible transformation of the data.

15.1.1 Session-Dialog

The *session-dialog* governs the user-login and performs all necessary actions to build up a users dialog. It is started on system-startup and waits for connects by an *session-controller*. There is only one session-dialog per *ISAC*-System.

The session-controller is a client-side program which performs the login and starts the *ISAC* frontend-applications on a users machine. Besides the authentication, the session-dialog is also responsible to instantiate and interlink the application dependent parts of the dialog-layer as there are the Browser-Dialog and the WorkSheet-Dialog.

Although one user can log in twice at a time, only one application dialog is created to ensure that all user-dependent information are up to date. Therefore the session-dialog has to maintain lists of logged in users and their respective dialogs.

For the dialogs providing user-guidance see chap.16 below.

15.1.2 Browser-Dialog and Worksheet-Dialogs

Both dialogs are discussed in a separate chapter, in chap.16. TODO.WN060705

15.2 User Data and Access Rights

15.2.1 Dialog Guide and User Model

15.2.2 User-Administration

The *user-administration*-module helps the dialog to maintain the users and control the access for *courses*.

user

Information stored in an user-module contain personal information like name and login, as well as important administrative information like the courses he is member of and a reference to the *user-model* which contains all information which are relevant to build up an environment for the user (solved examples, history, ...)

```
user:
   FirstName:String
   LastName:String
```

```
login: String
password: String (encrypted)
courses: {course*}
solved_examples, history, ...
temporary_environ: hierarchy_opt
```

The field *temporary_environ* is used in case of an examination. If this field is set to non-empty, a user has only permissions to the elements of the referenced hierarchy. This hierarchy typically contains the examples to solve and a few explanations which are permitted within the exam.

course

A *course* is a object within $\mathcal{ISAC}$ to define a learning environment for different users. Typically, a course contains explanations to the mentioned theories and examples which are organized within an hierarchy. There are tools which help a course-designer to build up a hierarchy by e.g. copying a part of an other hierarchy. A typical reason for doing this is to copy a part of the problem-hierarchy and afterwards enrich the items with explanations and examples fitting the audience of the course

Additionally, examples and general explanations can be added.

The hierarchy of a course is located within the *KE-Store* while the object describing the actual *course* is located within the *user-management-module*. Permissions are handled based on the user and *KE-Objects* (hierarchies, examples, explanations, knowledge)

```
course:
   metadata:
   name:string
   hierarchy: hierarchy-id
   members:{user-id, user-id, ...}
   admin: user-id
```

There are tools which perform the tasks of adding an removing users to a course. While adding a user to a course is relatively simple – ensure the user has proper rights for all *KE-Objects* within the used hierarchy – the task of removing a user from a course is more complicated: the user might be member of more than one course which access the same example. Removing access-rights for such an example might collide with the course the user is still member of. Therefore, the tool has to check all courses of the user before removing any permissions.

The data-structure is usable for another set of tools which can act on them – like hide an example for all members of a course till they are able to solve them or contact all of the course-members.

15.2.3 Permissions-module

The *permissions-module* stores and maintains informations about which user might access to which *KE-Objects*. The administration of the permissions is based upon the users stored within the *user-administration*-module

Permissions are applied, whenever an KE-Object is accessed. When informations "leave" the dialog – e.g. when they are delivered to the *browser*, the (browser-) dialog can remove links whose destinations are not accessible. Note, that the dialog can decide not to show an example to a user although he has the proper permissions – in this case, the references are filtered out by an didactic filter.

Permissions can be set at once to a number of users using tools which can access the objects within the *user-management-module*. E.g. a tool is used to to add and remove users from a course. These tools traverse the members of the course and set the proper permissions for them. There is a special mode to limit access, when the user participates to a exam. In this case access is limited to the environment given in the *temporary_environ* field of the user-object.

Chapter 16

Dialog Guide

[1]

16.1 Browser Dialogs and WorkSheet Dialog

In contrast to most currently available algebra systems, *ISAC* bases its calculations entirely on rules and knowledge visible to the user. For every step done in a calculation, there is a justification in *ISAC*'s knowledge base, which can be displayed on request. Even more importantly, these justifications are meant to be understood by the user, as they are expressed in terms of human mathematical reasoning, not in sophisticated and optimised algorithms.

As *ISAC*'s knowledge base can be understood by humans, it can be used as a reference or even as a learning tool. Interaction with the knowlegde base is moderated by a dialogue controller (see also section 14.2.1), in a way similar to the interaction with the math engine in an ongoing calculation. Such a dialogue controller is responsible for the processing of actions coming from the math engine or the knowledge base and also for the response to user actions.

As it was already discussed in UR.3.1.1 the design took the designers of *ISAC* to 4 different browsers and guarantee a well designed abstraction every browser gets its own dialogue controller. Such a dialogue controller is one of the basic parts inside of *ISAC*. In the following paragraphs the term browsers will be used to describe the three knowledge browsers and the example browser. So these browsers with their corresponding dialogue controllers can be seen as one subsystem.

Mathematical knowledge is normally very static, but with *ISAC* you can also make the current problem, method, theory or example available thru one of your browsers. So the basic design led to two subsystems which can be used separately, each with a presentation layer and a dialogue controller, but they can also access the actual context of each other.

---

[1] Begin of copy from Alan Krempler [Kre05] p.61-62 with updates by Georg Kompacher [Kom07].

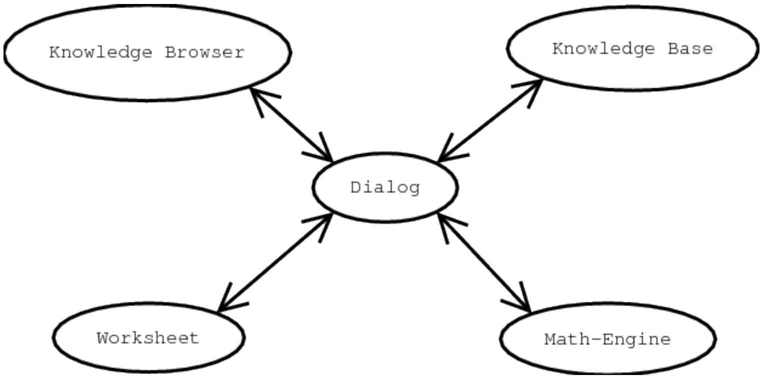


Figure 16.1: The first sketch for *ISAC*'s architecture

The two subsystems interact in the following points:

- Both dialogue controllers share a common User Model, for a inventory of knowledge supposedly known to the user, be it from browsing the respective knowledge item, be it from having used specific knowledge in a calculation.
- When browsing thru the knowledge base, an example illustrating the presented concept can be calculated.
- When doing a calculation, items from the knowledge base justifying the correctness of the calculation can be displayed. This kind of context-based access to math knowledge is considered an efficient method of learning in *ISAC*. The other reason for accessing the Knowledge Base from a calculation is, to explore the application of nearby knowledge-items to the calculation.
- If the user is within a calculation and wants to apply a theorem to the current formula, he can switch to the theory browser and apply any formula of the knowledge base which matches. This communication between a browser and an active worksheet is done between their dialogue controllers.

  Separate design considerations about the Browser Dialog see chap.16.8.

[2]

[3]

---

[2] End of copy from [Kre05] p.61-62.

[3] Begin of copy from [Kre05] p.57-58.

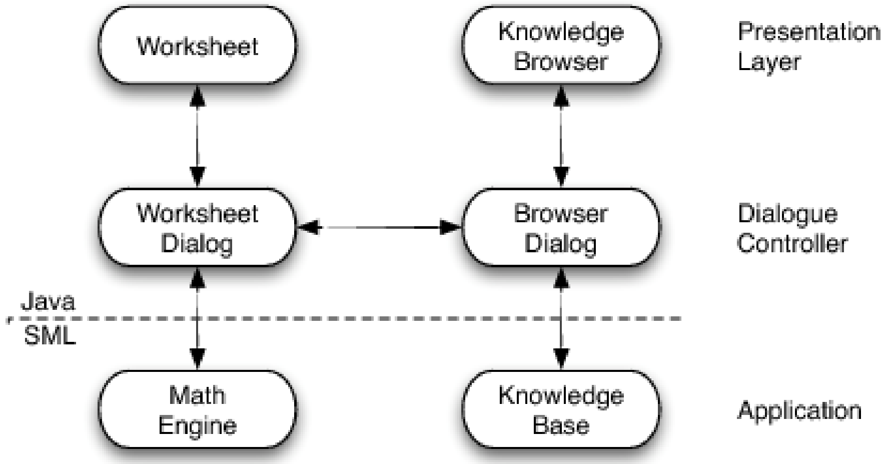


Figure 16.2: Design based on the Seeheim model and showing the separation of browsing the knowledge and calculating

16.2 Location of the Dialog Guide

With at least two machines involved - the user's computer with the Worksheet and the server with the Math engine - the question where to put the Dialog Guide remains. The Dialog Guide accesses the Math Engine, the Worksheet and the User Model frequently. For simplicity, mobility, security and centralisation reasons, the User Model cannot reside on the user's machine. The same is true for the Dialog Guide. The Dialog Guide with the persistent data of the User Models could run on the server together with the Math Engine or on an other server of his own. The Dialog Guide is designed with the ability to run on a machine of its own in mind. The final decision on the location of the Dialog Guide will be based on tests with the prototype implementation.
[4]
[5]

16.3 The Interfaces to the WorkSheet Dialog Component

Data exchanged at the interfaces of the WorkSheet Dialog component include:

Examples to be started When initialising the Dialog to moderate a process of calculation, the starting point can be an empty worksheet or an Example

---

[4] End of copy from [Kre05] p.57-58.
[5] Begin of copy from [Kre05] p.65-67.

from the example collection. In case of starting from an Example, the Example has to be passed to the Dialog.

Notifications about updates in a calculation With present technology, calculations done by the Math Engine may take longer than the average user would wait. Moreover, response times are not easily predictable, so waiting for a call to return would block the WorkSheet Dialog - hence user interaction - for too long a period of time. Therefore calls to the Math Engine return immediately, with asynchronous notifications being sent when the Math Engine completes a request. In addition to continuous attention to the user, this approach allows for several users watching one and the same calculation on their respective Worksheets and being even notified of updates in the calculation requested by other users. For efficiency reasons, the update notifications contain hints about which parts of the Calc Tree may be affected by the update. These notifications are passed from the Math Engine to the Dialog and from the Dialog to the Presentation Layer.

The calculation itself The Dialog needs access to the Calc Tree stored in the Math Engine and passes a filtered version of the tree to the Worksheet for display. The Dialog cannot understand the mathematical meaning of Formulas, but is is interested in identifying Tactics. It is the Tactics which the user is learning to apply and the WorkSheet Dialog has to provide appropriate user guidance for.

Calc Head As with the Calc Tree during the Solving Phase, during the Specifying Phase a Calc Head has to be shared between Math Engine, WorkSheet Dialog and Worksheet.

Notifications about user requests The Dialog has to be informed about actions the user triggers on the Worksheet. The Dialog in turn translates the user actions into internal state changes or requests to the Math Engine.

Requests to the Math Engine As the Math Engine stores the only instance of the Calc Tree significant to further processing, all manipulations of the tree have to be done by the Math Engine. Request to edit the calculation originating from the user are processed by the WorkSheet Dialog and execution is requested from the Math Engine.

Information touched Records of the user's interaction with $\mathcal{ISAC}$'s knowledge are kept in the User Model and abstracted to $\mathcal{ISAC}$'s view of the user's knowledge and abilities. The User Model is informed about every interaction of the user with the calculation or the Knowlegde Base. The User Model's abstraction is in turn queried by the WorkSheet Dialog to decide on details of user guidance.

Dialog Atoms Information about the Dialog Atoms involved in user interaction is passed to the User Model to record not only the fact that the user interacted witch certain parts of knowledge but also the nature of the interaction.

User settings The user's preferences about the way he wishes to be guided have to be communicated to the WorkSheet Dialog, whereas preferences about the visual appearance of the GUI are communicated directly to the Worksheet.

[6]

[7]

16.4 Controlling the Course of Interaction

The WorkSheet Dialog is responsible for guiding the user the way to obtaining a solution to a problem.

16.4.1 Dialog Phases

On a coarse level, interaction goes through several phases with a fixed sequence, independent of the particular problem being solved (UR.2.2.3). These so-called Dialog Phases have been modelled on a state machine with well-defined states and transitions between the states. During each of these phases, the WorkSheet Dialog behaves differently and reacts to different requests. Regardless of mathematical context, the Dialog Phases provide a certain degree of error-robustness by recognising out-of-order events.

With the Dialog Phases and their relationships becoming more complex in future development, providing separate sub-classes for the Dialog's behaviour in different states may be appropriate, as described in the State pattern [GHJV95].

Initialising

To start interaction, the WorkSheet Dialog has to establish connections with the components it interacts with, a Worksheet representing the Presentation Layer and a Math Engine representing the application (UC.30.1.1.3). In addition to that, the WorkSheet Dialog needs information about the user it deals with, to be able to adapt its behaviour accordingly (UC.30.1.1.1, UR.2.1.1). Only after being provided with a CalcHead to act upon, optionally filled in with a Formalization of a pre-defined Example (UC.30.1.2.1), the Dialog can enter the Specification Phase.

Specifying

The goal of this phase is to gather enough - and consistent - information to start solving (UC.??). During this phase, the user can add information to a

---

[6] End of copy from [Kre05] p.65-67.

[7] Begin of copy from [Kre05] p.67-76.

CalcHead, in arbitrary order. After every item added, the CalcHead is checked with the Math Engine for consistency and completeness (UC.30.2.1.1, UR.4.3.1).

Requests for help entering items cannot be answered by the WorkSheet Dialog because of lacking mathematical knowledge. Such requests are passed to the Math Engine, if allowed by the user's settings. The Specifying Phase can be finished only after the CalcHead is confirmed being complete and consistent by the Math Engine (UC.30.2.2.2).

Solving

To enter the Solving Phase, a valid CalcHead is required. Consequently, the mathematical situation initially described by the CalcHead is transformed towards a situation called result. The transformation is performed in steps (UR.2.2.7) which are recorded in a CalcTree (UR.2.2.10). As opposed to the Specifying Phase, the steps are not cumulative, but sequential. This implies that while it is possible to change steps already taken, such an action is likely to render subsequent steps invalid (UC.30.3.2.5). The transformations are not performed by the WorkSheet Dialog itself but by the Math Engine. Requests to take a step are passed to the Math Engine and steps entered by the user are checked by the Math Engine (UR.2.3.6). Note that the WorkSheet Dialog does not know anything about mathematics, it knows only about the structure of interaction in problem-solving. As stated before, transformations can be done entirely by the user or by the Math Engine, or with combined effort of both. This opens up a spectrum of interactional possibilities how to take a step and the various possibilities are described as Dialog Atoms (see section **??**).

With the concept of a state machine in mind, additional phases can be added easily in the course of future development to handle more complex sequences.

Solving Subproblems

Subproblems are calculations within calculations; in principle, they do not differ from top-level problems.

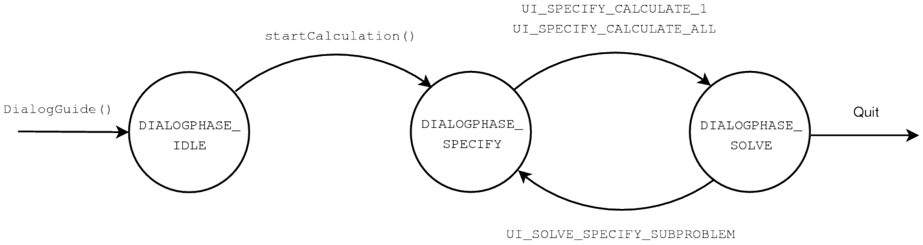


Figure 16.3: A state machine for the Dialog Phases

16.4.2 Dialog Atoms

As opposed to the high-level Dialog Phases, Dialog Atoms are basic building blocks of system-user interaction at the level of a single interaction. For configuring the WorkSheet Dialog's interactional behaviour (UR.6.2.2), we aim at developing an abstract language with Dialog Atoms (UR.6.2.1)as part of the vocabulary. The WorkSheet Dialog could contain an API for programming its behaviour, with a lower-level interface implementing an Interpreter pattern [GHJV95] for Dialog Atoms and a high-level interface implementing the Strategy pattern.

Let us quote [Neu01] again:

> The dialog atoms are the following, ordered by descending 'activity' of the learner: All atoms concern a step from the current formula f applying a tactic *tac* which yields the resulting formula f' (the derivation of f), i.e. $f \longrightarrow^{tac} f'$.
>
> 1. given f , input the next formula f'
> 2. given a partial f (supplied by $\mathcal{ISAC}$), complete f such that it is a derivation of f
> 3. given f , input a tactic *tac* to be applied to f
> 4. given f , select *tac* from a list (supplied by $\mathcal{ISAC}$) to be applied to f
> 5. given f and a partial *tac* , complete the *tac* (i.e. a theorem, a substitution, etc.) such that it can be applied to f
> 6. given f , *tac* , and a partial f' , complete f' such that it is the result of applying *tac* to f
> 7. given f and f' , input *tac* such that f' is the result of f applying *tac*
> 8. given f and f' , select *tac* from a list (supplied by $\mathcal{ISAC}$) such that f' is the result of f applying *tac*
> 9. given f , f' and a partial *tac* , complete *tac* such that f' is the result of f applying *tac*

Note that exchanging the parts of the user and $\mathcal{ISAC}$ in the above proposal yields another set of Dialog Atoms, which can be treated as equivalent from the Dialog's point of view. Taking atom 1 as an example, it is essentially the same whether f is supplied by $\mathcal{ISAC}$ and f' is expected to be input by the user or the user asks $\mathcal{ISAC}$ to derive f' from f . In abstract terms, in both cases one part provides f and the other part is expected to supply f' . For this reason, the WorkSheet Dialog tries to provide symmetric Dialog Atoms and make use of this symmetry in the implementation.

16.5 Sharing the Calculation with other Components

16.5.1 Representing the Model and the Specification

A Model (UR.8.4.2) storing lists of formulas called Given, Find, Where, Relate and a Specification (UR.8.4.3) storing identifications of a Theory, a Method and a Problem comprise all data necessary to specify a calculation to the Math Engine. In the *ISAC* system, this information is called a CalcHead, indicating that every calculation (a subproblem, too) has a header specifying a starting point. Once the Solving Phase of a calculation is started, this information does not change any more.

16.5.2 Representing the Path to the Solution

The path to the result is represented by a tree-like structure (UR.2.2.10), alternating formulas and tactics (UR.2.2.8). There is always exactly one Tactic being applied to a formula. In the course of calculating a result, the structure grows as new formulas are added by the user or the Math Engine.

16.5.3 Treating Subproblems

A subproblem is a calculation within a calculation. As such, every subproblem is preceded by a CalcHead. Once specified by a CalcHead, a subproblem could be treated as a calculation of its own and solved independently of the enclosing calculation. Two possibilities for treating subproblems and feeding their results back into the main calculation were explored.

Independent CalcTrees

Every subproblem could be stored in an independent CalcTree. This would emphasise the fact that a subproblem can be solved independently of the enclosing problem and reflect that fact in the data structure used. As an advantage, every calculation would have exactly one CalcHead and exactly one CalcTree associated with it, representing the data involved in the Specifying Phase and the Solving Phase, respectively. This would allow for clearly separating the CalcHead from the CalcTree thus reflecting the separation of the Specifying Phase from the Solving Phase in the storage of data. On the other hand, such an approach would complicate feeding back the results of a subproblem into the enclosing calculation.

One Common CalcTree

As an alternative, all data of a calculation, including associated subproblems, could be stored in a single data structure, subproblems being stored as branches of the tree. While this approach reflects the fact that a subproblem is part of the enclosing calculation, the distinction between Specifying Phase and Solving Phase becomes blurred, as the CalcHeads specifying subproblems must be stored within a CalcTree. Having subproblems tightly integrated into the enclosing calculation eases using their results.

This approach was chosen for implementation, in part due to the fact that the already-implemented Math Engine stores calculations in a single tree.

16.5.4 Accessing Calculation Data

CalcHead

With the CalcHead having a fixed number of fields, all members can be accessed directly. Wherever components share a CalcHead, the object itself is referenced or passed.

CalcTree

For accessing data in the dynamically growing CalcTree, the Iterator pattern [GHJV95] was chosen for its main advantage of hiding the internal representation of the data accessed. Several reasons suggested hiding the internal representation:

- An Iterator can serve the additional purpose of referencing elements of a calculation.
- An Iterator is likely to be a much smaller object than the calculation it points into.
- Several components residing on different machines imply having to pass information about the calculation across the network. Using compact Iterators instead of the entire calculation would make efficient use of network bandwidth and save computing time needed for serializing large objects.
- At an early stage in design, the final structure of a calculation as stored in the Java-implemented part of *ISAC* was not yet decided upon. Using Iterators made it possible to start development of other components without knowing which data structure would be eventually implemented.
- There was much debate about runtime efficiency versus ease of development in representing a calculation. Hiding the internal representation would allow for implementing efficient data structures at a later time without having to redesign the entire system.
- Neither the WorkSheet Dialog nor the Presentation Layer need to know how a calculation is actually stored. On the other hand, both components are interested in the structure of a calculation as presented to the user. Using Iterators would allow traversing a calculation in a user-oriented manner independent of the actual implementation.

16.5.5 Communicating Changes in the State of Calculation

As a component sitting between the Application and the Presentation Layer, one of the tasks of the WorkSheet Dialog is to propagate information about changes or events in one component to the other. The WorkSheet Dialog intercepts the flow of information and modifies it by implementing *ISAC*'s logic of user-interaction.

Wrapper-based Design

One approach is wrapping the objects representing the calculation - the Calculation Tree and associated objects - into objects with the same interfaces but different behaviour, following the Decorator pattern [GHJV95]. This way, the WorkSheet Dialog can filter information considered not appropriate for being presented to the user by simply removing these data from the representation accessible to the Presentation Layer. For an example, if the user is not interested in the Tactics transforming one formula into another, the Tactics simply do not show up in the representation of the calculation seen by the Presentation Layer. This has the advantage of simplicity - the Presentation Layer and the Application need not consider filtering taking place or even know about filtering at all. Moreover, the same interface can still be used if one would want a system without the intervention of a WorkSheet Dialog for direct communication between the Application and the Presentation Layer.

Event-driven Design

In addition to data representing the state of calculation, there is data representing changes in time. Many of these changes occur asynchronously at unpredictable intervals - such as interactions of the user - or with considerable unpredictable delay after the event that that triggered them - such as results of a lengthy transformation becoming available. This sort of changes is communicated through event messages, with objects interested in such notifications registering as Observers [GHJV95] with sources of events. The main sources of events are the Presentation Layer for user actions and the Application for new information about the calculation becoming available. The WorkSheet Dialog intercepts and filters these messages using the Mediator pattern [GHJV95].

16.6 Configuring the User-Interface

The WorkSheet Dialog and the Presentation Layer have to cooperate closely in user interaction. Consider a button on the screen triggering some action. It is the Presentation Layer's responsibility to render the button and to notice the user clicking it. It is the WorkSheet Dialog's responsibility to decide whether the user is allowed to request such action and to trigger appropriate action in the Application.

The division is not always so clear-cut. Consider internationalisation of the user-interface (UR.2.3.1): Is it the Presentation Layer, which is responsible for rendering in general and the language environment, that decides which text to set on the button? Or is it the WorkSheet Dialog, which knows the meaning of the action triggered by the button? For the following considerations, we will stick to the example of the button.

16.6.1 The Presentation Layer in Control

If the Presentation Layer controls every aspect of a button, such as visual appearance, placement on screen and the actions triggered, everything seems easy. Problems arise if we consider buttons which are needed only in special contexts. A button asking for the next Tactic to be applied to a formula does not make sense during the Specifying Phase, where no Tactics occur.

We could show the button all the time, with the WorkSheet Dialog simply ignoring requests when not appropriate. This has the disadvantage of confusing the user with lots of buttons which can be clicked but make no sense in the current context.

We could show buttons only if clicking them makes sense. If the Presentation Layer were to solve this problem, it would need information about the current phase of the dialog. While this may seem feasible, in other situations the applicability of the button might depend on the user's role or privileges, or on the user's level of expertise, which in turn might change even during a session. Especially if buttons depend on didactic strategies, this involves knowledge which has nothing to do with presentation but is clearly part of the WorkSheet Dialog.

16.6.2 The WorkSheet Dialog in Control

If we put the WorkSheet Dialog in control of the buttons, it becomes easy to solve problems with context, but this would require the WorkSheet Dialog to care about internationalisation and visual appearance, which is out of interactional context and should be left to the Presentation Layer.

16.6.3 Splitting up Responsibilities and Providing for Interaction

It seems best to have the various aspects of a button controlled by the component which possesses the information necessary to do so.

While the Presentation Layer should control every visual aspect of a button such as text, shape and placement on screen, the WorkSheet Dialog should control the context in which the button appears and the action it triggers. See [SBCO01] for the discussion of a related problem.

First attempts aimed at providing means for the WorkSheet Dialog to enable or disable buttons otherwise controlled by the Presentation Layer. In the meantime, the goal changed to having the WorkSheet Dialog control the creation and destruction of elements of user-interaction as well.

The WorkSheet Dialog creates a element of user-interaction by providing an identification of the action it triggers. The presentation layer need not even understand the meaning of the action, it uses the identification merely for notifying the WorkSheet Dialog which action has been triggered by the user. In addition to that, the WorkSheet Dialog provides the Presentation Layer with hints about the context of the user interaction, such as whether the action relates to a single formula or the user's session as a whole. The Presentation Layer can use this information to choose an appropriate visual representation. Moreover, the the WorkSheet Dialog does not even request the trigger to be a button. It requests that a means for the user to trigger a request be created and it is left to the design of the Presentation Layer to offer a button, a menu item or both. This approach bears similarites to the Factory pattern [GHJV95], but the created object remains with the Presentation Layer and is not passed back to the WorkSheet Dialog.

16.7 Obtaining and Storing Configuration Data

Much of the WorkSheet Dialog's behaviour can be parameterised (UR.6.2), and many of these parameters are individual to a user (UR.6.3). Moreover, some of the parameters are modified by the system itself during a session (UR.6.3) based on data collected (UR.6.1).

16.7.1 The User Settings

By user settings we denote preferences on aspects of the system set by the user (UR.6.2.2, UR.6.2.4), such as amount of data shown, levels of difficulty or customisations of the visual appearance of the program. As these settings do not pertain only to the WorkSheet Dialog but also to other parts of the system, they will be managed outside the WorkSheet Dialog. It is to be noted that this information does not change very often, so efficient processing is not an issue.

16.7.2 Permissions and Security Issues

With $\mathcal{ISAC}$ being developed as groupware (UR.7.1.1), special attention has to be paid to the fact that settings will be set not only by the individual user, but also by privileged persons such as course administrators (UR.2.1.4). In addition to that, changing some of the settings may be subject to restrictions (UR.7.2.1) depending on the role of the user. It is assumed that reconciliation of contradictory settings and security issues have already been resolved by user management and that the WorkSheet Dialog has access to the settings in effect for the current session.

16.7.3 The User Model

As required in UR.6.3, the WorkSheet Dialog will adapt to the assumed knowledge and abilities of the user. Decisions about which information to show and which actions to take will be based on an internal abstraction of the experience the system had with the user, the User Model.

The User Model is notified about every interaction between user and mathematical knowledge and information about the knowledge involved (UR.6.3.4). The user's performance (UR.6.3.3) is recorded together with information about the context of the interaction, i.e. the Dialog Atom used in the interaction. For efficiency reasons, data is stored as statistical digest rather than as log. The User Model is accessed frequently, at least once per interaction both for query and for recording the outcome, and logging every event would grow the amount of data processed unmanageable very soon.

Note that the User Model gathers and processes the data, but is not aware of their meaning - knowledge items and Dialog Atoms are processed as identifying numbers. Interpretation of the data is left to the components which use them, i.e. the WorkSheet Dialog and, in the future, the Browser Dialog.

Abstractions on a higher level than the presently used statistics could be added in the future, as cooperations of *ISAC* in the field of didactics could provide abstract measures e.g. for a user's familiarity with a topic. In any case, the User Model provides descriptive information about the user and decisions about further actions are always taken by the WorkSheet Dialog.

As the user's history has to be regarded as well (UR.6.3.5), the User Model will be stored across sessions along with the user's settings.

[8]

16.8 Browser Dialog

The browser dialogs are responsible to process the informations coming from the user interactions on the knowledge browsers (see chap.18) and also for gathering information from the knowledge base or (indirectly over the worksheet dialog) from the math engine.

There is no possibility for the learner to manipute the data presented by the knowledge browsers while using *ISAC*'s tutoring-system. The only way to change data from the KE-store is via *ISAC*'s authoring-system. There are two sources where the browser dialogs fetch their information:

1. When Data has to be fetched from the KEStore (chap.19) it solely depends on the UserModel (the membership to a course, a session within a written examination, etc.) how much informationen if any can be retrieved form the KEStore.

2. Data from the MathEngine (chap.20) which concerns a context to a certain calculation. The context gives a concrete interpretation of the meaning of an item of the KEStore. The user can choose if he wants to switch the context on or off. When the context is switched off only the static information from the KEStore will be presented.

---

[8] End of copy from [Kre05] p.67-76.

In analogy to the worksheet dialog the browser dialog has a central role in *ISAC*'s architecture following the 'Seeheim Model' (see sect.16.2).

There is one dialog for one browser (see chap.18). But the functionality of the four browsers is more different than their layout shows. Thus there is a more sophisticated (subclass-)relation between the dialogs than betwenn the browsers. The dialog can be

- an example browser dialog, which is responsible for starting the execution of examples
- a knowledge browser dialog, which is responsible for displaying the respective parts of the math knowledge; thus there is a
  - theory dialog
  - problem dialog
  - method dialog

The dialog has to handle links from each browser to each other. The dialogs are created at the same time at setup of the session.

16.8.1 Browser Dialog and Worksheet Dialog:

There is four browser dialog for one session, while there may be $0 \ldots n$ worksheet dialogs (according to the number of worksheets opened). The relations between the two kinds of dialogs are discussed in 16.1.

16.8.2 Survey on requirements

[9] The *browser-dialog* is the peer for the browser within the dialog and gathers the informations for the request. Depending of the type of request, it contacts the *KE-Store* and/or the WorkSheet-dialog. Afterwards, the informations are filtered depending a users actual permissions and duties. It is possible, that pages are blocked as a whole — e.g. if an example is not accessible before an other one is solved. Alternatively, some fields might be blocked. e.g. if a learner has to solve a problem while performing a test, he might see all available problems – but the dialog will suppress the fields which tell the user if (and why) an actual problem does not fit.

The permissions are gained from different sources.

- The course-designer may define a group of users which may access an example.
- The course-designer may define preconditions before a example is displayed (e.g. a date, or a set of examples to solve first)
- The course-designer may set the user to use a "temporary environment" which alters all permissions to a set of *KE-Objects*(e.g. while an exam)

---

[9] This is a survey from an early design phase documented in [Gri03] p.42

- The dialog-layer maintains a user-history to find out which examples and parts of the knowledge are already visited and solved.

The informations about a user are stored within the user-model. Informations like the history are not only gathered and used by the browser-dialog but by the whole dialog-layer.

16.9 Dialog Guide and User Model

The term 'Dialog Guide' addesses an abstraction which comprises both, the Worksheet Dialog and the Browser Dialog. The Dialog Guide shall be subject to simple implementation and parameterization of 'intelligent dialog behaviour' by 'dialog authors' in the future.

There is a detailed discussion in sect.15.2.1 p.77 about the relations beetwenn the dialoges an the UserModel within the context of session management.

Chapter 17

Worksheet

The worksheet is the center of user interaction in the *ISAC* system. From the Users point of view, the Worksheet is the place where the calculation happens. The *ISAC*'s knowledge base is constantly expanding and the user interface has to be flexible enough to allow the user to use every function provided by the mathematics engine.

17.1 The Presentation Model

As already mentioned when designing *ISAC*, the entire was split into components according to the Seeheim Model. The Worksheet component according to this model represents the Presentation Layer. According to Seeheim Model, the Presentation Layer is the component that serves for the simultaneous interactive communication with the user. This component defines the user interface on a lexical level by specifying the user interface widgets presented to the user. These user interface widgets serve the purpose of supplying output to the user and receiving input from the user.

17.2 Communication between the Presentation and Dialogue Control Layer

As mentioned in the previous section, the main purpose of the Worksheet is to translate the user interactions to the requests to the system. In this section we shall discuss how to implement the communication between the Dialogue and the Presentation Layer. To further simplify things, the communication is divided into two categories:

- User Interface Events, which consist of user requests and system messages
- Calculation Events, which represent the output of the mathematic engine

17.2.1 User Interface Events

The most common way to implement this type of communication is to hold a reference to the system interface of the Worksheet Dialogue in the Worksheet and for every user interaction event, simply call specific methods of the interface to execute the command. However, the designers of *ISAC* had to take in the consideration that the Worksheet and the rest of the system are two different and separate applications, meant to run on different computers and communicate over network. Second consideration was that the Worksheet application does not necessary need to be an AWT or SWING application, but could be a pure text mode interface or even a 3D enhanced virtual reality interface. Additionally, the components had to be as flexible as possible, but the interface should remain the same, thus allowing for the separate development of Presentation and Dialogue Layer.

The logical answer was to use the "command objects" as a practical mean for network transport, because they can be easily serialised. The receiving module of the system has only one method for communication that handles all user interaction events. For each event this method receives the appropriate "command object" as the parameter and acts accordingly. This keeps the interface between the objects simple and stable.

In case of *ISAC* Worksheet the "command objects" are called Actions. The Actions can be sent from Worksheet Dialog to Worksheet or vice versa. The actions sent from the Worksheet Dialog to Worksheet are called UIActions and represent different User Interface Elements the Worksheet needs to present to the user. When the Worksheet receives the UIAction, it adds a widget to the User Interface. To each of these widgets a certain UserAction is assigned. When the User activates the widget, the Worksheet makes sure that the proper UserAction is sent to the Worksheet Dialog. The Worksheet self has no knowledge about the meaning of the Action.

As already mentioned in this document, there is a strong discrepancy in the architectural requirements set for the system. The Presentation Layer should have no knowledge of the state in which the dialogue currently is and the Dialogue Layer should have no knowledge over such things as placement and internationalisation. The solution is to establish a certain hierarchy within the set of UIActions. This hierarchy lets the Presentation Layer choose the widgets for representation, but leaves the possibility for the Dialog Layer to choose the circumstances in which the widget is available. Certain actions are always available, like the actions applied to the entire calculation, whereas other actions can only be performed on a single (or even specific) formula, like changing the tactic or assumption and are available only in certain dialogue phases.This hierarchy is implemented in *ISAC* with "contexts". For example the actions applied to the entire calculation have a certain context and all such actions are rendered as push-buttons in the upper part of the worksheet.

It is important to note that according to Seeheim Model, the Presentation Layer is not responsible for the context management, so the context of a certain action is sent to the Worksheet as a part of the UIAction object.

17.2.2 Calculation Events

Second part of the communication between the Worksheet and the mathematical engine are the calculation events. They are fired each time the mathematical engine has some data for the worksheet. These events are handled asynchronously in a separate method of the Worksheet. The asynchronous communication can have its disadvantages. If the mathematical core processes several different calculations from several users, the user may have to wait several seconds for the result to appear. This can sometimes be a serious usability problem. However, the major advantage is that the asynchronous message passing allows for more parallelism.

17.3 Calculation views

According to the User Requirements, the user can start using the Worksheet in two different modes:

- the specifying phase
- the solving phase.

From the Worksheet of view the two modes are the same (like a blank piece of paper). The Worksheet has no knowledge of the current phase.

17.4 Calchead Panel

If the CalcHeadPanel is displayed because the user started a calculation from scratch, then the user first has to *model* the problem. As specified in the User Requirements, the Calculation Model is made up of the following fields :

- 'given' which includes the input-items
- 'where' which includes the pre-condition on the input-items
- 'find' which includes the output-items
- 'relate' which includes parts of the post-condition.

To provide a full specification, the user must also make inputs about the problem type, the theory that is necessary to solve the problem and the method on how to solve the problem. Each input value is checked by the mathematical engine. If an input value is incorrect or cannot be handled by the mathematical engine, then the user has to be informed about this fact. So far *ISAC* can only inform the user that something went wrong but is not in the position to provide help to correct the problem.

If the user wants to refine a problem or to match a problem, then the model of the current example and the specified problem from the problem hierarchy are compared against each other by the mathematical engine. The task of the CalcHeadPanel is to colorize the fields which are either correct (match) or not correct (do not match).

Implementation Details

The CalcHeadPanel helps the user to model and specify a problem. If the user decided to calculate a prepared example from the example hierarchy then the CalcHeadPanel is already filled with values that are stored in the XML file that represents the example (see listing below) otherwise the user has to input all values that are necessary for a complete formalization of an example.

```
...
 < DESCRIPTION >
  < FORMALIZATION >
    < VARIANT >
      < TERMLIST >
   < MATHML >
     < ISA > equality (1+-1*2+x=0) < /ISA >
   < /MATHML >
   < MATHML >
     < ISA > solveFor x < /ISA >
   < /MATHML >
   < MATHML >
     < ISA > solutions L < /ISA >
   < /MATHML >
      < /TERMLIST >
      < SPECIFICATION >
   < THEORY > Test.thy < /THEORY >
   < PROBLEM >
     < KEY >
       < ID > linear < /ID >
       < ID > univariate < /ID >
       < ID > equation < /ID >
       < ID > test < /ID >
     < /KEY >
   < /PROBLEM >
   < METHOD >
     < KEY >
       < ID > Test < /ID >
       < ID > solve_linear < /ID >
     < /KEY >
   < /METHOD >
      < /SPECIFICATION >
      < HIDE > < /HIDE >
      < DETAIL > < /DETAIL >
    < /VARIANT >
  < /FORMALIZATION >
</EXAMPLE>
```

It is very cumbersome to input a complete formalization; therefore, the CalcHeadPanel has views with different detail levels for the formalization of an example: A `FullCalcHeadView` in which all items of a formalization have to be input and a `SimpleCalcHeadView`. The `SimpleCalcHeadView` provides a view that is similar to existing algebra systems where the user has to input only the example he wants to calculate (e.g. $x + 1 = 2$). At the time of this writing $\mathcal{ISAC}$ is only able to provide the `SimpleCalcHeadView` for equations.

As soon as a complete and correct formalization has been entered, $\mathcal{ISAC}$ will be able to calculate the formalized example and can go into solving phase.

Communication with the Calchead Panel

The Calchead Panel is an integral part of the Worksheet and communicates with the Worksheet Dialog through aforementioned User Interface and Calculation Events.

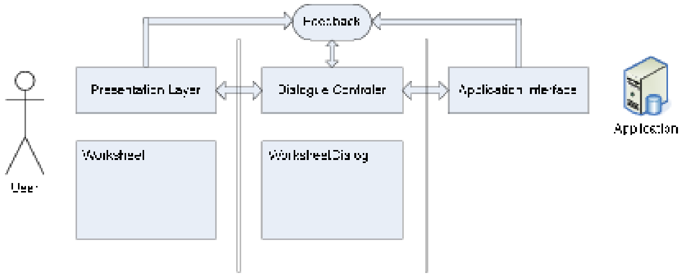
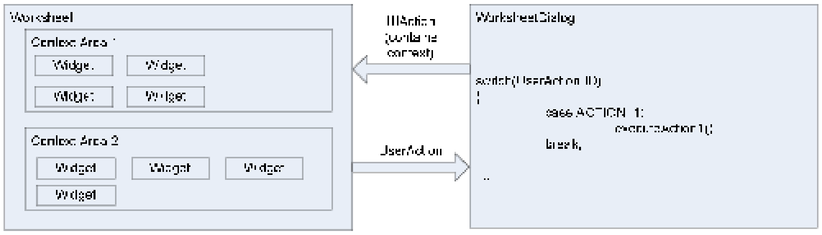


Figure 17.1: Worksheet as a part of Seeheim Application Model



```
switch(UserActionID)
{
        case ACTION_1:
                executeAction1();
        break;

 ..
```

Figure 17.2: User Interface Events

Chapter 18

Knowledge Browser

18.1 Survey on the requirements

[Gri03] p.38 gives the following survey on the requirements as seen from an early stage of design.

A *ISAC-knowledge-browser* is used to enter the *knowledge* and gain information about the content and use of the special *ISAC* System. The the design is split into layers which separate the functionality of knowledge-gathering and knowledge-presentation.

A few points to keep in mind to understand the structure of the design.

- multiple layer structure to ease exchangeability of the knowledge-presentation
- Depending on their type, information is structured to different hierarchies. An user can browse through the problems to gain an overview about the capabilities of an *ISAC* site, or view the provided examples. Although this views might look logically different for the user, the same software is used for all of them.
- A browser can either deliver static informations or react interactively on each request. Both modes require a filtering-mechanism to adjust the delivered data. This mechanism is located inside the *dialog*. Reasons to block information include security reasons (block information while an exam is in progress) as well as educational reasons (do not swamp a user with examples he is not expected to solve)
- A browser has different “roles” e.g. “find a problem” or just browsing through the knowledge or the examples.
- The different roles require interaction with other *ISAC* modules. This interaction is established and controlled by the dialog.
- the user can open a worksheet by clicking a link in the browser-window. This call for a worksheet is tunneled through *ISAC*s dialogs to avoid direct

communications between frontend-applications. On the other way round, the worksheet might call an browser (e.g. to find a matching problem) – this leads to a new BrowserWindow which is created by the dialog.

18.2 Kinds of browsers and their differences

The task of the browsers is to provide acces to the KEStore, see chap.C.0.5 and reflects the

The browsers for the four parts of knowledge, for examples, theories, problems and methods look very similar: they display a hierarchy containing all elements of the respective part and an element if selected in the hierarchy.

The browsers all work very similar; the most significant difference is in the interactions available (for instance, there is a <Refine> button on the Problem-Browser only). But as the front-end has no idea about the meaning of buttons and other interactive elements, the difference can be neglected.

[1]

18.3 Browsers and dialogs

The architecture of the browsers and their dialogs is shown in figure 18.1 and 18.2 and shall be discussed here. Note, that not all attributes and methods of the classes are shown in the diagrams for simplicity.

The browsers are built totally equal. The way, elements of the knowledge base or the example collection are presented should be equal anyway. The only thing that differs is the behavior of some elements (e.g. the buttons) and their content, they are displaying. The browser itself does not care about the meaning of the elements it is showing. The meaning of the elements is only known by the dialog, controlling the browser. That fact, that the browsers are all built equal can be taken as an indication that the representation of the elements is separated well from their meaning.

18.3.1 Communication between Browsers and Dialogs

The browser and the browserdialog communicate over Java RMI (Remote Method Invocation) [**?**]. This is a Java technology to call methods of objects, which do not have to run on the same Java Virtual Machine, they do not even have to run on the same physical device. Whenever information has to be passed from the browser to the dialog, the `notifyUserAction()` method of the `IBrowserDialog` interface is used to pass a `UserAction`. There are different kinds of `UserActions` carrying different information. Whenever information has to be passed from the browser dialog to the browser, the `IToGuiInterface` implemented by the `BrowserFrameRMI` is used to hand over a `UIAction`. The `UIAction` carries a `EUIContext`, which is implemented as enumeration. It determines, which

---

[1] Begin of [Kö06] p....– p....

component of the browser the `UIAction` concerns. The `UIAction` is forwarded to this component, where it is finally handled.

18.3.2 Binding a Browser to a Dialog

The binding of a browser to the according browser dialog happens after login. The `login()` method of the `UserManager` creates a new `Session` and returns an interface to this session. The `WindowApplication` registers to the session by use of the `registerBrowserFrame((IToGUI) this)` method, passing itself. The session does now execute the `initializeWindowApplication()` method which calls the `openNewBrowserFrame()` method for all four kinds of browser, passing the according dialog. A new `BrowserFrame` is created, taking the according dialog as argument of the constructor. The `registerBrowserFrame()` method of the dialog is called to register an interface to the `BroserFrame` for the dialog.

This explanation might sound frightening at first. However, it leads to a clear separation of the tasks. The browsers do not need to know anything about the dialogs except of an interface to pass `UserActions`. The dialogs do not need to know anything about the browsers except of an interface to pass `UIActions`. The `WindowApplicaion` does only know the browser but not the dialogs. The session does only know the dialogs but not the browsers themselves.

18.3.3 The Processing of Links

Whenever a link is selected, the `Minibrowser` creates a `UserActionOnLink`. How this is actually done, and how the `Minibrowser` works, can be read in 26.4. The `UserAction` is sent to the according dialog, where it is evaluated.

Whenever a dialog decides to display a link target in its registered browser, an `UIActionOnLink` containing the link target is sent to the `BrowserFrameRMI`. The `EUIContext` of this object is set to UI_CONTEXT_MINIBROWSER, so it is forwarded to the `Minibrowser`, where the page is finally loaded.

[2]

---

[2] End of [Kö6] p....– p....

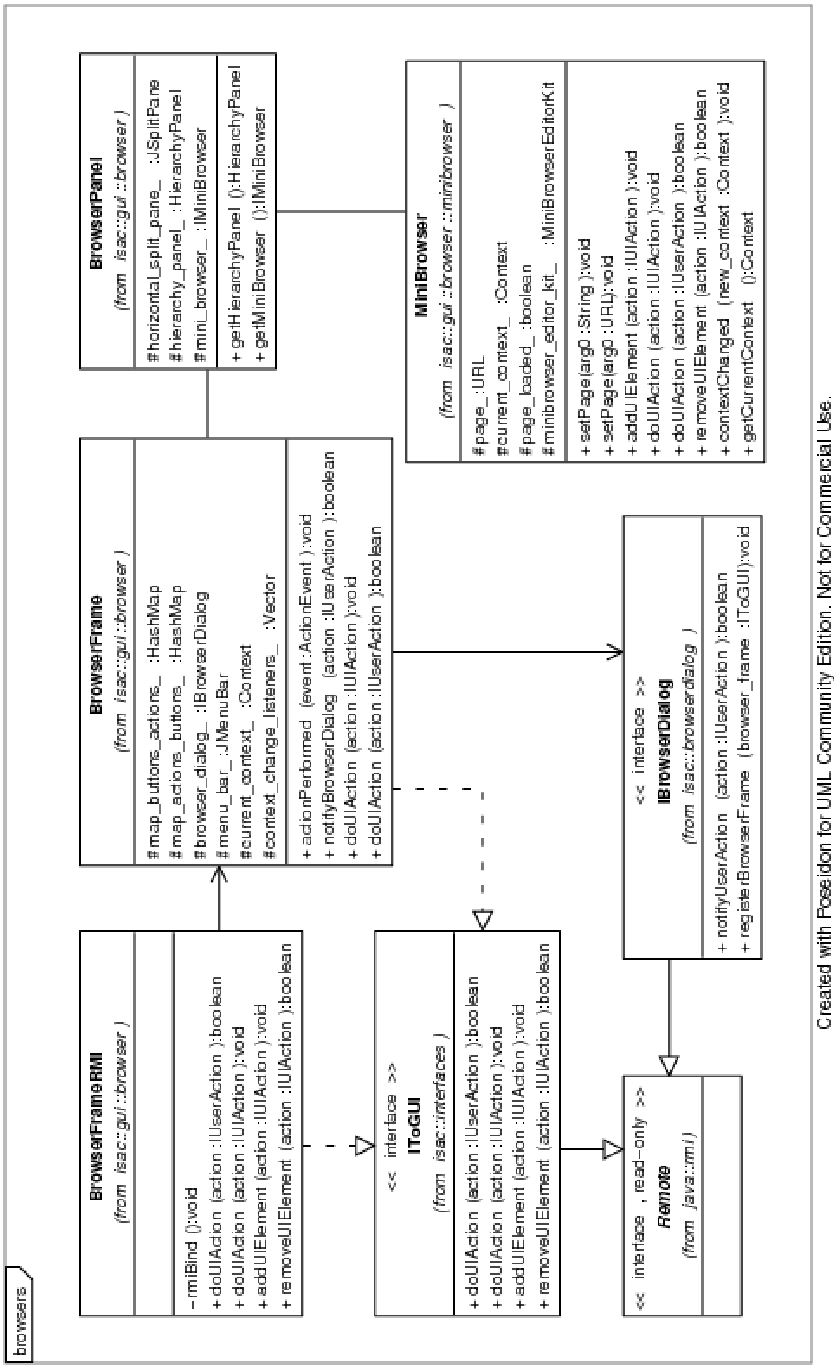
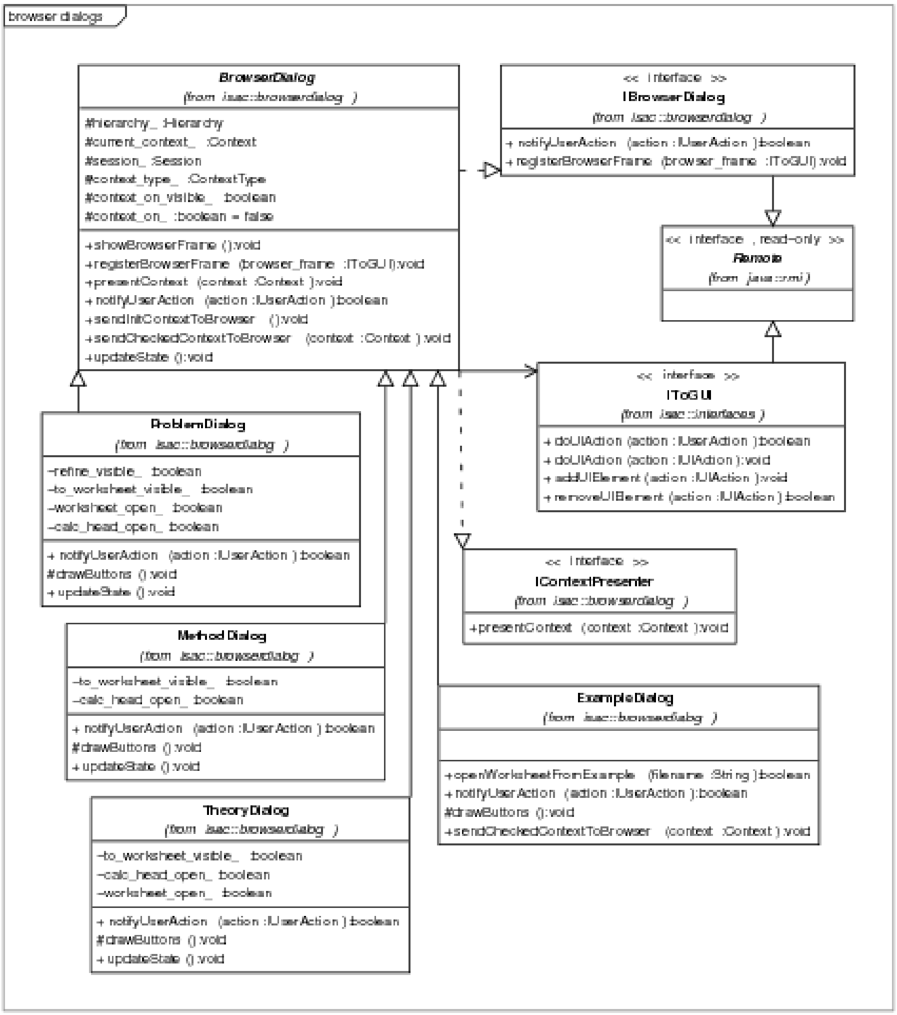

browsers
BrowserFrameRMI
(from isac::gui::browser)
- rmiBind ():void
+ doUIAction (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ addUIElement (action :IUIAction):void
+ removeUIElement (action :IUIAction):boolean
BrowserFrame
(from isac::gui::browser)
# map_buttons_actions_ :HashMap
# map_actions_buttons_ :HashMap
# browser_dialog_ :IBrowserDialog
# menu_bar_ :JMenuBar
# current_context_ :Context
# context_change_listeners_ :Vector
+ actionPerformed (event :ActionEvent):void
+ notifyBrowserDialog (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ doUIAction (action :IUserAction):boolean
BrowserPanel
(from isac::gui::browser)
# horizontal_split_pane_ :JSplitPane
# hierarchy_panel_ :HierarchyPanel
# mini_browser_ :IMiniBrowser
+ getHierarchyPanel ():HierarchyPanel
+ getMiniBrowser ():IMiniBrowser
<< interface >>
IToGUI
(from isac::interfaces)
+ doUIAction (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ addUIElement (action :IUIAction):void
+ removeUIElement (action :IUIAction):boolean
<< interface >>
IBrowserDialog
(from isac::browserdialog)
+ notifyUserAction (action :IUserAction):boolean
+ registerBrowserFrame (browser_frame :IToGUI):void
MiniBrowser
(from isac::gui::browser::minibrowser)
# page_ :URL
# current_context_ :Context
# page_loaded_ :boolean
# minibrowser_editor_kit_ :MiniBrowserEditorKit
+ setPage(arg0 :String):void
+ setPage(arg0 :URL):void
+ addUIElement (action :IUIAction):void
+ doUIAction (action :IUIAction):void
+ doUIAction (action :IUserAction):boolean
+ removeUIElement (action :IUIAction):boolean
+ contextChanged (new_context :Context):void
+ getCurrentContext ():Context
<< interface , read-only >>
Remote
(from java::rmi)
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 18.1: The Design of the Browsers


browser dialogs
BrowserDialog
(from isac::browserdialog)
#hierarchy_ :Hierarchy
#current_context_ :Context
#session_ :Session
#context_type_ :ContextType
#context_on_visible_ :boolean
#context_on_ :boolean = false
+showBrowserFrame ():void
+registerBrowserFrame (browser_frame :IToGUI):void
+presentContext (context :Context):void
+notifyUserAction (action :IUserAction):boolean
+sendInitContextToBrowser ():void
+sendCheckedContextToBrowser (context :Context):void
+updateState ():void
<< interface >>
IBrowserDialog
(from isac::browserdialog)
+notifyUserAction (action :IUserAction):boolean
+registerBrowserFrame (browser_frame :IToGUI):void
<< interface , read-only >>
Remote
(from java::rmi)
<< interface >>
IToGUI
(from isac::interfaces)
+doUIAction (action :IUserAction):boolean
+doUIAction (action :IUIAction):void
+addUIElement (action :IUIAction):void
+removeUIElement (action :IUIAction):boolean
ProblemDialog
(from isac::browserdialog)
-refine_visible_ :boolean
-to_worksheet_visible_ :boolean
-worksheet_open_ :boolean
-calc_head_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void
<< interface >>
IContextPresenter
(from isac::browserdialog)
+presentContext (context :Context):void
MethodDialog
(from isac::browserdialog)
-to_worksheet_visible_ :boolean
-calc_head_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void
ExampleDialog
(from isac::browserdialog)
+openWorksheetFromExample (filename :String):boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+sendCheckedContextToBrowser (context :Context):void
TheoryDialog
(from isac::browserdialog)
-to_worksheet_visible_ :boolean
-calc_head_open_ :boolean
-worksheet_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void

Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 18.2: The design of the Browserdialogs

Chapter 19

KE-Store

19.1 Notes WN

These notes are an *un*-structured collection of (user- and software-) requirements and (architectural- and software-) design consideration[1].

- The (K)(E)(Stor)e encapsulates a database (stor)ing *ISAC*s mathematics (K)nowledge and the (E)xamples.
- The elements of the knowledge are given by the need of *ISAC*s mathematics engine to automatically solve the examples in the KEStore.
- See the user requirements in sect.4.4.1
- The hierarchy of examples can be rearranged for specific courses
- Each element of the KEStore knows the respective path in the (current – examples!) hierarchy.
- The elements of the KEStore store sufficient information on formatting for automated generation of the presentation of single elements as well as for collections of elements.
- The generation of the presentation format is located in the system as close to the front-end as possible (in order to feature dfferent formats like html, pdf etc).
- The access to the database (internal to the KEStore) is encapsulated by methods of the KEStore. The arguments carry the course (the student is assigned to), the time (for time constraints during exams) etc. The return values of these methods may be up to additional filtering and subsequent calls of specific methods.

---

[1] These notes where compiled as a prerequesite for re-designing the dialoguies in the *ISAC* summerterm 06

-
-
-

19.2 The initial structure with xml- and html-files

19.3 Some old design considerations

[2]

This module is used to store and provide all static informations within *ISAC*, a user can obtain. Informations are supplied to the *KE-Store*-module by either importing them from XML or entering them directly through an *ISAC*-KE-Object-editor

Parts of the content contain mathematical informations (*formalizations*) along to *descriptions* and *explanations*. Though the details of attached informations vary, all objects are handled in an similar way. Therefore, the umbrella term *KE-Object* is introduced. Different types are described in the following:

simple KE-Object Is a collection of informal data without any mathematical information (no *formalization*). It can be used to build up explanations of any content.

Knowledge *ISAC* *knowledge* is taken from the mathematics-knowledge-base and consists solely of formal information. There are different types of mathematical knowledge which build up the three dimensions of the *ISAC*-KB.

This informations reflect the knowledge of the math-engine, but a possible change to the informations will not change any informations within the KB! Therefore, *knowledge* informations have to be immutable to avoid ambiguities[3].

Decorated Knowledge a course-designer or maintainer of an *ISAC* site can add a informal *explanation* to the mathematical knowledge. This combination is called *decorated knowledge*

Examples An *example* consists of the an informal *description* to tell the user what to do and an formal representation (*formalization*) to enable the MathEngine to solve it.

There is a special form of examples called *composite-examples* which are used to describe examples which are partitioned to several parts (a., b., c., …). More details on that in subsection 19.3.5

---

[2] Design considerations from [Gri03] p.43-53.

[3] *knowledge-objects* are synched with the *KnowledgeBase* using unique identifiers

Example Collections As the name suggests, a *example collection* is a object to combine a number of examples to collections. It does not contain any mathematical informations. An example can appear an arbitrary number of example-collections.

All objects can contain additional informations like images or descriptions to enhance the understandability. These infos are attached using the *presentation* and reference to extra data (images).

hierarchy: *KE-store-object*s are used to build up the learning-environment for the user. The compilation of the proper parts is achieved using the *hierarchy-object* which combines the single items to an hierarchy and provides the basis for a tree-like navigation through the content.

There are some standard-hierarchies which reflect the structure of the knowledge of the MathEngine. These hierarchies are maintained by the math-engineers within the SML-part of the system and only synched with them. Vice versa, a change on the explanations will **not** change the knowledge! Therefore these hierarchies may not be changed (like the *KE-objects*, the “nodes” of these hierarchies)

19.3.1 XML-Import/Export

KE-Objects can be imported and exported from and to XML. They contain informal data which are enriched by a set of tagged formulas which contain the mathematical meaning necessary to solve it. This summary of data can be transferred to the MoWGLI XML-format which provides a machine-readable and machine-*understandable* way to present mathematical documents. Export can also be used to transfer the informations to an other $\mathcal{ISAC}$-engine (the destination $\mathcal{ISAC}$-system has to have the necessary mathematical capabilities which can not be ensured by the document-format.

The different types can be treated in very similar ways – they differ only in the amount of given data. In case of an import of *knowledge*, the resulting *KE-Object* has to be write-protected to avoid any alteration.

Objects like *problems* and *examples* contain a formalization which consists of tags and related pieces of information – most likely formulas. These formulas are also encoded in MoWGLI. The tags have to be preserved because they give the meaning to the formulas.

Parts of the content (like images and links to related examples) are implemented using textual references. So, unique identifier for all KE-Objects have to be used to store this relations in an textual form.

Note, that the dialog can add automatically generated references which are not part of the KE-Object and though not stored within the export-file.

The *hierarchies* have a different structure than KE-Objects and are stored in own documents. Again, unique identifiers are necessary to restore the references. When an hierarchy is exported, its referenced KE-Objects can be exported in own files in the same operation.

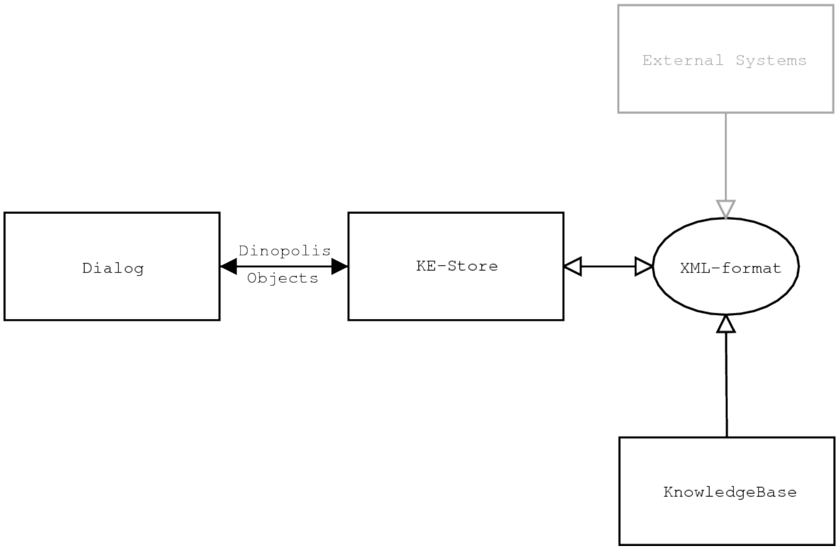


Figure 19.1: parts of the *KE-Store* module

19.3.2 Communication with the *dialog*

To communicate with the rest of the system, the informations are put into a hierarchy of (*dinopolis*)-objects which are transferred to the dialog (again using *dinopolis*). These objects can cope with the dialogs need of restricting several kinds of details. The object-structure is finer grained than the junks within the underlying XML-Structure. e.g. it keeps its formulas in own objects.

These objects also can be modified (with sufficient access-rights) by an example-author or a course-designer. Also, new *KE-Objects* can be created. Several constraints on edition of examples have to be mentioned.

- there is a maximum of one client at a time which is allowed to alter an object. This is because if two editors alter an object concurrently, they would overwrite each others changes. (A lock is required)
- if an object is changed, all users of the object have to be informed to update their representation.
- the structure of the knowledge may not be altered. These informations are stored within the SML-part and are only changed by the math-engineers. To keep the storages in sync, unique identifiers have to be used.

The object-representation can also be generated out of XML-files without storing them within the *KE-Store*-module. The interface to the Dialog keeps the same but without the ability to change the information. A module like that can be used to calculate an external example "on the fly" (e.g. out of the ActiveMath system) if it contains the proper formalization.

19.3.3 Relations

Contained items can be interlinked in various ways. On the one hand, there are hierarchical structures which can be used to provide an "navigation-tree". A structure like this is given by the *problem*-tree. Each node (except the root-node) has an "ancestor" and an arbitrary number of child-nodes. In case of the mathematical data, these structures are given and may not change, in case of an example-hierarchy, a course-designer may change them - respectively each course has a own hierarchy (an example can appear on different places on different hierarchies)

On the other hand, relations can be located within an object. They are part of the object. For example if a *KE-Object* contains links to related examples.

Both types of relations can be suppressed by the *dialog*. Respectively it can choose which ones it shows. Again, this decision depends on a users experience/permissions and a course-designers intentions.

Relations within $\mathcal{ISAC}$ are realized as dinopolis-references. When they are exported, they are stored in their unique textual-representation.

19.3.4 Presentation

The Presentation has to meet several requirements depending on the object to show. While *problems* (like the other automatically created informations) have a uniform appearance and a known set of informations, *examples*, *simple KE-Objects* and especially *example_collections* have a higher need of individual layout. The requirements are even hardened because of the dynamic parts of this informations.

The uniform presentation of the mathematical data can be left to the browser which transforms the data using templates.

Other data has to be enriched with layout-hints which have to be preserved on its way through the dialog. This can be accomplished by adding an layout-template with "slots" to fill the informations in. If an detail is suppressed by the dialog, default-values have to be given.

This template has to be created by the example-, respectively the KE-Object-editor which is the only one who knows how to present its informations in the proper way.

It is important to mention that presentations can be "nested". This means, hat a document is created out of a number of objects. This mechanism is obvious when we look at a *KE-Object* which contains formulas.

The enclosing object is the description which contains both text blocks and fields which contain references to the formula and the image. When the document is to be displayed (creation of am HTML-Document), the formula is asked for its presentation (which most likely results in an MathML - text).

This mechanism is expandable to more general cases – *KE-Objects* can be used to set up the layout by just generating the frames to put other object presentations in. Objects like this will be used to give an visible context for

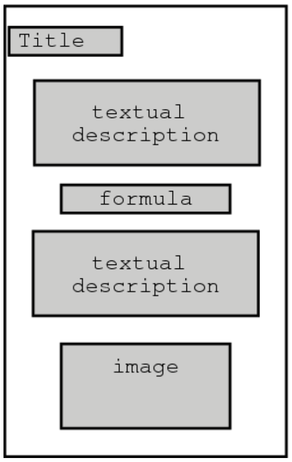


Figure 19.2: nested presentation

other objects like e.g. to summarize different sections to one page or building up an *example-collection*[4]

To accomplish this behavior, the layouter has the option to either create an link to an embedded object, or to make it “in-line”.

19.3.5 Example Collections and composite examples

Example collections are used to summarize related examples to one page. This page can be designed to match a given layout – e.g. to ease the visual matching between an example-subsection in a textbook and the presented corresponding *ISAC*-page. The contained examples are stored as references to them.

An example-collection does not have any mathematical informations about its content - it is just used to summarize and layout a set of related examples.

Composite examples go a step further. A composite example is for instance a text example which defines a task and a set of sub items (a.), b.), c.)) which define what to solve.

Examples like this are split into two parts:

description Describes the task of the example in an informal (usually text, formulas and images) way. It does not have any formal information about the task it describes. The full *formalization* is stored within the sub-examples. Though, the only difference to a *simple KE-Object* is the reference to its sub-examples.

sub-examples One description can be used for one or more sub-examples which contain formalization. Although there are common parts in sub-examples with the same description, the sub-examples contain the full

---

[4] more details on example collections and composite examples see subsection 19.3.5

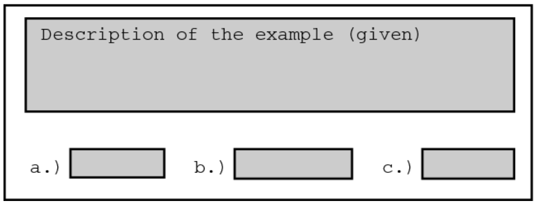


Figure 19.3: composite examples

formalization. The sub-example can be calculated from the ME without interference with an other object.

In opposition to a simple example-collection, a nested example needs fix links between the task description and the sub-examples. This is because the informal description of these objects is split.

When an sub-example is displayed, it cares sole for its own informations – not for the informations displayed by the enclosing description. In figure 19.2, the enclosing object cares of the position of the example-description, the positions of the sub-example-descriptions and the numeration. The fields for the sub-examples are filled with the sub-examples presentation. If the designer of an example-collection wants to limit the shown sub-examples he can do this by setting up the dialog to suppress the unintended references (see subsection 15.1).

19.3.6 Object structure

We have seen in the last sections that – although the informations to present look very different at first glance – we finally achieve very similar needs.

All objects might have a *presentation* which describes the way they *want* to be displayed. This presentation is *not* obligatory – a displaying device (browser) might choose to ignore this layout. (respectively it might be incapable to follow the hints given by the presentation). The layout built up by presentation contains fields which are to be filled by other objects. The names of this fields are also given within the list in the *additional* field. If a output-device ignores the presentation, at least the order of this list should be preserved. Listed elements also act as keys in the object to map to the final content.

The *formalization* contains a mathematical presentation of the object if needed. The content of this field is given to the *worksheet* respective the *math-engine* if there are some calculations to perform on this object.

Parent and *childs* are used to build up a content-dependent hierarchy. With content-dependent is meant, that the content of an object is not complete without the content of the referenced object (e.g. examples).

Meta contains meta-information like severity or target group. More details on this are given in subsection 19.3.7, *metadata*

```
object
    presentation:
    additional:
    formal_data: isac-specification & formalization
    parent:
    childs:
    meta: <metadata>
    *<items listed in additional>:<values>
```

simple KE-Object The *simple_KE-Object* consists solely of a presentation and the references of the therein used fields. Most likely, these references will lead to images and formulas to display. *Formalization*, *parent* and *childs* stay empty.

```
simple_KE_object
    presentation: XML
    additional:[KE_object (ID), ...]
    formalization: empty
    formal_data: isac-specification & formalization
    parent: empty
    childs: empty
    meta: <metadata>
    *<items listed in additional>:<values>
```

mathematical object Mathematical objects are the way to represent formal informations of the *mathematical knowledge base* within the KE-Store. They are generated through an automatic import. Although there is the possibility to attach informations depending their presentation, usually this fields will stay empty and the task of presentation is left to the output-modules which utilize standard-templates for them.

```
knowledge_object
    presentation: empty
    additional: empty
    formal_data: isac-specification & formalization
    parent: empty
    childs: empty
    meta:<metadata>
```

simple example The *simple example* extends the capabilities of the *simple KE-Object* by adding an example formalization. The *parent* and *child* fields are left empty.

```
eimple_example_object
    presentation: XML
    additional: [KE_object (ID), ...]
    formal_data: isac-specification & formalization
    parent: empty
    childs: empty
    meta: <metadata>
    *<items listed in additional>:<values>
```

composite example - task-description *composite example*s are the most complex objects within the *KE-Store-module.* They consist of a *presentation* which describes the general task (a train drives with a speed of 160 km/h from A to B ...) as well as hierarchical informations. Instead of a formalization, it contains links to its sub-examples in its *childs* field. This object is just a container for its sub-examples.

```
KE_object
    presentation: XML
    additional: [KE_object (ID), ...]
    formal_data: isac-specification & formalization
    parent: empty
    childs: [composit-example subexample (ID), ...]
    meta: <metadata>
    *<items listed in additional>:<values>
```

composite example - sub-example This object builds the counterpart of the *task-description.* Its *presentation* contains only information about the sub-example (most likely the “question”:”when will the train arrive at B”). The *formalization* field contains the full formal description.

```
KE_object
    presentation: XML
    additional: [KE_object (ID), ...]
    formal_data: isac-specification & formalization
    parent: composit-example task-description (ID)
    childs: empty
    meta: <metadata>
    *<items listed in additional>:<values>
```

Its theoretically possible to build a deeper hierarchy of *composite examples.* In this case, the informations about a sub-example are gathered by traversing all parents till an object with *parent=empty* reached.

hierarchy

All objects within the *KE-Store-module* can be arranged within a hierarchy to let the user navigate through them. This hierarchy is stored within an own *hierarchy-object* which is different to the other items in the *KE-Store-module*.

Like the other objects, it does contain metadata which describe the purpose of the hierarchy. All other entries serve the representation of the structure.

Every *node* contains name, value and childrens. The value is a reference to an *KE-Object* which has to be called when the node is selected. *name* is a string to show when the node is displayed. *children* is a list of *node id*s where a node id is an identifier which uniquely identifies one node.

The root node has a predefined *node id*. Each node id may only occur once within an hierarchy.

```
hierarchy:
   metadata: (name, target group, ...}
   rootnode: {name:string, value:ke-object-id,
              children[nodeid, nodeid, ...]}
   <nodeid>*:{name:string, value:ke-object-id,
              children[nodeid, nodeid, ...]}
```

The *KE-Store* can import and export a complete hierarchy using one file for the hierarchy and one for each used *KE-Object*.

Assembly of the nodes could also happen in an nested way like usual in XML. The advantage of the presented version is that contained nodes can be found at first glance – this is useful to restrict access to an “temporary environment” as proposed in section **??**.

19.3.7 Metadata

All objects contain a field called *meta* which contains key-value tuples for various informations. This informations have to serve different purposes:

copyright issues This type of fields contain informations which have to be considered when the object is displayed or exported (for the purpose to pass on to other systems). Among others, this fields can contain informations about the *author* and *usage-permission*.

hints for the dialog As mentioned, the dialog has to decide if and when an object is to be displayed or filtered. *metadata* is used to assist this decisions by giving informations about targeted users and difficulty of an example. This hints are not the only mechanism to determine if an object is to be displayed – a more detailed description of this is given in subsection 15.1, *dialog*

search hints Metadata can also be used by search-engines to categorize an objects content. This fields should also be passed on to an searchable output-device like an HTTP-based Browser.

administrative data like date of creation, last change, ...

Most data fields do not belong to only one of this groups but are used for different purposes. For instance although an *author* is mainly used to give information about the copyright, it might also be the subject of an search process.

There are a number of meta-schemas which are used to classify informations on the web[5].

[6]

19.4 KE-Objects and external Informations

[7]

*ISAC*gets its math-descriptions out of the *KE-Store*-module which processes the stored informations and builds an object-structure for the dialog to work with. The process of building up the internal structure can also be based upon well structured Documents like MathML and OMDoc. Vice versa, an export into an XML-format is necessary to utilize the rich set of already built utilities. The first process is used to import informations from external information-sources like ActiveMath. The export to XML can be used to render the output for representation as used for the Browsers.

questions his thesis) to answer: (WN: stated by AG at finishing his thesis)
Has user *user1* permission to read/write an object ?
Who is member/admin of course *course1*

short USE-Case: fetch a course-startpage

- Browser wants to enter a course. he knows the
    - name (ID) of the user
    - ID of the course to display
- browser asks the dialog for the startpage of the course
- dialog determines the hierarchy of the course
- dialog fetches the informations from the KE-Store using the users permissions (exception if the permissions do not suffice)
- dialog filters the hierarchy according to the user-model and delivers it to the browser-dialog

---

[5] See also section **??**

[6] End of copy from [Gri03] p.53

[7] Begin of copy from [Gri03] p.59

- browser removes links to ungranted KE-Object and creates the hierarchy (tree)
- browser asks for the startpage (rootnode)
- dialog fetches it (exception if mot granted) and filteres it according to the user-model
- browser removes links to ungranted KE-Objects and builds the presentation (accessing the presentation-fields - exception if not granted)
- ⇒ XSLT-transformations

Chapter 20

Bridge Java – SML

20.1 Design der Klassenhierarchie

Diese Klassen MathEngine, bilden das Interface zum Frontend und zum Dialog.

20.1.1 MathEngine

Diese Klasse wird zu vom Dialog gestartet und stellt die Verbindung zur eigentlichen Bridge her. Alle Objekte am Frontend kommunizieren nur ber diese Komponente mit der Bridge. MathEngine ist ein Singleton (siehe **??**).

20.1.2 CalcHead

Diese Klasse reprsentiert den Kopf einer Berechnung in SML. Der CalcHead enthlt das Model und die Spezifikation einer Berechnung (siehe Glossar). Er muss ausgefllt und korrekt sein, bevor die eigentliche Berechnung gestartet werden.

20.1.3 Formula

Diese Klasse reprsentiert eine Formel in einem CalcTree.

20.1.4 Tactic

Diese Klasse reprsentiert eine Tactic in einem CalcTree.

20.1.5 CalcElement

bergeordnete Klasse von CalcHead, Formula, und Tactic.

20.1.6 CalcTree

Diese Klasse reprsentiert eine Berechnung im SML-Kernel. Wenn der Benuzter am Frontend eine neue Berechnung startet, wird intern ein solches Objekt erzeugt.

ICalcIterator iterator() liefert einen neuen Iterator zu diesem CalcTree.

public ICalcIterator getActiveFormula() liefert einen neuen Iterator, der die aktive Formel[1] markiert.

public void moveActiveFormula(ICalcIterator newActiveFormula) bewegt die aktive Formel auf eine neue Position, die von dem mitgegebenen Iterator markiert wird.

public int replaceFormula(Formula newFormula) ersetzt die aktive Formel durch eine neue, mitgegebene Formel.

public int appendFormula(Formula newFormula) fgt diese neue Formel unter der aktiven Formel ein.

public int setNextTactic(Tactic tactic) teilt dem Kernel mit, welche Taktik im nchsten Schritt angewandt werden soll.

public Tactic fetchProposedTactic() liefert die vom Kernel vorgeschlagene Taktik fr den nchsten Schritt zurck.

public Tactic[] fetchApplicableTactics(int scope) liefert ein Feld mit allen Taktiken in einem bestimmten Bereich *scope* zurck, die an der aktiven Formel als nchster Schritt angewandt werden knnen.

public int autoCalculate(int scope, int nSteps) fordert den Kernel auf, die Berechnung bis zu einem bestimmten Punkt selbststndig weiterzurechnen. Der Parameter *scope* gibt den Bereich an, der beim weiterrechnen nicht verlassen werden soll, und kann folgende Ausprgungen besitzen:

- Aktuelles Subproblem der Berechnung
- Aktuelle Subberechnung
- Die ganze Rechnung

Der Parameter *nSteps* gibt an, wieviele Schritte maximal weitergerechnet werden soll. Wenn *nSteps* den Wert 0 besitzt, wird bis zum Ende fertiggerechnet.

---

[1] Damit ist diejenige Stelle im CalcTree gemeint, an der die Berechnung fortgesetzt wird.

20.1.7 CalcIterator

Der CalcIterator dient dazu, sich im CalcTree zu bewegen und eine bestimmte Formel oder Taktik zurckzuliefern, oder um eine Stelle zu markieren, ab der weitergerechnet werden soll. Diese Klasse verwendet das Design Pattern Iterator (siehe **??**).

Methoden

boolean moveRoot() setzt den Iterator auf die erste Position des CalcTree

boolean moveUp() bewegt den Iterator zur vorhergehenden Formel

boolean moveDown() bewegt den Iterator zur nchsten Formel

boolean moveLevelDown() bewegt den Iterator eine Verschachtelungsebene tiefer. Nur relevant bei Berechnungen mit Subrechnungen.

boolean moveLevelUp() bewegt den Iterator eine Verschachtelungsebene nach auen.

boolean moveTactic() bewegt den Iterator zur Taktik, die zu der aktuellen Formel fhrte.

boolean moveFormula() bewegt den Iterator nach Aufruf von *moveTactic* wieder zur Formel zurck.

int getLevel() liefert den Level (=Anzahl der Subrechnungsebenen) der aktuellen Formel.

ICalcElement getElement() liefert das aktuell markierte Element der Berechnung (Formel, Taktik, CalcHead) zurck.

Object clone() klont den Iterator und liefert ein zweites Iterator-Objekt zurck, das auf die selbe Position im CalcTree zeigt.

int compareTo(Object o) vergleicht diesen Iterator mit einem zweiten. Die Iteratoren werden als gleichwertig betrachtet, wenn beide auf die selbe Position im CalcTree

20.1.8 BridgeMain

Die zentrale Komponente des Bridge-Systems. Sie startet die anderen Prozesse, die zur Kommunikation mit SML dienen, leitet die Anfragen an den Kernel weiter.

20.1.9 SMLThread

Hier wird der SML-Prozess gestartet und die Ein- und Ausgabe der Daten gemanagt.

20.1.10 XMLParser

Hier wird der XML-Output des Kernels geparst und in Java-Objekte umgewandelt, die dann wieder an das Frontend weitergereicht werden.

Part IV

Software Design Document

Chapter 21

Session Management

.[1]

21.1 Logging into the System and Bootstrapping

As a distributed system, *ISAC* is started in several steps. For the following discussion, we will differentiate *ISAC*'s components into centralised static components and components created dynamically on a per-session or per-user basis.

Static, centralised components:

- Math Engine
- Knowledge Base
- Example Collection
- Object Manager

Components created per session or per user:

- Session Dialog
- Presentation Layers
- WorkSheet Dialog
- Browser Dialog
- User Model

---

[1] Begin of copy from [Kre05] p.63-65.

As for the centralised components running on dedicated servers, we will assume they have been started by their respective administrators and their services are available at login time of an individual user.

For initializing the components of $\mathcal{ISAC}$ running on the user's machine, several actions have to be performed:

- Identify the user
- Start components individual to a user or session
- Retrieve the user's User Model from centralised storage
- Connect the components - centralised and individual - to each other

One of the main problems in designing this part of $\mathcal{ISAC}$ is finding a bootstrapping procedure capable of connecting all the relevant components, which essentially means making sure that every component can address any component it needs to cooperate with.

Whereas the always-present, centralised components can easily be adressed by entering the network address of their respective servers or storing the addresses in configuration files, the dynamically created per-user and per-session components are harder to localise. In principle, only the instance which created a component can know how to address it in the first place. As a solution, a Session Dialog component was introduced to initialise and keep track of all dynamically created components.

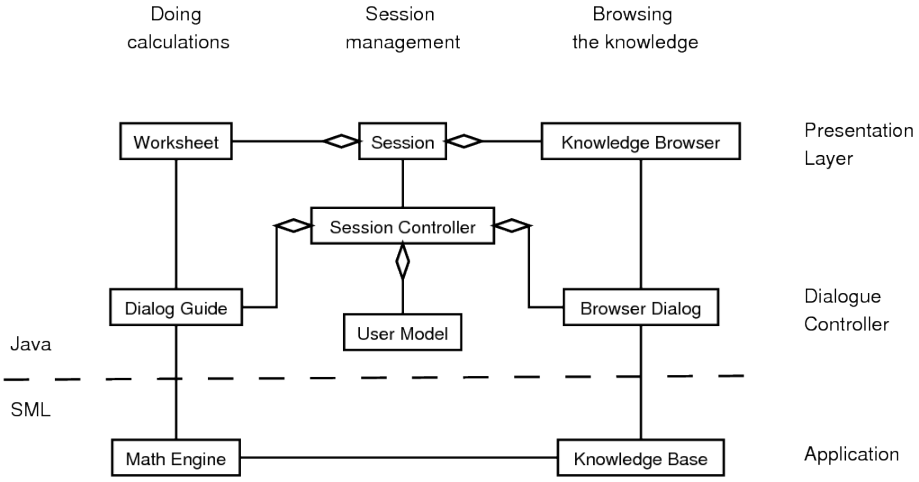


Figure 21.1: Adding session management and a shared user model

Another problem is communicating with web browsers running on the user's machine unless new content has been requested by the browser. This makes it impossible to notify the user of a change in the state of one of the centralised components until the browser polls for new content. While this does not hinder browsing the Knowledge Base which presents data of rather static nature, watching a calculation being modified by the Math Engine requires a minimal software component capable of handling requests from the network running on the user's machine.

Once it was clear that some sort of active software on the user's machine would be required, parts of the Session Dialog functionality were moved onto the user's machine along with the login procedures and a basic GUI coordinating the display requirements of the various components. The component running on the user's machine is simply referred to as GUI.

It is hoped that features and stability of the startup process will greatly improve once the Dinopolis middleware system becomes available.

For details on the design of the bootstrap process and session management see [Gri03, Hoc04].

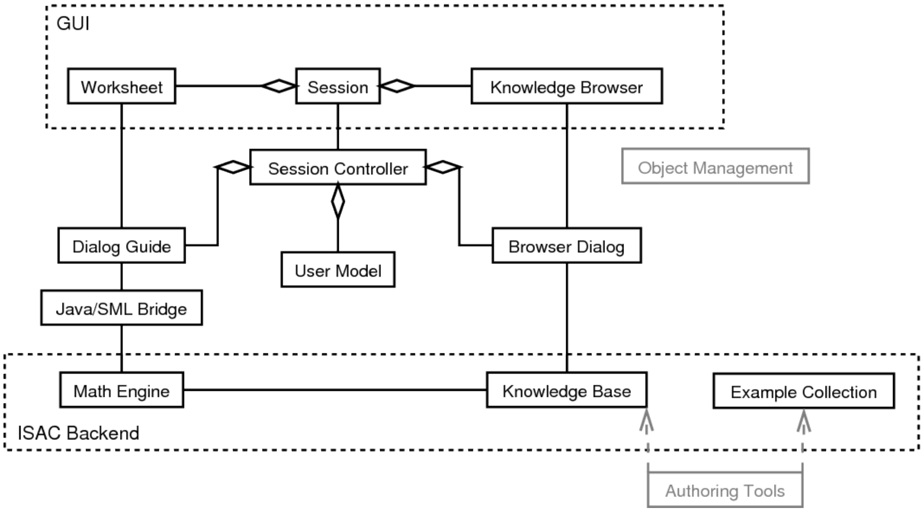


Figure 21.2: The overall design of the $\mathcal{ISAC}$ system

[2]

Attention, please: the following is copied from [Hoc04] and not up to date since May 2005.

21.2 SessionDialog, BrowserDialog and WorksheetDialog

The architectural design as described in sect.16.1 and in chap.WN-add-BrowserDialog led to the following software design

The figure Fig.21.3 p.134 shows the dialoges within the session manegement.

---

[2] End of copy from [Kre05] p.63-65.

21.3 Starting a session

As already mentioned several times, one task of the graphical user interface is to establish a connection to *ISAC*. This process is shown in figure 21.4.

The different steps in the initialization process are described in detail as follows.

21.3.1 Communication with the InformationProcessor

1. **Authorize the user**
   Authorization within *ISAC* is not only necessary for rights management; additionally, the WorksheetDialog must identify the user to record the user history details.

   The method *login* from the InformationProcessor takes two parameters, namely the username and password, both are of type *string*. If the user has authorized correctly, the method returns a *session id*. This session id is a unique identifier for the user. This means the user can log in with different applications and he will always get the same session id.

2. **Establish connection to the SessionDialog**
   The method *getSDialog* from the InformationProcessor is parameterless and returns an object that implements the *SDialog* interface. The way the InformationProcessor establishes the connection to the SessionDialog is transparent to the client. More details about the inner workings of the InformationProcessor can be found in [Gri03].

3. **Load the knowledge base entries**
   The entries of the knowledge base are stored as XML documents. These XML files are structured hierarchically in accordance with the file system structure. The InformationProcessor provides a method called *loadHierarchy*, which takes as parameters the session identifier and the hierarchy type. The type describes which part of the knowledge base (theories, methods, problems see section **??**) should be loaded. The method also makes it possible to load the example hierarchy which strictly speaking is not part of the knowledge base.

   The method returns the XML file hierarchy of the selected type in *string* representation, which needs to be converted into a hierarchical representation (see section 21.3.3).

21.3.2 Communication with the SessionDialog

4. **Create WorksheetDialog and connect it with the current session**

   The WorksheetDialog (in the *ISAC* system architecture also called *DialogGuide*) must know the current user; therefore, the WorksheetDialog needs to be connected to the SessionDialog, which makes it possible to identify the current user.

The SessionDialog provides a method called *openDGuide* that takes as a parameter the session id and returns a dialog id. The dialog id identifies the DialogGuide which the SessionDialog created transparently for the client.

5. **Establish connection to the DialogGuide**
   To get the DialogGuide object created in step 21.3.2, the SessionDialog provides the method *getDGuide*. This method takes as parameter the dialog id that the method *openDGuide* returned. The return value of *getDGuide* is an object that implements the *DGuide* interface. This object needs to be passed to the *Worksheet* and is responsible for handling the communication between the *Worksheet* and the *backend* (the calculation part of *ISAC*).

21.3.3 XMLHierarchyParser

The XMLHierarchyParser class generalizes the functionality that each knowledge base parser has in common (see figure 21.5). The task of each concrete value of the XMLHierarchyBrowser is to convert the hierarchy of a knowledge base part (loaded in step 3 of the initialization process) into a representation that can be properly displayed by the GUI, especially by the HierarchyBrowser. Note that for the theories part of the knowledge browser there is currently no XMLTheoryBrowser designed, as the data format of theories is still original Isabelle, and not XML.

Nevertheless, the XMLTheoryBrowser can always be included as a subclass of XMLHierarchyParser.

The last two steps to complete the initialization process are described in the following:

6. **Get the root element of the hierarchy**
   A concrete value (e.g. XMLMethodParser) of the XMLHierarchyParser parses the string representation of the hierarchy that was returned by the method *loadHierarchy* in step 3 of the initialization process. The method *getRoot()* returns the root node of the hierarchy.

7. **Create hierarchy that can be displayed in the HierarchyBrowser**
   The root node created in the step above needs to be given as parameter for the method *createXXXNodesHierarchy*. XXX is a placeholder for one of following concrete values: Method, Problem, Example. This method returns the hierarchy in a representation that can be displayed by the HierarchyBrowser.

21.4 User Data

The present prototype, as of Feb.2012, holds all data in text files. Apparently, all these data should go into a database.

21.4.1 The hierarchy of data

```
*  isac-host1 (e.g. IICM)
   * domain11 (e.g. KFU Mathematik)
      *  group111 (e.g. Analysis 3.Semester)
         * student 1111
         * student 1112 (Max mustermann)
         * student 1113
         :
         * student 111i
      *  group112 (e.g. Algebra 5.Semester)
         * student 1121
         * student 1122 (Max mustermann)
         * student 1123
         :
         * student 111j
      :
   * domain12 (e.g. Informatik TUG)
      *  group121 (e.g. Signal Processing 5.Semester)
         * student 1211
         * student 1212
         * student 1213 (Max mustermann)
         :
         * student 121k
      *  group122 (e.g. Computational Geometry 3.Semester)
      :
:
*  isac-host2 (e.g. RISC Linz)
   * domain21 (e.g. WIFI OOe)
     :
```

For instance, for an exam Signal Processing, Max is uniquely identified by 'domain11/group121/Max'. The file 'accounts.txt' determines the access rights at login:

```
domain11/group111/Max=password
domain11/group112/Max=password
domain12/group121/Max=password
domain11/group111/moritz=pwd2
```

21.4.2 Data of a single user

All data of a single user are in four files, where the username is a prefix of the filenames. Below we assume user 'x':

1. x_admin.txt will be deleted: *ISAC* provides math learning services depending on access rights as described above; so any other personal data like student number, date of birth, etc does *not* belong to *ISAC*.

2. **x_settings.txt** concerns technical aspects like language of the learner, background color on the GUI, etc.

   ```
   KEY4_LANGUAGE=VAL4_ENGLISH
   TODO
   ```

3. **x_profile.txt** concerns learning aspects; initializes the user-model first time and in case of being 'exclusive'. presently there is one profile for all exams.

   ```
   exclusivse=true

   visibile_examples = None
   # only whole branches shall be handled?...
   # visibile_examples = [["Examples","IsacCore","Simplification"],
   # ["Examples","Biology"]]

   KEY1_START_SPECIFY=VAL1_SKIP_SPECIFY_TO_START_SOLVE
   KEY2_NEXT_BUTTON=VAL2_FORMULAE_ONLY
   KEY3_STATUS=VAL3_LEARNER
   TODO
   ```

   Presently it is not clear, how to relate the information in 'accounts.txt' to multiple profiles (e.g. different rules in the exam for Analysis and in Signal Processing) and multiple UserModels ('*_model.txt: different learning strategies in Analysis and in Signal Processing). So the present code allows only one profile and one model per user in different groups in different domains at a time.

4. **x_model.txt** concerns learning aspects over time and represents the UserModel. The design will be in principle: the UserGuide records all user interactions (history) and upon ending a session, the history will be abstracted and stored (not yet decided what, how, where). At start of a new session, the UserModel might be reconstructed from the previous session — in certain cases. One case is clear: 'exclusive' profiles inhibit such reconstruction.

   ```
   #dynamic dialog data of user 'x'
   #WN070510
   TODO-key=TODO-value
   ```

21.5 WindowApplication

The WindowApplication is, besides the initialization of *ISAC*, responsible for the properly responding to user actions. The WindowApplication is split into two areas, one in which the different hierarchies are displayed (`BrowserPanel`) and one in which different Worksheets can be displayed as *InternalFrames* (`WorksheetDesktopPane`).

The WindowApplication class provides, among other things, the following methods:

- `openNewWorksheet` opens a new Worksheet as an `JInternalFrame`; this allows the GUI to display a window within another window.
- `getActiveWorksheet` returns the currently active Worksheet. This is the Worksheet with the *cursor* in it.
- `actionPerformed` is called if the user has generated a special event (e.g. clicking a button). This method then makes the proper actions for that event.

A complete list of the methods with short description can be found as JavaDoc on the CD accompanying this thesis.

21.5.1 Internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes.

An internationalized program has the following characteristics[3]:

> - With the addition of localization data, the same executable can run worldwide.
> - Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead, they are stored outside the source code and retrieved dynamically.
> - Support for new languages does not require recompilation.
> - Culturally dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
> - It can be localized quickly.
>
> Locale-specific data must be tailored according to the conventions of the end user's language and region. The text displayed by a user interface is the most obvious example of locale-specific data.

---

[3] http://java.sun.com/j2se/1.4.2/docs/guide/intl/index.html

The GUI of *ISAC* uses Java's locale support for internationalization purposes. Therefore, all locale-specific data is stored in *Properties* files, for each supported locale a separate Properties file must be created (e.g. `MenuBar_de_DE.properties` for German locale support and `MenuBar_en_US.properties` for English locale support). Properties files are text files with key value pairs, in which the value is stored in the corresponding locale. The `ResourceBundle` class of the Java programming platform loads the Properties file, which matches the locale settings of the current user.

21.5.2 Window Management

The GUI uses the multi-document interface (MDI)[4] technology for the management of the Worksheet windows. The MDI technique uses a primary window, called parent window, to visually contain a set of related document or child windows.

The `JDesktopPane` from the Java programming platform provides the functionality that is necessary to use the MDI technology. Each window that should appear in the Worksheet area is added to the `JDesktopPane` as an instance of the `JInternalFrame` class.

21.5.3 XMLHierarchyParser

The XMLHierarchyParser uses the *Xerces2*[5] *DOMParser* to parse the different hierarchy files like the problem hierarchy shown in listing 21.1.

```
< NODE >
  < ID > Problemhierarchy < /ID >
< NODE >
  < ID > e_pblID < /ID >
  < CONTENTREF > pbl_1.xml < /CONTENTREF >
< /NODE >
< NODE >
  < ID > equation < /ID >
  < CONTENTREF > pbl_2.xml < /CONTENTREF >
  < NODE >
    < ID >univariate< /ID >
    < CONTENTREF > pbl_2_1.xml < /CONTENTREF >
    < NODE >
      < ID > linear < /ID >
      < CONTENTREF > pbl_2_1_1.xml < /CONTENTREF >
    < /NODE >
    < NODE >
      < ID > root < /ID >
      < CONTENTREF > pbl_2_1_2.xml < /CONTENTREF >
      < NODE >
        < ID > sq < /ID >
        < CONTENTREF > pbl_2_1_2_1.xml < /CONTENTREF >
```

---

[4] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/ch10g.asp and also available on the CD accompanying this thesis (file Microsoft_MDI.pdf).

[5] http://xml.apache.org/xerces2-j/index.html

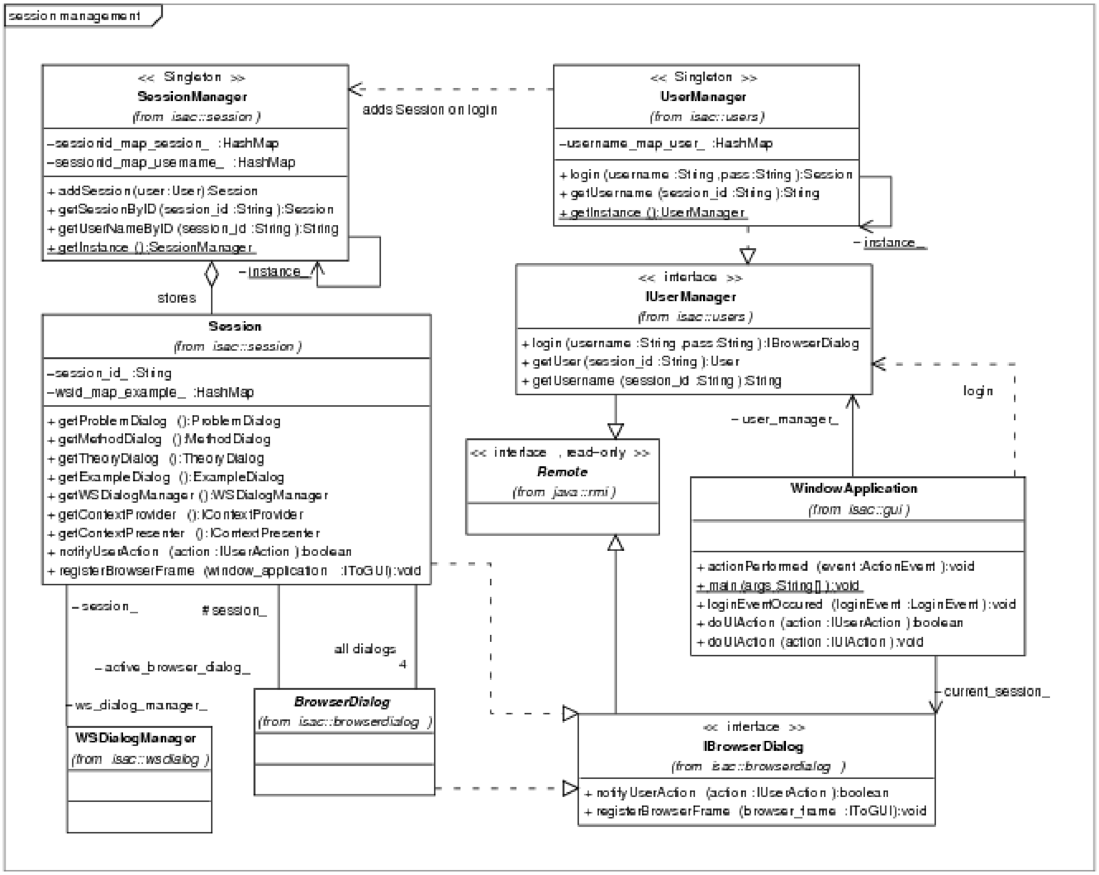
```
      < NODE >
        < ID > rat < /ID >
        < CONTENTREF > pbl_2_1_2_1_1.xml < /CONTENTREF >
      < /NODE >
    < /NODE >
    < NODE >
     ...
```

Listing 21.1: Problem hierarchy in XML representation.

[6]

---

[6] Begin of Georg Kompachers contribution in [Kom07].


session management
<< Singleton >>
SessionManager
(from isac::session)
–sessionid_map_session_ :HashMap
–sessionid_map_username_ :HashMap
+ addSession (user :User):Session
+ getSessionByID (session_id :String):Session
+ getUserNameByID (session_id :String):String
+ getInstance ():SessionManager
– instance_
adds Session on login
<< Singleton >>
UserManager
(from isac::users)
–username_map_user_ :HashMap
+ login (username :String ,pass :String):Session
+ getUsername (session_id :String):String
+ getInstance ():UserManager
– instance_
stores
Session
(from isac::session)
–session_id_ :String
–wsid_map_example_ :HashMap
+ getProblemDialog ():ProblemDialog
+ getMethodDialog ():MethodDialog
+ getTheoryDialog ():TheoryDialog
+ getExampleDialog ():ExampleDialog
+ getWSDialogManager ():WSDialogManager
+ getContextProvider ():IContextProvider
+ getContextPresenter ():IContextPresenter
+ notifyUserAction (action :IUserAction):boolean
+ registerBrowserFrame (window_application :IToGUI):void
<< interface >>
IUserManager
(from isac::users)
+ login (username :String ,pass :String):IBrowserDialog
+ getUser (session_id :String):User
+ getUsername (session_id :String):String
login
– user_manager_
<< interface , read-only >>
Remote
(from java::rmi)
WindowApplication
(from isac::gui)
+ actionPerformed (event :ActionEvent):void
+ main (args :String[]):void
+ loginEventOccured (loginEvent :LoginEvent):void
+ doUIAction (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
– session_
# session_
all dialogs
4
– active_browser_dialog_
– ws_dialog_manager_
– current_session_
WSDialogManager
(from isac::wsdialog)
BrowserDialog
(from isac::browserdialog)
<< interface >>
IBrowserDialog
(from isac::browserdialog)
+ notifyUserAction (action :IUserAction):boolean
+ registerBrowserFrame (browser_frame :IToGUI):void

Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 21.3: The dialoges in a session

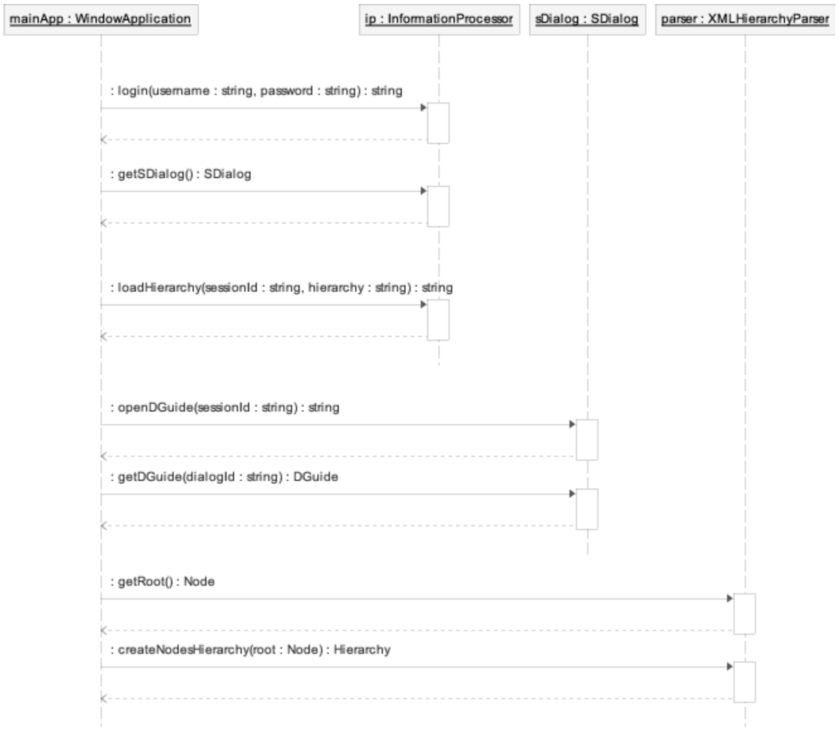

mainApp : WindowApplication
ip : InformationProcessor
sDialog : SDialog
parser : XMLHierarchyParser
: login(username : string, password : string) : string
: getSDialog() : SDialog
: loadHierarchy(sessionId : string, hierarchy : string) : string
: openDGuide(sessionId : string) : string
: getDGuide(dialogId : string) : DGuide
: getRoot() : Node
: createNodesHierarchy(root : Node) : Hierarchy

Figure 21.4: Sequence diagram about the required initialization steps to build up the communication with $\mathcal{ISAC}$.

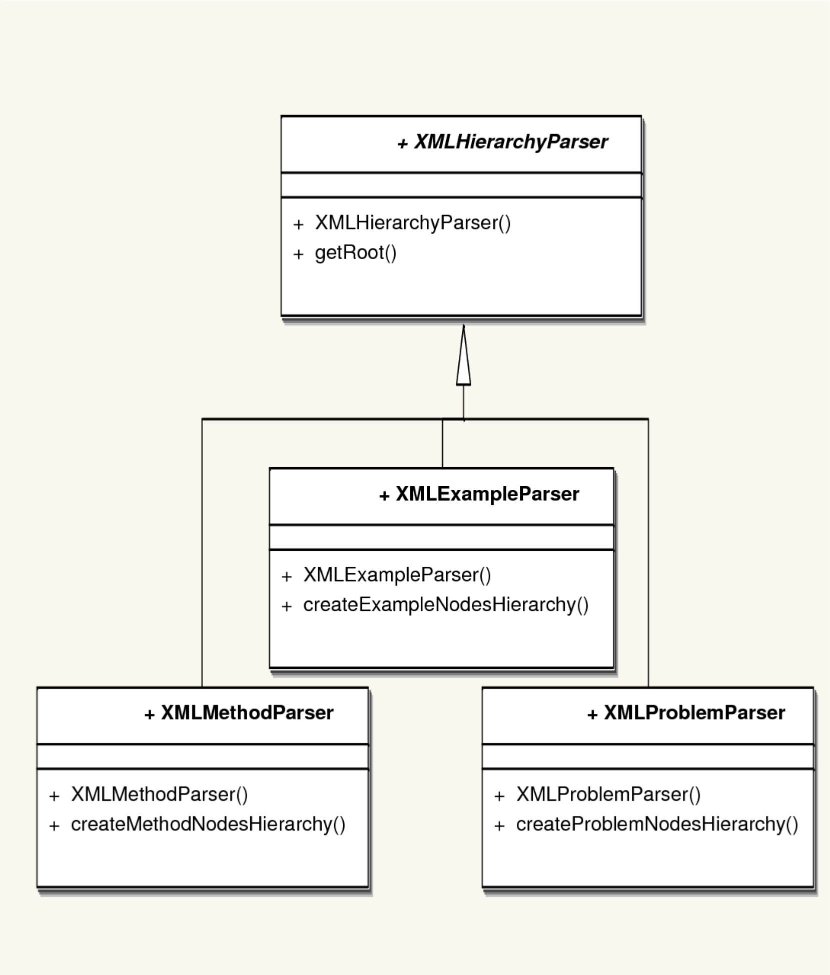
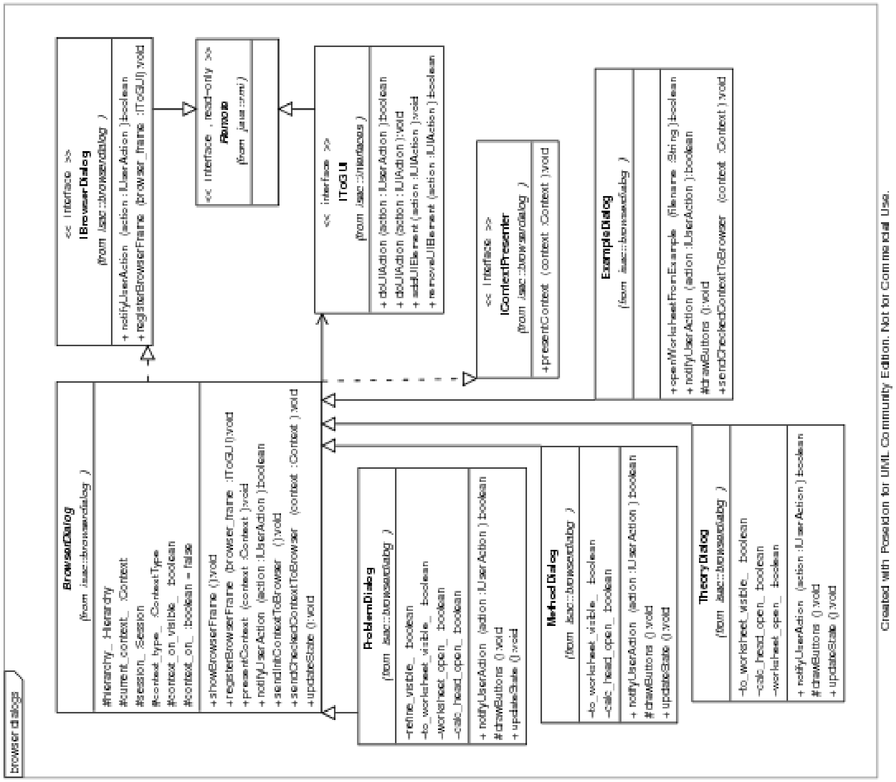


Figure 21.5: Abstract class XMLHierarchyParser acts as superclass for the XMLParser that parses the different knowledge base hierarchies.

Chapter 22

Browser Dialogs

The figure fig.22.1 p.137 shows the class diagram for the browser dialoges.



browser dialogs
<< interface >>
IBrowserDialog
(from isac::browserdialog)
+notifyUserAction (action :IUserAction):boolean
+registerBrowserFrame (browser_frame :IToGUI):void
<< interface , read-only >>
Remote
(from java::rmi)
<< interface >>
IToGUI
(from isac::interfaces)
+doUIAction (action :IUserAction):boolean
+doUIAction (action :IUIAction):void
+addUIElement (action :IUIAction):void
+removeUIElement (action :IUIAction):boolean
<< interface >>
IContextPresenter
(from isac::browserdialog)
+presentContext (context :IContext):void
ExampleDialog
(from isac::browserdialog)
+openWorksheetFromExample (filename :String):boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+sendCheckedContextToBrowser (context :IContext):void
BrowserDialog
(from isac::browserdialog)
#hierarchy_ :Hierarchy
#current_context_ :IContext
#session_ :ISession
#context_type_ :ContextType
#context_on_visible_ :boolean
#context_on_ :boolean = false
+showBrowserFrame ():void
+registerBrowserFrame (browser_frame :IToGUI):void
+presentContext (context :IContext):void
+notifyUserAction (action :IUserAction):boolean
+sendInitContextToBrowser ():void
+sendCheckedContextToBrowser (context :IContext):void
+updateState ():void
ProblemDialog
(from isac::browserdialog)
-refine_visible_ :boolean
-to_worksheet_visible_ :boolean
-worksheet_open_ :boolean
-calc_head_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void
MethodDialog
(from isac::browserdialog)
-to_worksheet_visible_ :boolean
-calc_head_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void
TheoryDialog
(from isac::browserdialog)
-to_worksheet_visible_ :boolean
-calc_head_open_ :boolean
-worksheet_open_ :boolean
+notifyUserAction (action :IUserAction):boolean
#drawButtons ():void
+updateState ():void
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 22.1: The class diagram for the browser dialoges

BrowserDialog implements IBrowserDialog, IContextPresenter

Each `BrowserDialog` is responsible for the communication to the `WSDialog`s via the `ContextProvider`-interface and also for the communication to the Browsers over Java RMI (Remote Method Invocation). Each `BrowserDialog` gets information from the Browser by triggered `UserActions`. In the other direction `UIAction`s are sent. Thes class has the following constructor:

`BrowserDialog(Session session)`: This class must have the `session_` locally stored to provide access to the active `WorksheetDialog` for this `BrowserDialog`. The `Session` stores the `WSDialogManager` and the `WSDialogManager` itself stores the active `WorksheetDialog`. The `UserLanguage` is also read out from the `session_` and stored locally.

The `BrowserDialog` offers the following public methods:

- `boolean notifyUserAction(IUserAction action)`: This method is responsible for handling all actions coming from the browser stored in `browser_frame_rmi_`. If the request has been accepted the return value is true otherwise it is set to false. The request can be denied if the user has no priviledges to do such a UserAction. To make this decision the BrowserDiaolg has to check up the `UserSettings` and the `UserModel`. But this would not lead to an exception. An exception is thrown if the current state of the `BrowserDialog` cannot handle the `UserAction` (`DialogProtocolException`).

- `void presentContext(Context context)`: This method belongs to the `IContextPresenter` interface. It is called by the `WorksheetDialog`, if context changes have to be shown in the browser. This method checks the status of the member variable `context_on_`. If `context_on_` is set to false, the `current_context_` won't change.

- `void registerBrowserFrame(IToGUI browser_frame)`: This method is called from the `WindowApplication` to register the `BrowserFrame` at the `BrowserDialog`. After registering the `BrowserFrame` the `hierachy_` is sent as an `UIActionOnHierarchy`.

- `void sendCheckedContextToBrowser(Context context)`: The Method is used when `context_on_` is set to `true`. Then the method checks the received context by use of the `checkContext` method of the `IContextProvider` interface. This interface returns again a context, which is sent to the `BrowserFrame`.

- `void sendInitContextToBrowser()`: This method is used when the varable `context_on_` is set from `false` to `true` or when `showBrowserFrame()` is called. Then the context of the `WorksheetDialog` of the active worksheet which has implemented the `IContextProvider` interface is the initial context and it is sent to the `BrowserFrame`.

- `void showBrowserFrame()`: If an `UserAction` with the `ActionID` is `UI_BROWSER_GET_FOCUS` received this method is called and the the `BrowserFrame` will get the focus.

The `BrowserDialog` offers the following protected methods:

- `void drawButtonContextOff()`: This method causes the Context Off button to be shown. If the button is already shown, nothing happens. Otherwise the Button for Context On is removed and the Button for Context Off is shown.

- `void drawButtonContextOn()`: This method causes the Context On button to be shown. If the button is already shown, nothing happens. Otherwise the Button for Context Off is removed and the Button for Context On is shown.

- `abstract void drawButtons()`: This method has to be implemented by the drived classes because every browser has a different set of buttons.

- `HierarchyKey getHierarchyKey(String xml_content)`: To get the correct `HierarchyKey` from the `xml_content` string the '`HierarchyKeyParser` will be used and the `HierarchyKey` will be returned.

- `void interpretLink(URL url)`: This method responsible to interpret a link coming from the `BrowserFrame`. Normal links fetch html-files which are shown in the correspondig browser (The first three letters in the filename show which browser is responsible for the link). For specific links coming from the ExampleBrowser which include `#COMMAND_EXEC_EXAMPLE#` a the method `openWorksheetFromExample(String kestore_key)` is called. The `kestore_key` is parsed from the `URL`.

- `Vector<String> loadContent(String session, String type, String filename)`: The content is loaded from the `KEStore` by using the `callKEStore` method whith encapsulates the communication via `XML-RPC`. The content is stored as the first element. The `Vector<String>` is used to get better structured information if an erros occurs.

- `Hierarchy loadHierarchy(ContextType type)`: The `Hierarchy` is loaded from the `KEStore` by using the `callKEStore`. If the type is unknown, `null` will be returned.

- `void sendContextToBrowser(Context context)`: This method sends a context to the browser but first it has do get the `HierarchyKey` and store it into the `context` object. The sent context is the new `current_context_`.

- `void sendGetFocusToBrowser()`: This method is responsible for making a `BrowserFrame` visible to the user, i.e. when `interpretLink` is called to open a html-file in a different browser this method will cause the `BrowserFrame` to come in front of all frames.

- `void sendLinkToMiniBrowser(Context context, ContextType type)`: Sends the browser the information to display the new selected problem. This method is called if a new context has to be shown in the browser. A link is created out of the `KEStoreKey` which is stored in the `Context` object. This created link is then sent to the minibrowser via `void sendLinkToMiniBrowser(String link)`.

- `void sendLinkToMiniBrowser(String link)`: The `user_status` is read out of the `Usermanager`, which is implemented as a singleton and then `void sendLinkToMiniBrowser(String link, int user_status )` is called.

- `void sendLinkToMiniBrowser(String link, int user_status )`: Calls the `doUIAction` method of the `BrowserFrame` with an `UIActionOnLink` but first the `user_status` has to be checked for the privileges of the user to decide how much information will be provided.

- `void setCurrentContextToWorksheet()`: This method takes the `current_context_` and calls the active Worksheet to set the `current_context_` as the new context for the worksheet. The ¡ToWorksheet¿ button is only visible if there is an active Worksheet and if the calcHead is opened.

- `void switchToMatchOff()`: This Method sends an `UIAction` to instruct the `BrowserDialog` to generate the Context off button (and also remove the Context on button first).

- `void switchToMatchOn()`: This Method sends an `UIAction` to instruct the `BrowserDialog` to generate the Context on button (and also remove the Context off button first).

The `BrowserDialog` offers the following private methods:

- `private Vector<String> callKEStore(String function, Vector<String> parameters)`: This method encapsulates the communication with `KEStore`. This communictaion is done via `XML-RPC`.

- `private Vector<String> filterContent(String session, Vector<String> result)`: Inside this method a `DOM` is build from the `String` stored in the result `Vector`. Then some Node of the `DOM` can be filtered depending on the privileges of the user.

Here are the member variables of the `BrowserDialog`. They are all set protected to be accessible from the derivated classes:

- `IToGUI browser_frame_rmi_`
- `Hierarchy hierarchy_`
- `Context current_context_`
- `Session session_`
- `ContextType context_type_`
- `boolean context_on_visible_`
- `boolean context_on_`
- `UserLanguage user_language_`

22.1 Communication with the Browsers

22.2 Communication with the Worksheet Dialogs

22.3 Communication with the KEStore

[1]

---

[1] End of Georg Kompachers contribution in [Kom07].

Chapter 23

Worksheet Dialog

The subsequent text follows [Kre05], the initial design considerations. The sections 23.1 downto 23.3 deal with representing calculation data for the WorkSheetDialog (and for the WorkSheet), and the sections 23.4 downto 23.6 deal with communicating with other components.

For the second part od the Dialog Guide, the BrowserDialog, see (see **??** on p.**??**). The details of interaction between BrowserDialog and Knowledge Browsers are analoguous to the interaction between WorkSheetDialog and WorkSheet. Thus in the sequel 'Dialog Guide' is used for both dialogs.

[1]

23.1 Storing Enumeration Types

In many contexts in the implementation, variables or parameters hold information about one out of a finite set of alternatives.

Such values are often stored as integer numbers with every alternative being represented by a number. Throughout the $\mathcal{ISAC}$ system, named constants (`public static final int`) are used to represent alternatives.

While this approach improves readability and the names of the constants are scoped, the values themselves are still indistinguishable from other integers and thus cannot be type-checked or range-checked for proper use at compile time.

In the future, these integers will be replaced by the Typesafe Enum pattern [Blo01] to prevent unintentionally wrong use of enumerated values. In this pattern, every enumeration has a class of its own with a private constructor and all possible values are public constants initialised internally using the private constructor.

Java version 1.5 already implements exactly this pattern; the development environment during writing this part of the documentation was still fixed to Java 1.4.

---

[1] Begin of copy from [Kre05] p.78

23.2 The Hierarchy of Mathematical Objects

23.2.1 `CalcElement`

`CalcElement`, accessible through the interface `ICalcElement`, is the common base class for all objects stored in a calculation. Every node in a `CalcTree` is a `CalcElement` or derived from it. `CalcIterator`s (see 23.3 below) reference `ICalcElement`s.

Data Stored

Apart from serving as a common reference, `ICalcElement` defines the attributes:

`Type` is the concrete type of the element, one of `CALCEL_FORMULA`, `CALCEL_TACTIC`, `CALCEL_CALCHEAD` or `CALCEL_ASSUMPTION`.

`Visibility` hints at whether to display the `CalcElement` to the user: `DISPLAY_VISIBLE_DEFAULT`, `DISPLAY_VISIBLE_HIGHLIT`, `DISPLAY_HIDDEN` (invisible but can be displayed on request), `DISPLAY_INVISIBLE` (remaining invisible).

`ViewStyle` hints at how detailed the element will be displayed to the user. `ViewStyle`s have been defined for `CalcHead` and will be added as alternative levels of detail for the other derived types are developed. The `ViewStyle` is chosen by the Dialog Guide to match the user's level of expertise.

`Rating` stores the result of the Dialog Guide's estimation of the user's familiarity with the `CalcElement` in question, obtained from evaluating a query to the User Model. As estimating the user's familiarity with the `CalcElement` is expected to be a complicated process, the results, once computed, are stored for later reference during the session.

It is debated whether the attributes `Visibility`, `ViewStyle` and `Rating` will stay with a data type used throughout the system or they will be moved to a derived type visible only to the Dialog Guide and the Presentation Layer.

Conversions

Regardless of the internal representation, a `CalcElement` provides the methods `toSMLString` and `toMathML` for delivering external representations of the data stored for the Math Engine and the Presentation Layer, respectively. As soon as appropriate parsers are implemented, the methods `fromSMLString` and `fromMathML` will be added. For the time being, the SML representation of the Math Engine is used internally for prototyping purposes and is likely to be replaced by a more efficient structured representation in the future.

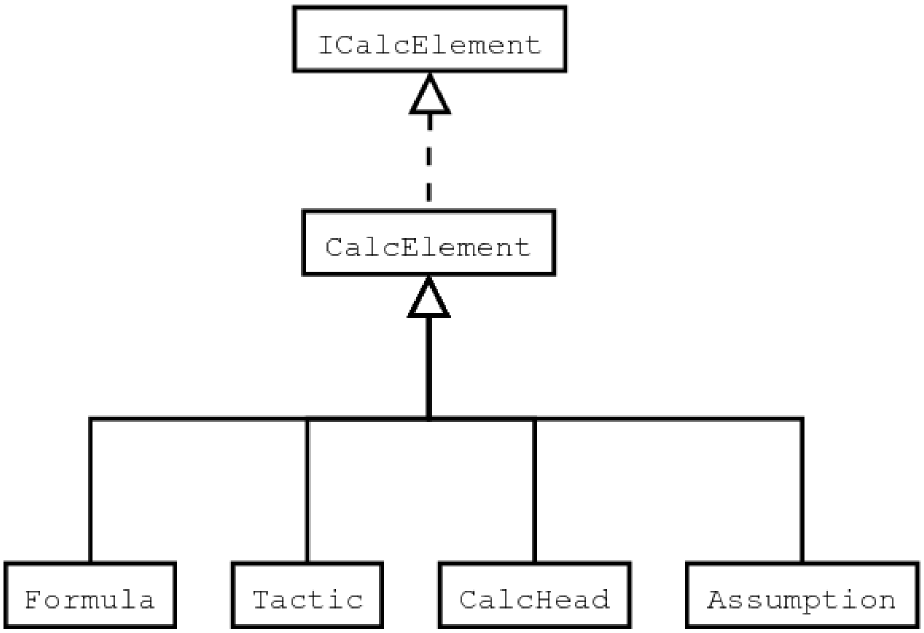


Figure 23.1: The hierarchy of mathematical objects

23.2.2 Classes Derived from `CalcElement`

Apart from `CalcHead`, the classes derived from `CalcElement` are implemented basically as mere stubs, as the Dialog Guide does not know the mathematical meaning of the data intended to be contained, hence does not use or manipulate them.

`Formula`

`Formula` is presently implemented storing its internal data as a `String` holding exactly the string representation of terms as defined by Isabelle.

At a later stage in development, the internal structure of a `Formula` will be made accessible to the user. To provide for this, a method `copyTerm` is added to the class to extract subterms (see section 23.2.3) for substitutions or similar purposes.

`CalcHead`

As a consequence of the decision to integrate subproblems into the main `CalcTree` (see section 16.5.3), `CalcHead` has to be derived from `CalcElement`.

According to 8.4.2 and 8.4.3, the `CalcHead` has the following additional attributes used during the Specifying Phase (see section 16.4.1):

`CalcHeadStatus` summarizing all the states of the `ModelItem`s: the state is "`correct`" or "`incorrect`".

`BelongsTo` tells whether the items belong to a `Model` of a problem(!) or a `Guard` of a method(!)

`Model` **or** `Guard` , consisting of `Given`, `Relate`, `Where`, `Find`: `ModelItemList`s, which are lists of `ModelItem`s.

`Specification` , consisting of `Method`, `Problem` and `Theory`, each being an identification referencing the Knowledge Base.

Every `ModelItem` is a formula associated with a `Status` for feedback from the Math Engine. During the Specifying Phase, the Math Engine checks the `CalcHead` and provides specific feedback on every `ModelItem` labelling it as one of `STATUS_CORRECT`, `STATUS_INCORRECT`, `STATUS_SYNTAXERROR`, `STATUS_TYPEERROR`, `STATUS_INCOMPLETE`, `STATUS_MISSING`, `STATUS_HELP_ME` or `STATUS_SUPERFLUOUS`.

The ViewStyles of a `CalcHead` are the following: `VIEWSTYLE_FULL`, `VIEWSTYLE_MODEL`, `VIEWSTYLE_SPECIFICATION`.

Again, the Dialog Guide does not know the mathematical meaning of the attributes of a `CalcHead`, but it uses the `Status` feedback to provide user guidance and decide about the completion of the Specifying Phase.

`Tactic`

A `Tactic` is presently implemented as a stub, adding the attributes (which are still questionable, too):

`Name` : To the user, every `Tactic` is identified by a `String` representing its name.

`Description` : Descriptive text according to 4.4.9.

and, as many Tactics use a theorem for rewriting:

`TheoremSymbolic` : A `Formula` as a symbolic representation of the theorem used for rewriting.

`TheoremName`

`TheoremDescription`

`Assumption`

`Assumption`s are presently implemented as a mere stub.

23.2.3 Subterms

Subterms of a `Formula` can be regarded as `Formula`s themselves. To navigate a `Formula` and select different subterms, the stub of a `TermSelector` class with methods `goRoot`, `goLeft`, `goRight`, `goDown` and `goUp` is provided for reference. These methods are inspired by the DOS-interface of the computer algebra system Derive.

23.3 Iterators for Navigating the `CalcTree`: `ICalcIterator`

Methods offered by iterators include the following:

`moveRoot` moves an iterator to the start of the calculation, i.e. the root of the `CalcTree`. Note that the root element of a `CalcTree` is always a `CalcHead` (see section 16.5.3).

`moveCalcHead` moves the iterator to the `CalcHead` specifying the part of the calculation the iterator is presently referencing. Note that this is not necessarily the root `CalcHead`, as subproblems are integrated into one single `CalcTree` (see section 16.5.3).

`moveUp` and `moveDown` move the iterator to the previous or next element in the tree, respectively.

`moveLevelUp` and `moveLevelDown` move the iterator one nesting level out of one level deeper into the tree. These methods might seem redundant to `moveCalcHead`, but they are not as there may be mathematical structures other than subproblems branching the `CalcTree`.

`getElement` returns the `CalcElement` referenced by the iterator, a `Formula` or a `CalcHead`.

`getTactic` returns the `Tactic` being applied to the currently referenced `Formula`. `getTactic` supersedes the previous idea of a `moveTactic` method, as it has been decided to couple `Formula` and `Tactic` more tightly and to reference `Tactic`s by referring to the `Formula` they are applied to.

All move methods return a `boolean` being `true` on success and `false` if there is no appropriate element to move to.

23.4 Data Types Used for Communication

The communication between the Dialog Guide and the front-end is the same for both parts: between KnowledgeBrowsers and BrowserDialog as well as between WorkSheetDialog and WorkSheet including the CalcHeadPanel. In the sequel we only mention the latter part.

23.4.1 Events

Events are used whenever a component needs to notify other components of something happening without knowing which component will eventually handle the situation and without having to wait until processing the event has finished. Apart from receiving requests from the user, events are used for notifications abouts changes in the state of the `CalcTree`. Events are sent to components having registered as being interested in notifications using the Observer pattern. Events as used by *ISAC* resemble the Command pattern [GHJV95].

Changes in a Calculation: `CalcChangedEvent`

A `CalcChangedEvent` is sent by the Math Engine every time the calculation represented by a `CalcTree` changes to notify other compoments to look for updates. Normally, the WorkSheetDialog listens to `CalcChangedEvent`s from the Math Engine and passes them on to the Presentation Layer, if appropriate. The WorkSheetDialog might decide not to pass the events if the changes affect only details of the calculation filtered away by the WorkSheetDialog.

Methods offered by `CalcChangedEvent`:

`getLastUnchangedFormula` returns an iterator pointing to the last `Formula` in the `CalcTree` not affected by the change to avoid examining parts of the tree which have not changed anyway.

`getLastDeletedFormula` returns an iterator pointing to the last `Formula` to be deleted below the `getLastUnchangedFormula` (i.e. we assume a coherent sequence of formulae within the `CalcTree`). If there is nothing to delete, the iterator points at `getLastUnchangedFormula`.

`getLastGeneratedFormula` returns an iterator pointing to the last `Formula` generated within this step; there may be several `Formula`s between `getLastUnchangedFormula` and `getLastGeneratedFormula`, even on different levels of the `CalcTree`.

`isCompleted` returns `false`, if the Math Engine has been requested to calculate more than one step to indicate that the `CalcTree` has changed but the request is not completed yet.

`Message` in case the preceding request for changing the calculation failed (WN0503 under consideration; concerns detailed design of the Dialog Guide).

Intervention by the User: `UserAction`

Every time a user request cannot be handled by the Presentation Layer alone, a `UserAction` event is sent to registered listeners, which is usually the WorkSheetDialog. See appendix C, p.110 in [Kre05] for a list of requests recognised by the WorkSheetDialog.

`getActionID` returns an `int` identifying the user's intervention or request.

Several classes are derived from `UserAction` if additional information has to be passed:

`UserActionOnIterator` carries an iterator, e.g. to move the active formula.

`UserActionOnCalcElement` carries any `CalcElement`, e.g. to replace the active formula by a version edited by the user.

`UserActionOnCalcHead` refers to a part of a `CalcHead` during the Specifying Phase, e.g. to have a part of the `CalcHead` completed by the Math Engine.

There are several analoguous `UserActionOn...`s for the BrowserDialogs.

####################WN051223 Manuel bitte: Kontext, und wie er fuer das CalcHeadPanel benutzt wird (durchreichen der Action WindowApplication - Worksheet - ???) ...

See 23.5.4.

Controlling the Presentation Layer: `UIAction`

####################WN051223 Manuel bitte folgendes updaten

Kontext analog zu UserAction (!??)

Although the naming is not adequate, `UserAction` objects are reused to send requests from the WorkSheetDialog to the Presentation Layer. Requests sent by the WorkSheetDialog include having the user edit a formula and activating or decativating buttons on the user interface (see appendix C.3, p.112 in [Kre05]).

See 23.5.3.

####################WN051223 Manuel bitte obiges updaten

23.4.2 Exceptions

Exceptions Handled by the WorkSheetDialog

Being embedded into a distributed system, `RemoteException`s from Java-RMI have to be handled by most components, including the WorkSheetDialog.

`NotInSpecificationPhaseException` is thrown by the Math Engine and probably more exceptions originating from the Math Engine will be added in the future.

Exceptions Thrown by the WorkSheetDialog

`DialogProtocolException` is thrown if a request arrives out-of-order, e.g. calculating without having finished the Specification Phase. Every `DialogProtocolException` carries additional information about which `UserAction` caused the error and which Dialog Phase the WorkSheetDialog was in when the error occured. Several exceptions have been derived from `DialogProtocolException` to indicate special situations:

`DialogMathException` is thrown if the WorkSheetDialog cannot recover from an exception thrown by the Math Engine.

`DialogNotImplementedException` is thrown if a requested feature is not yet implemented by the WorkSheetDialog.

`DialogUnknownActionException` is thrown if the WorkSheetDialog has been passed an unknown `UserAction`.

23.5 Interfaces used by the Dialog Guide

The interfaces `IToCalc` and `IToUser` are used for operating on the calculation.

The interfaces `IWorkSheetDialog` and `IToGUI` are used for guided interaction with the user and are implemented by the WorkSheetDialog and the Presentation Layer, respectively.

`IToGUI` is used by the BrowserDialog for the KnowledgeBrowsers as well.

23.5.1 Communicating Towards the Calculation: `IToCalc`

This interface contains two methods for accessing a calculation.

`addDataChangeListener` registers a component implementing the `IToUser` interface as listener for `CalcChangedEvent`s to be notified of changes in the calculation.

`iterator` returns an iterator for navigating a calculation.

Other methods of this interface provide direct manipulation of a `CalcTree`. These methods are intended to be used by the WorkSheetDialog but not directly by the Presentation Layer, as user access to these operations is abstracted in the requests listed in appendix C, p.110 in [Kre05].

`getActiveFormula` returns a `CalcIterator` indicating the position in a calculation where modifications take place. This includes editing as well as the starting point for automatic calculation.

`moveActiveFormula` sets the position where modifications to the calculation will be applied.

`appendFormula` appends a new, empty `Formula` to a calculation.

`replaceFormula` replaces the active formula, if the correctness of the new `Formula` can be confirmed.

`fetchProposedTactic` asks for the next `Tactic` the Math Engine would apply.

`setNextTactic` sets the next `Tactic` to be applied.

`modifyCalcHead` applies modifications to a `CalcHead` and asks for feedback from the Math Engine.

`completeCalcHead` completes a `CalcHead` automatically.

`autoCalculate` calculates a number of steps automatically.

23.5.2 Communicating Towards the User: `IToUser`

This interface is implemented by the WorkSheetDialog to be able to be notified by the Math Engine about changes in the calculation as well as by the Presentation Layer to be able to be notified by the WorkSheetDialog.

The only method `calcChanged` is passed a `CalcChangedEvent` as parameter. In the present implementation, the WorkSheetDialog passes the `CalcChangedEvent` to all registered listeners without filtering or other processing taking place.

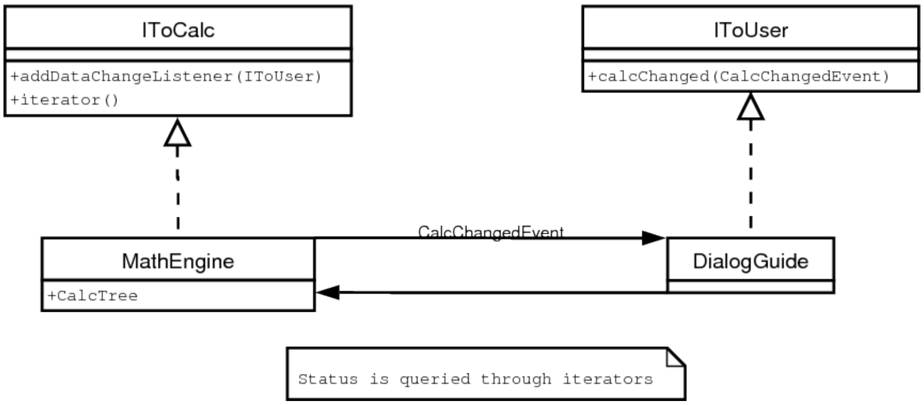

IToCalc
+addDataChangeListener(IToUser)
+iterator()
IToUser
+calcChanged(CalcChangedEvent)
CalcChangedEvent
MathEngine
+CalcTree
DialogGuide
Status is queried through iterators

Figure 23.2: The WorkSheetDialog(TODO.rename.DialogGuide) and the Math Engine communicating updates in a calculation using the Observer pattern

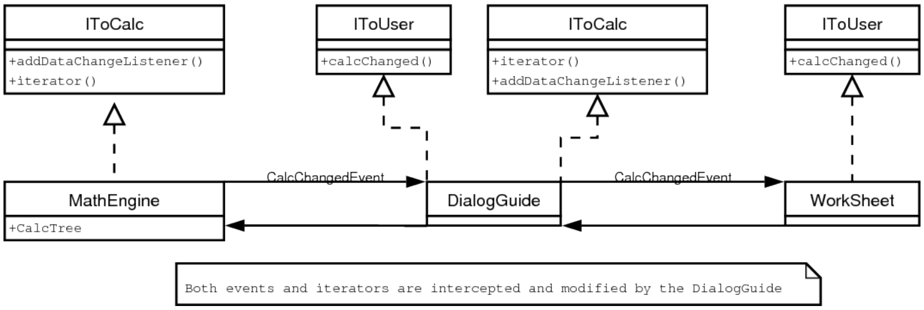


Figure 23.3: The WorkSheetDialog(TODO.rename.DialogGuide) intercepting communication between the Math Engine and the Presentation Layer using the Decorator and Mediator patterns

23.5.3 The Presentation Layer as Seen from the Dialog Guide: `IToUser`

This interface is implementedb by the Presentation Layer to provide a means for the Dialog Guide controlling the user interface as described in section 16.6.3. For instance, the BrowserDialog provides different buttons depending on whether a problem is shown with a plain model, or a model matched with an active `CalcHead`.In addition to that, the WorkSheetDialog can request actions from the user, such as editing a formula.

The means for interaction are passed to the presentation layer and withdrawn again by the methods `addUIElement` and `removeUIElement` respectively.

The only method concerning interaction,`doUIAction`, is passed a `UIAction` to be processed by a user interface as parameter.

23.5.4 The WorkSheetDialog as Seen from the Presentation: `IWorkSheetDialog`

`startCalculate` initialises the WorkSheetDialog by providing an identification of the user, a `Formalization` and `STARTFROM_EMPTY_WORKSHEET`, `STARTFROM_PROBLEM` or `STARTFROM_EXAMPLE` to indicate how much information will be available for user guidance during the Specifying Phase. Note that the user does not interact with this WorkSheetDialog until he chooses to start a calculation, so use cases UC 30.1.2 are handled outside the WorkSheetDialog.

The identification of the user is used to retrieve the appropriate `UserSettings` and `UserModel`. The `Formalization` is passed on to the Math Engine during initialisation.

After successfully contacting a Math Engine, the WorkSheetDialog enters the Specifying Phase or the Solving Phase, dependent on the UserSettings.

`registerUIControlListener` registers a component implementing the `IToWorksheet` interface as listener for `UserAction`s controlling a user interface.

`notifyUserAction` is the central method for communicating with the WorkSheetDialog. A `UserAction` as listed in appendix C in [Kre05] is passed to the WorkSheetDialog. The return value is `true` if the request has been accepted and will be processed and `false` if processing the request is denied. Processing a request can be denied according to the user's privileges or the user's experience, a decision the WorkSheetDialog takes after consulting the `UserSettings` and the `UserModel`. Note that denying a request is not an error condition from the WorkSheetDialog's point of view. An error condition would be a request impossible due to the WorkSheetDialog's state, such as not matching the current Dialog Phase. In this case, there has been an error in the communication between the WorkSheetDialog and the Presentation Layer and a `DialogProtocolException` is thrown.

Once a request is accepted, further action resulting from the request is communicated through `UserAction`s sent by the WorkSheetDialog or `CalcChangedEvent`s originating in the Math Engine.

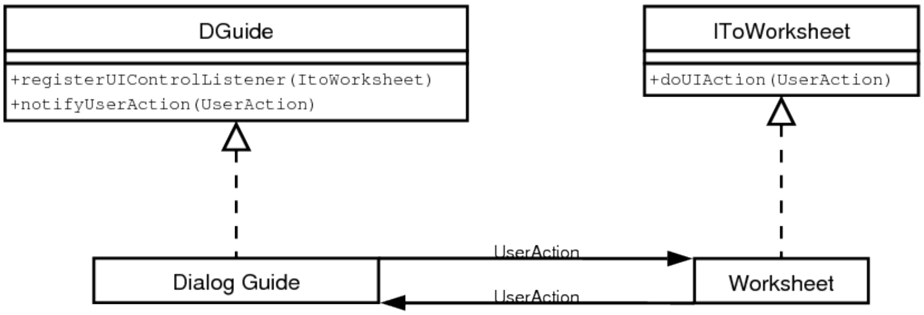


Figure 23.4: The WorkSheetDialog and the Presentation Layer communicating user interaction FIXME.WN0512 DialogGuid –¿ WorkSheet+Browsers: UIAction

23.6 Communicating with the `UserModel`

On the level of items in the Knowledge Base, such as Tactics, and the context of Dialog Atoms, the `UserModel` provides statistical data about prevoiusly recorded interactions with the user.

`getTouchedCount` returns the number of previously recorded interactions.

`getSuccessCount` returns the number of previously recorded successful interactions.

`getTouchedLast` returns the timestamp of the last interaction recorded.

`getSuccessLast` returns the timestamp of the last successful interaction recorded.

`getTimeSpentAvg` returns the average time the user spent with the interaction. Time is provided ny the SessionHandler (i.e. by the system running this module).

`getTimeSpentMax` returns the maximum time the user spent with the interaction.

Statistical data are gathered as the WorkSheetDialog announces the start and end of every interaction between user and Knowledge Base.

`startInteraction` is called with identifications of the Knowledge Base item and the Dialog Atom used.

`endInteraction` is called with an `int` indicating the success of the interaction. As there is no appropriate measure for the success of an interacton presently available, the values `0` and `1` will be used for prototyping.

2

23.7 The important classes

The figure fig.23.5 p.153 shows the classes involved in the WorksheetDialog.

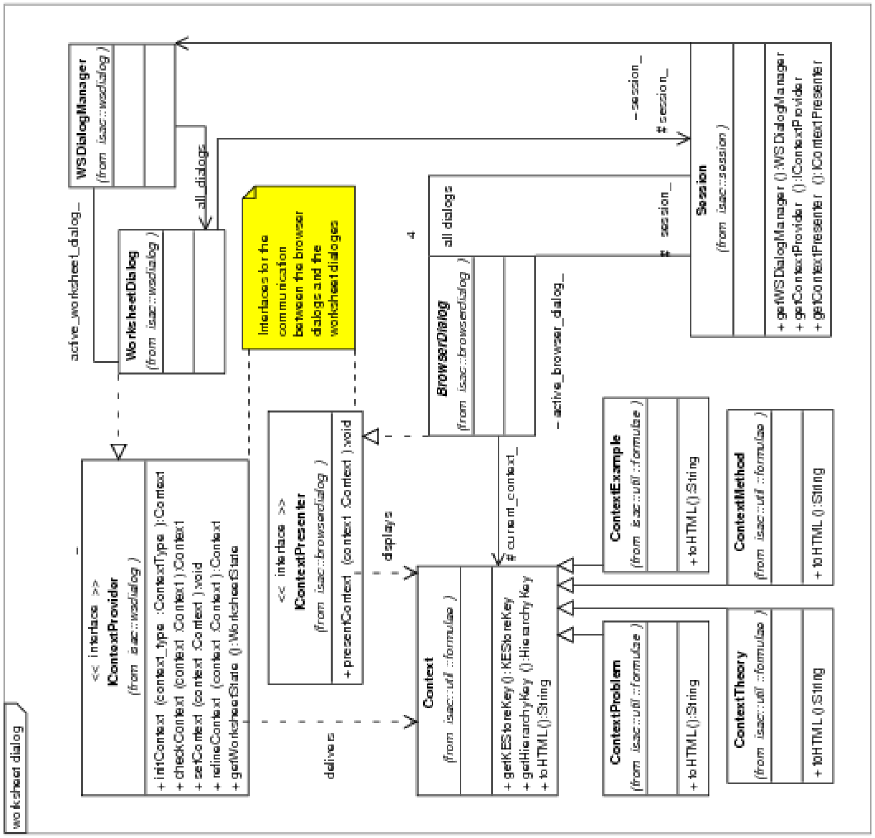

worksheet dialog
<< interface >>
IContextProvider
(from isac::wsdialog)
+ initContext (context_type :ContextType):Context
+ checkContext (context :Context):Context
+ setContext (context :Context):void
+ refineContext (context :Context):Context
+ getWorksheetState ():WorksheetState
delivers
WSDialogManager
(from isac::wsdialog)
active_worksheet_dialog_
WorksheetDialog
(from isac::wsdialog)
all_dialogs
Interfaces for the communication between the browser dialogs and the worksheet dialoges
<< interface >>
IContextPresenter
(from isac::browserdialog)
+ presentContext (context :Context):void
displays
Context
(from isac::util::formulae)
+ getKEStoreKey ():KEStoreKey
+ getHierarchyKey ():HierarchyKey
+ toHTML():String
BrowserDialog
(from isac::browserdialog)
4
all dialogs
- active_browser_dialog_
# current_context_
- session_
# session_
# session_
Session
(from isac::session)
+ getWSDialogManager ():WSDialogManager
+ getContextProvider ():IContextProvider
+ getContextPresenter ():IContextPresenter
ContextExample
(from isac::util::formulae)
+ toHTML ():String
ContextMethod
(from isac::util::formulae)
+ toHTML ():String
ContextProblem
(from isac::util::formulae)
+ toHTML ():String
ContextTheory
(from isac::util::formulae)
+ toHTML ():String
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 23.5: The dialoges in a session

---

[2] End of copy from [Kre05] p.89

Chapter 24

Worksheet

The WorkSheet provides the user with the means to view a calculation or to interactively construct a calculation. For specifying a problem or a subproblem there is an additional view called CalcHeadPanel.

24.1 Communication with the WorkSheetDialog

The connection to the WorksheetDialog has already been established at initialization time as described in section 21.3.2 step 21.3.2. Each Worksheet holds exactly one connection to a WorksheetDialog and a WorksheetDialog is also connected to exactly one Worksheet (1:1 relation). This relationship is necessary as the WorksheetDialog needs to record the user actions in the user history, and the WorksheetDialog also holds the complete *calculation tree* for the corresponding calculation in the Worksheet.

The entire connection goes over Java RMI interface thus enabling that the WorksheetDialog and Worksheet be on physically separate machines. The class WorksheetRMI wraps up the RMI interfaces on the Worksheet side.

The WorksheetDialog communicates with the Worksheet over two interfaces:

- IToGui - implements the handling of UIActions through the *doUIAction(UIAction)* method (see section 23.5).
- IToUser - implements the handling of Calculation Events through the *calcChanged(CalcChangedEvent)* method (see section 23.5.2).

These interfaces are implemented by the WorksheetRMI wrapper class, which passes the UIAction and CalcChangedEvent objects to the aggregated Worksheet.

24.2 The classes or the WorkSheet

The Worksheet is the main component within the graphical user interface and is responsible for the interaction with the user while doing a calculation. The main task of the Worksheet is the representation of the calculation steps. Each step is stored as a node in the CalcModelHierarchy (see section 24.2.4) and these nodes are displayed in a hierarchical structure by the Java JTree component. How detailed each step is displayed on the worksheet is agreed in accordance with the WorksheetDialog and depends on the user history. The communication from Worksheet to WorksheetDialog is performed over Java RMI. Each user action is processed in the Worksheet and if the requested action needs the functionality of the mathematical engine then the request is forwarded to the WorksheetDialog. In the current phase, there are very few user actions that are handled solely by the Worksheet. Upon receiving the user action, the WorksheetDialog handles the request (e.g. forwards calculation tasks to the mathematical engine) and returns the result back to the Worksheet.

As shown in figure 24.1, the class `Worksheet` is designed around a lot of helper components.

These components are described in more detail in the following sections.

24.2.1 TreeModel

The mathematical calculation is represented in ISAC SML core as a hierarchical tree structure. The perfect component for rendering such structures is the SWING JTree component. However in order to render such data, this component requires the data provider to implement the `TreeModel` interface.

In the ISAC system the `TreeModel` Interface is implemented by an abstract class `AbstractTreeModel`. This class implements the basic functionality of the interface which is universal for all TreeModels, regardles of the data types of their Nodes. The `AbstractTreeModel` is inherited and extended by the `CalculatonModel` class which represents the bridge between ISAC's calculation hierarchy and the JTree display component.

The Figure 24.2 shows the class diagram of the Tree Model part of the Worksheet.

The `CalculationModel` class serves as an event wrapper for the the `CalcModelHierarchy` where all calculation nodes are stored.

24.2.2 CalcModelHierarchy

As previously mentioned the `CalcModelHierarchy` stores each node of the calculation tree in a representation that can be displayed by the standard SWING component `JTree`.

Figure 24.3 shows the communication required to propagate data changes to the `JTree`. The WorksheetDialog sends a `calcChangedEvent()` (over RMI) which notifies the Presentation Layer that a node has been added to the calculation tree. The `CalcModelHierarchy` accepts the added node and stores it to its

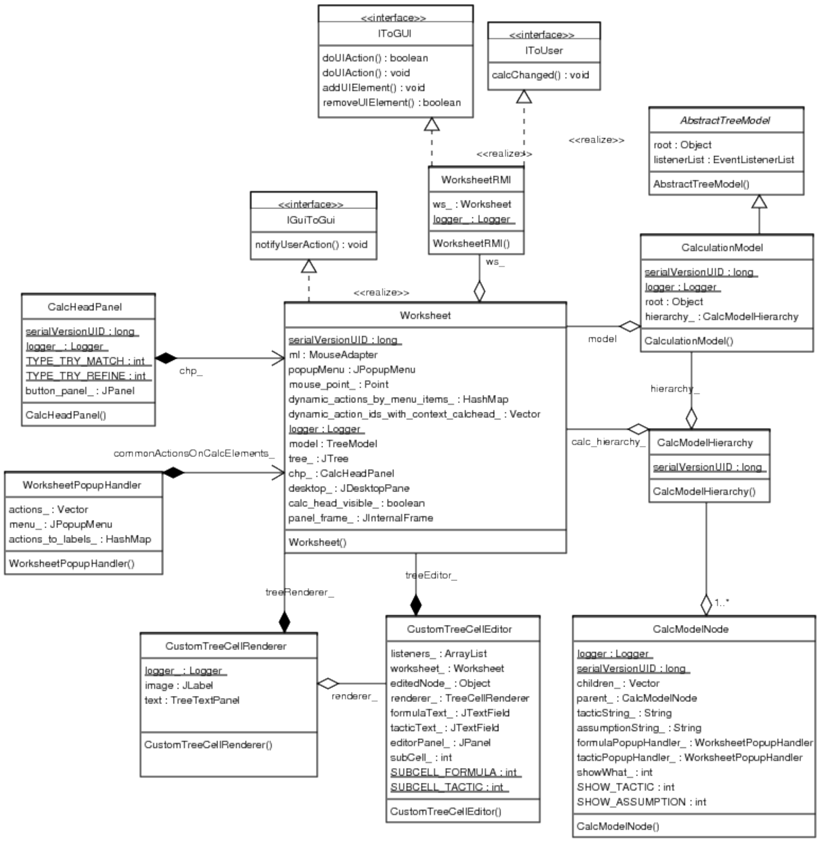
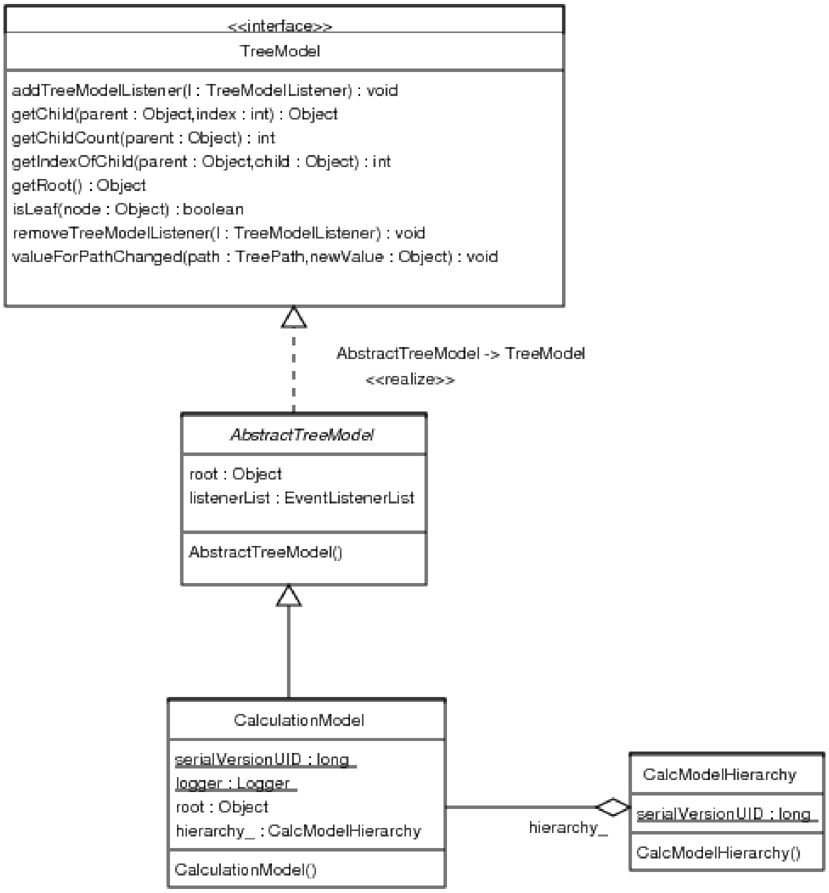


Figure 24.1: The worksheet class and its dependencies.

internal tree data structure. After storing the node, the `CalcModelHierarchy` sends message to the `CalculationModel` about the added hierarchy node. Finally the `CalculationModel` notifies all registered listeners (including the `JTree`) of the inserted node.

The communication procedure in the opposite direction is shown on Figure 24.4. The User has changed a node in the `JTree` so the component notifies its listeners. However, the `CalculationModel` ignores the `valueForPathChanged()` event, because the user changes are not stored in the calculation model hierarchy unless they are authorised by the mathematic engine. The `Worksheet` class is registered as the JTree listener and it propagates the change event to the WorksheetDialog. The WorksheetDialog then checks the result with the mathematical engine and the results of the requested operation are delivered to the `Worksheet` as a `calcChangedEvent`.

The `CalcModelNodes` are stored in the `CalcModelHierarchy` object in a tree like structure where each node holds references to its children as well as its



<<interface>>
TreeModel
addTreeModelListener(l : TreeModelListener) : void
getChild(parent : Object,index : int) : Object
getChildCount(parent : Object) : int
getIndexOfChild(parent : Object,child : Object) : int
getRoot() : Object
isLeaf(node : Object) : boolean
removeTreeModelListener(l : TreeModelListener) : void
valueForPathChanged(path : TreePath,newValue : Object) : void
AbstractTreeModel -> TreeModel
<<realize>>
AbstractTreeModel
root : Object
listenerList : EventListenerList
AbstractTreeModel()
CalculationModel
serialVersionUID : long
logger : Logger
root : Object
hierarchy_ : CalcModelHierarchy
CalculationModel()
hierarchy_
CalcModelHierarchy
serialVersionUID : long
CalcModelHierarchy()

Figure 24.2: The Tree Model class diagram.

parent.

The `CalcModelNode` object holds an instance of the formula as a `CalcElement` as well as the string representations of the tactic and assumption (used for display purposes). Each `CalcModelNode` has two `WorksheetPopupHandler` objects which store the available actions the user can perform on a certain formula. These actions are currently rendered as a popup menu accessible with the right click on formula or tactic.

24.2.3 CustomTreeCellRenderer

The JTree SWING Component provides the programmer with a possibility to customise the rendering of single cells. This possibility is currently used in the ISAC system in order to render both the tactic (or assumption) and a formula

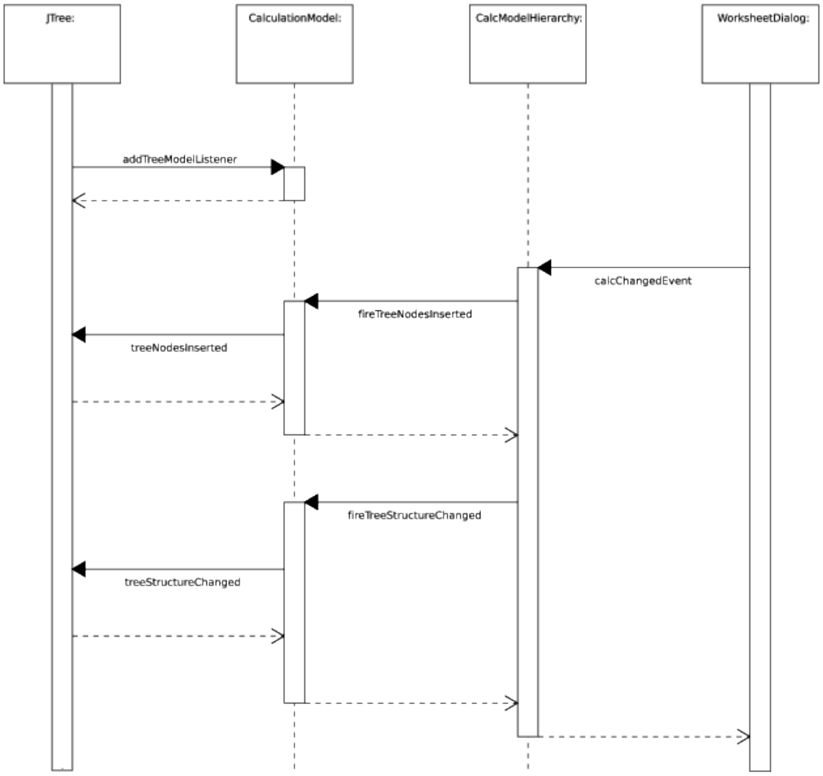


Figure 24.3: Communication flow necessary to propagate the node inserting event from the calculation tree to the presentation component.

in a single node. Therefore, each node consists of two `JLabel` objects which are used to display the corresponding content. In future extensions this class could implement some renderer capable of graphical representation of ISAC formulae.

24.2.4 CustomTreeCellEditor

Similar to the `CustomTreeCellRenderer` the `CustomTreeCellEditor` provides for a way to edit the nodes in the JTree. Currently, the `CustomTreeCellEditor` consists of two `JTextArea` components capable of editing multi-line text. The choice which line of text to edit comes from `Worksheet`. This class catches the mouse events, calculates the position of the mouse click and notifies the `CustomTreeCellEditor`. Needles to say the future enhancements should provide a graphical editor and implement its own mouse click handler.

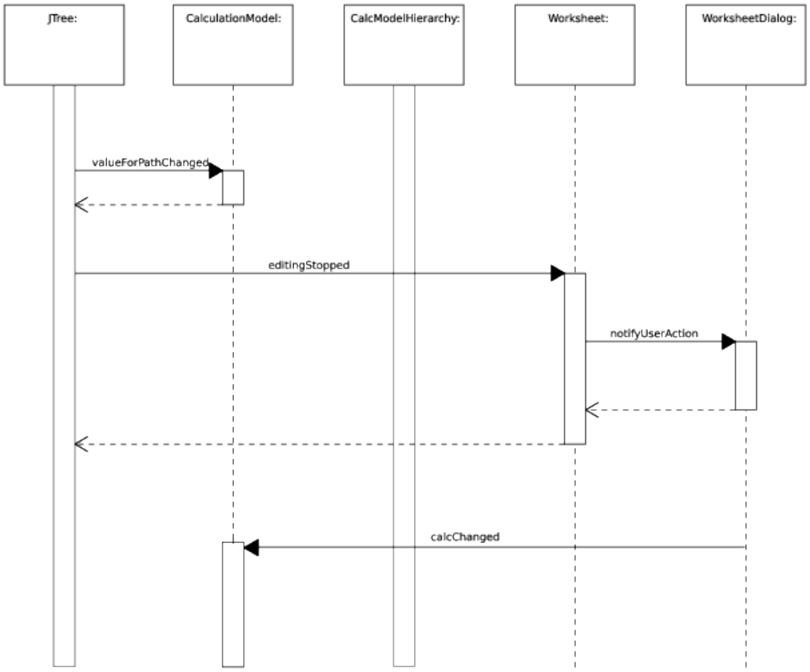


Figure 24.4: Communication flow necessary to propagate the change of the node in the presentation component to the calculation tree.

24.2.5 CalcHeadPanel

The CalcHeadPanel is displayed by the GUI if the user wants to specify a (sub-)problem or if the WorkSheetDialog decides for an explicit specifying phase. Additionaly, the user can choose to open the CalcHeadPanel to get an insight into the specification or the model of the problem started from the example collection.

The CalcHeadPanel is implemented in the main `CalcHeadPanel` class and a couple of presentation components. The class diagram of these components is shown on the Figure 24.5.

The `Worksheet` communicates with the `CalcHeadPanel` by directly calling its two methods:

- `fillCalcHeadFromTextFields` and
- `fillTextFieldsFromCalcHead`

These methods take single parameter an instance of the CalcHead to display or to fill. As the user ends the modeling/specification phase, the Worksheet asks

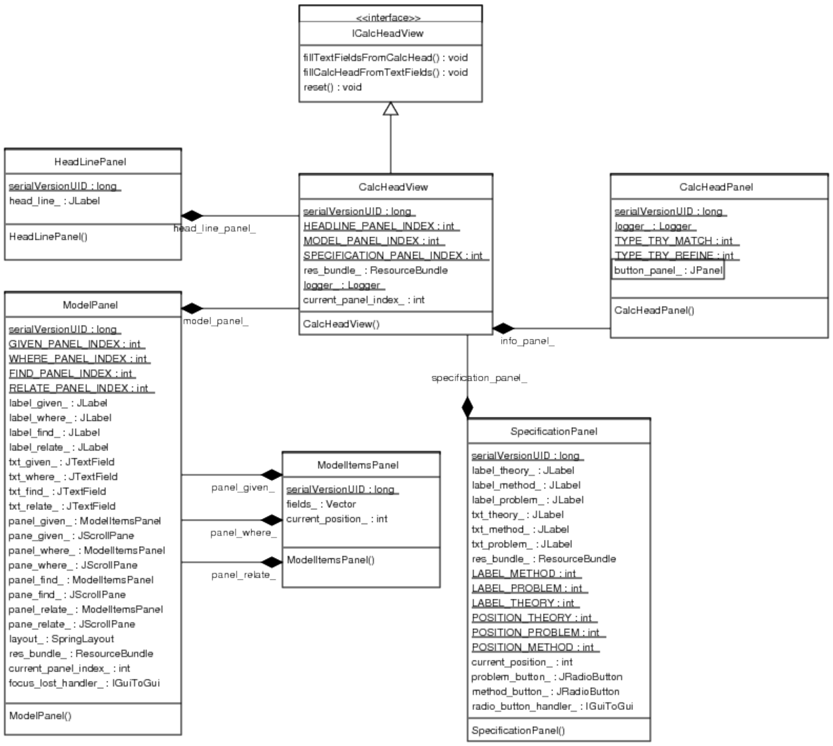


Figure 24.5: Communication flow necessary to propagate the change of the node in the presentation component to the calculation tree.

the CalcHeadPanel to fill out the CalcHead that gets sent to the WorksheetDialog. The second method is used to initialise the CalcHeadPanel from an existing CalcHead, when the user has started the calculation from an example, but wishes to see the model and the specification.

The `CalcHeadPanel` can display the following ViewStyles:

- `VIEWSTYLE_FULL`
- `VIEWSTYLE_MODEL`
- `VIEWSTYLE_SPECIFICATION`

For each ViewStyle of the CalcHeadPanel, there is a corresponding presentation component. The full ViewStyle simply shows the both components.

ModelPanel

As the name `ModelPanel` indicates, this component is used to display the model data of the CalcHead. The model consists of four fields : given, where, find and relate. The display and editing of these fields is realised through four `JTextArea` components.

What is more the ModelPanel gets the composite data from the Worksheet and can show the user whether the given data is correct, complete, missing, superflous etc.

SpecificationPanel

The `SpecificationPanel` allows the user to view or edit the theory, method or the problem connected to the edited CalcHead. It is implemented in the same way as the `ModelPanel`.

Chapter 25

KEStore

Chapter 26

Knowledge Browser

26.1 The relation between the browsers

The architectural design in sect.18.3 on p.102 relates browsers to the respective dialogs. The relation is the reason for the structure of the browsers detailed here.

The figure fig.26.1 p.169 shows the class diagram for the browsers.

What is seen by the user has a uniform structure for examples and the knowledge (theories, problems, methods).

26.2 The classes for a browser

The HierarchyBrowser's task is to display the hierarchy which was generated by the XMLHierarchyBrowser in a way that the user can browse through this hierarchy.

The HierarchyBrowser was designed (figure 26.2) according to the model architecture that is used in the programming language *Java*[1] within the *Swing*[2] library.

This design principle makes it possible to define a model interface (*BrowserTreeModel*) that mainly defines accessor methods. It defines no operations to manipulate the structure of the data – no methods to insert, remove or change data items in the model. This makes implementing a model and using it with a component (in the *ISAC* design this is *TabPanel*, which uses a standard component, namely *JTree*, to render the data of the model) somewhat different from using the more conventional MVC (model-view-controller) design principle. This difference allows greater flexibility. It means that different components can

---

[1] Flanagan [Fla02] provides an essential quick reference about the Java programming platform

[2] Flanagan [Fla99] also provides a reference about the *Java Foundation Classes* which includes Swing.

use the existing data structure without having to make modifications to that structure to make it conform to the required interface.

The HierarchyBrowser has to display the entries of the three parts of the knowledge base and the entries of the example hierarchy. As these hierarchies are logically connected and the user may want to switch fast between them, the HierarchyBrowser uses the so-called *Tabbed Window* technology. Each hierarchy is displayed in an own tab window. This lets the user switch back and forth between the different knowledge base hierarchies.

26.2.1 TabPanel

TabPanel is designed as an abstract class and it generalizes the drawing of the different hierarchies, as this is done the same way for each of the hierarchies. The drawing itself of the hierarchy can be done by a standard component that almost each windowing toolkit framework provides. TabPanel only coordinates the initialization of the model which needs the hierarchy data for a correct initialization. The model must then be connected to the component that is responsible for the drawing of the data that the model provides.

26.2.2 BrowserTreeModel

The BrowserTreeModel provides the component that takes over the rendering of the hierarchy with data from the different hierarchies. As the BrowserTreeModel does not provide any operations to manipulate the structure of the data, another mechanism must be used.

The behavioral design pattern *Observer*, described by Gamma et al. [GHJV95] and implemented as *Event-Listener-Concept* within the Java programming platform, is the chosen mechanism within the *ISAC* architecture. The notation of the Java programming platform is used within *ISAC*; therefore, the names *listener* and *event source* are used afterwards instead of *observer* and *subject*.

Figure 26.3 shows the communication that is required to propagate data changes to the renderer. Each step that is necessary is described in more detail in the following:

1. **Register Listener**
   Each application or object that needs to be informed about changes in the structure of the data must register itself as a listener at the rendering component.

2. **Receive Event**
   The application receives events from the rendering component and determines the actions to take.

3. **Change Data**
   The application changes the underlying data model, in this case the hierarchy of the knowledge base.

4. **Report Change**
   The application reports the changes to the BrowserTreeModel.

5. **Notify Rendering Component**
   The BrowserTreeModel notifies its listeners (the rendering component automatically registered itself as a listener for the BrowserTreeModel component) of the change.

6. **Request Data**
   The rendering component asks the BrowserTreeModel for data that it has to display.

7. **Delegate Request**
   The BrowserTreeModel, which has a connection to the hierarchy, delegates the request for data to the hierarchy.

8. **Return Data**
   The hierarchy returns the complete data structure to the BrowserTreeModel.

9. **Return Result**
   The BrowserTreeModel sends back the data to the rendering component, which can now display the new data.

26.2.3 BrowserPanel

The three BrowserPanel classes—ProblemBrowserPanel, MethodBrowserPanel and ExampleBrowserPanel—extend the functionality of the TabPanel class.

This extra functionality is described in detail for each class in the following:

- **ProblemBrowserPanel**
  As described in the Use Cases **??** and **??**, the user can either match the model of the current example against a problem in the problem hierarchy or try to refine the model of the current example with the help of the problem hierarchy and *ISAC*'s mechanism to automatically find the best matching problem. The user can choose the desired action with a right click in the ProblemBrowserPanel.

  As soon as the user accepts the matched model he can start or continue (if the user refined or matched the model of a subproblem) the solving phase. The user request is then forwarded to the Worksheet, which takes over the control during the solving phase.

- **MethodBrowserPanel**
  For MethodBrowserPanel currently no special functionality is described in the requirements; therefore, the MethodBrowserPanel at the moment only shows the content of the XML-file for a specific method if the user selects one.

- **ExampleBrowserPanel**
  As described in Use Case **??**, the user can start a calculation from the example hierarchy. If the user selects an example that he wants to calculate then a formalization (see section **??**) is generated from the XML file that describes this example. The WindowApplication is responsible for opening a new Worksheet that takes as parameter the formalization.

Note that for the theories part of the knowledge browser there is currently no TheoryBrowserPanel designed, for the same reason as for the XMLTheoryParser. Nevertheless, the TheoryBrowserPanel can every time be included as a subclass of TabPanel.

26.3 Implementation details

The HierarchyBrowser renders the parsed hierarchy of the XMLHierarchyParser with the help of the Java `JTree` object. To use `JTree` for this task there are some steps necessary that are described in the following sections. Figure 26.4 shows the result of the rendering by `JTree`.

26.3.1 HierarchyNodes

Each `NODE` tag (see listing 21.1) represents an entry in the hierarchy and is converted to an object that is derived from `HierarchyNode`. `HierarchyNode` provides *getter* and *setter* methods for `ID` and `CONTENTREF`. The nodes of the example and method hierarchy also have additional tags to `ID` and `CONTENTREF` for each additional tag the objects provide *getter* and *setter* methods.

26.3.2 Hierarchy

The `HierarchyNode`s are stored in the `Hierarchy` object. The `Hierarchy` object is needed by the `BrowserTreeModel` object. `BrowserTreeModel` implements the Java `TreeModel`, which is necessary for the Java `JTree` object to render the different hierarchies in a tree representation.
[3]

26.4 Minibrowser

Displaying HTML content in a Java GUI is in no big problem since most of the `javax.swing.*` components, which are used to display plain text, are also able to display HTML content. The task of displaying HTML content is done by the `Minibrowser` as shown in figure 26.1. The `Minibrowser` is an extension of the `javax.swing.JEditorPane`. The `JEditorPane` is per default able to deal with the following content:

---

[3] Begin of [Kö06] p...– p....

- text/plain
- text/html
- text/rtf

The type of content can be selected with the `setContetType()` method. There are two methods for displaying HTML content:

- **setText(String html)**: shows the passed HTML string directly. The disadvantage of this method is, that the HTML content has to be loaded into a string manually. According to SR.**??**, urls pointing to the local file system must work as well as urls pointing to documents available over the World Wide Web via http. Thus, an extra loading-class would be necessary to get the HTML content independent from the location. Additionally, the whole user interaction should not block while the file is loaded, so the loading-class would have to run in an extra thread, as required in SR.12.1.4. All the problems of thread-synchronization would have to be solved. However, the big advantage would be, that the HTML string could be manipulated directly. The content of HTML tags could be replaced or modified by dynamically generated HTML parts with a single `String.replace()` command.
- **setPage(String url)** loads the content of the passed url. This works from the World Wide Web as well as from the local file system. The page can be loaded synchronously or asynchronously. Dynamic changes in the static HTML document are not that easy, but they are possible by use of a so called `EditorKit`. This is explained in section 26.4.2 in detail.

The use of the `setPage()` method clearly results in less work to be done, so it was declared to be the weapon of choice for the `MiniBrowser`.

26.4.1 The Processing of Links

Links are handled with so called `HyperLinkListeners`. Whenever a link in the `JEditorPane` is selected, the registered `HyperLinkListener` is called. In our case, the `BrowserFrame` implements the `HyperlinkListener` interface. It is added as the one and only `HyperlinkListener` of the `Minibrowser`. If a link is selected, the `hyperlinkUpdate()` method is called. The BrowserFrame creates and sends an `UserActionOnLink`. If a new page has to be displayed, and `UIActionOnLink` containing the link target gets to the `Minibrowser`. It is loaded by use of the `setPage()` method of the `JEditorPane` class.

26.4.2 Dynamic Modification of the Static HTML Content

According to SR.12.2.6, the content displayed in the knowledge browsers can be enriched with context dependent information. A context refers to the active formula in the active worksheet. The reader may have a look at figure 14.1 while reading the following explanation.

The worksheetdialog is the module which knows the active formula. It queries the math engine over the bridge for a specific context (a context to theories, problems or methods) for this formula. The math engine responses on this query with the XML representation of the desired context. The context is parsed, which leads to a `Context` object (either `ContextTheory`, `ContextProblem` or `ContextMethod`) carrying the information in attributes. This Context object is passed to the corresponding browserdialog. The browserdialog remembers the context as current context and forwards it to the browser.

As mentioned in section 26.4, the HTML content itself is never available in a string to be modified directly. The `JEditorPane` uses an `HTMLEditorKit` for displaying HTML content. The only possibility for manipulating the representation is, to use an `EditorKit`. The solution is shown in figure 26.5.

An `HTMLFactory` is used to create components to be displayed out of HTML elements. Instead of the standard `EditorKit`, a self made `EditorKit` can be used. A `MinibroserEditorKit` is derived from the `HTMLEditorKit`. The method `getViewFactory()` is overridden by a method which returns a self written view factory called `MiniBrowserViewFactory`. The `create()`-method is called with all HTML elements of an HTML document. The returned `View` is displayed. This `MiniBrowserViewFactory` contains a map between HTML tag strings and `IHTMLElementRenderer` interfaces. If an HTML tag is not in the map, the `create()` method returns the standard `View`. If the tag is in the map, the `renderElement()` method of the corresponding interface is called to get the `View`. The class behind the interface can create an own object, which is derived from the `ComponentView`. The `createComponent()` methode can now implement the representation of the HTML element just as desired.

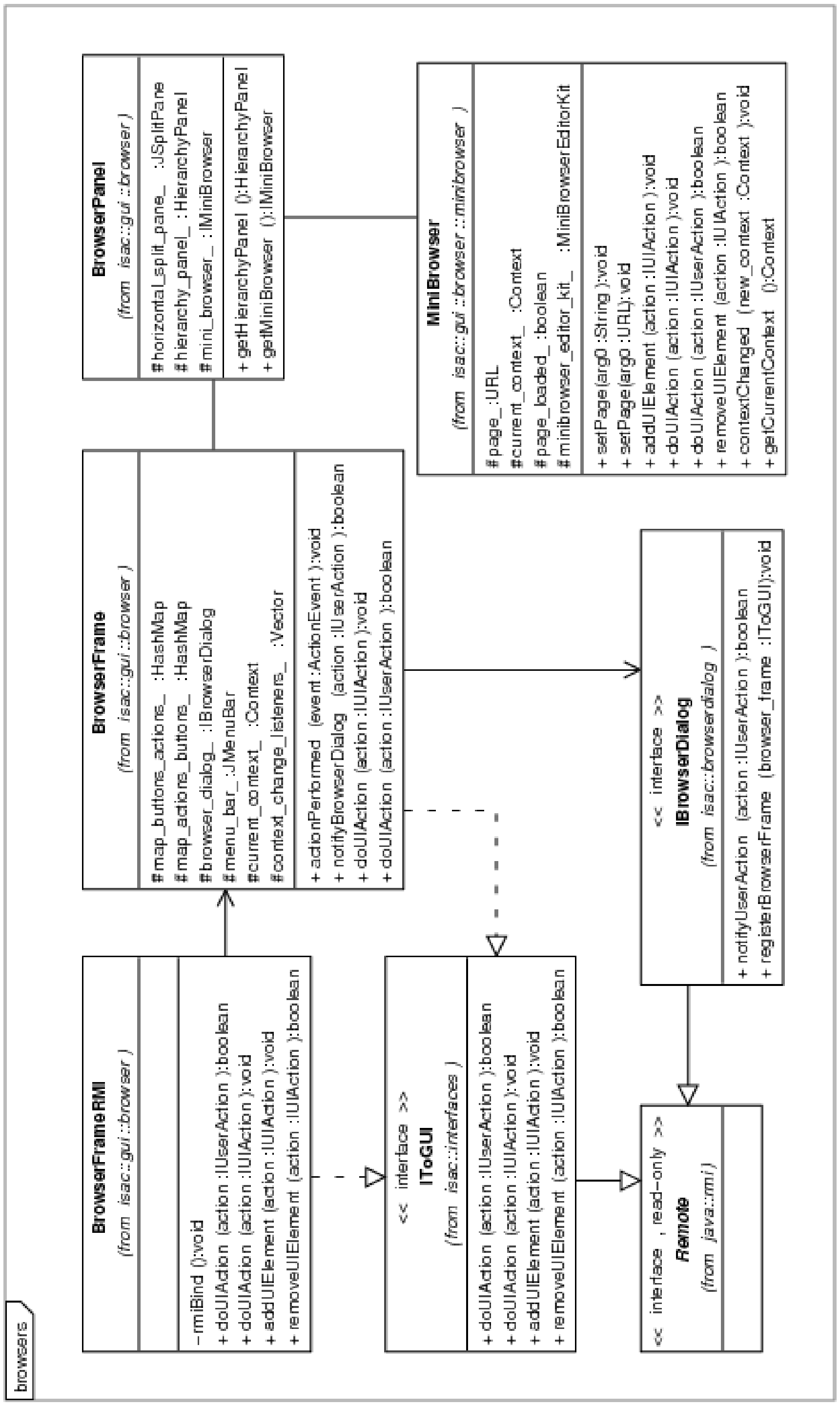
This design can be used very flexibly. Whenever an HTML tag has to be displayed in an special manner, a new `ContextRenderer` which implements the `IHTMLRenderer` interface and a `View` which creates the `Component` has to be programmed. The tag to be processed and the according `Renderer` has to be inserted into the map by use of the `addElementPresenter()` method of the `MiniBrowserViewFactory`.

In the current state of the *ISAC*-software, this feature is only used to replace everything which is inside a `<class="context"></class>` tag with the context information from the math engine. The string `'context'` is inserted together with a `ContextRenderer` into the map of the `ViewFactory`. This class creates a `ContextView`. The `createComponent()` method of this class returns a `Component` displaying the HTML representation of the (dynamic) context instead of the static content between the `<class="context"> </class>` tags. The `toHtml()` method is called on the `current_context_`. This method returns an HTML string presenting the attributes of the context object.

[4]

---

[4] End of [Kö6] p....– p....


browsers
BrowserPanel
(from isac::gui::browser)
# horizontal_split_pane_ :JSplitPane
# hierarchy_panel_ :HierarchyPanel
# mini_browser_ :IMiniBrowser
+ getHierarchyPanel ():HierarchyPanel
+ getMiniBrowser ():IMiniBrowser
BrowserFrame
(from isac::gui::browser)
# map_buttons_actions_ :HashMap
# map_actions_buttons_ :HashMap
# browser_dialog_ :IBrowserDialog
# menu_bar_ :JMenuBar
# current_context_ :Context
# context_change_listeners_ :Vector
+ actionPerformed (event :ActionEvent):void
+ notifyBrowserDialog (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ doUIAction (action :IUserAction):boolean
MiniBrowser
(from isac::gui::browser::minibrowser)
# page_ :URL
# current_context_ :Context
# page_loaded_ :boolean
# minibrowser_editor_kit_ :MiniBrowserEditorKit
+ setPage(arg0 :String):void
+ setPage(arg0 :URL):void
+ addUIElement (action :IUIAction):void
+ doUIAction (action :IUIAction):void
+ doUIAction (action :IUserAction):boolean
+ removeUIElement (action :IUIAction):boolean
+ contextChanged (new_context :Context):void
+ getCurrentContext ():Context
BrowserFrameRMI
(from isac::gui::browser)
− rmiBind ():void
+ doUIAction (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ addUIElement (action :IUIAction):void
+ removeUIElement (action :IUIAction):boolean
<< interface >>
IToGUI
(from isac::interfaces)
+ doUIAction (action :IUserAction):boolean
+ doUIAction (action :IUIAction):void
+ addUIElement (action :IUIAction):void
+ removeUIElement (action :IUIAction):boolean
<< interface >>
IBrowserDialog
(from isac::browserdialog)
+ notifyUserAction (action :IUserAction):boolean
+ registerBrowserFrame (browser_frame :IToGUI):void
<< interface , read−only >>
Remote
(from java::rmi)

Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 26.1: The class diagram for the browsers

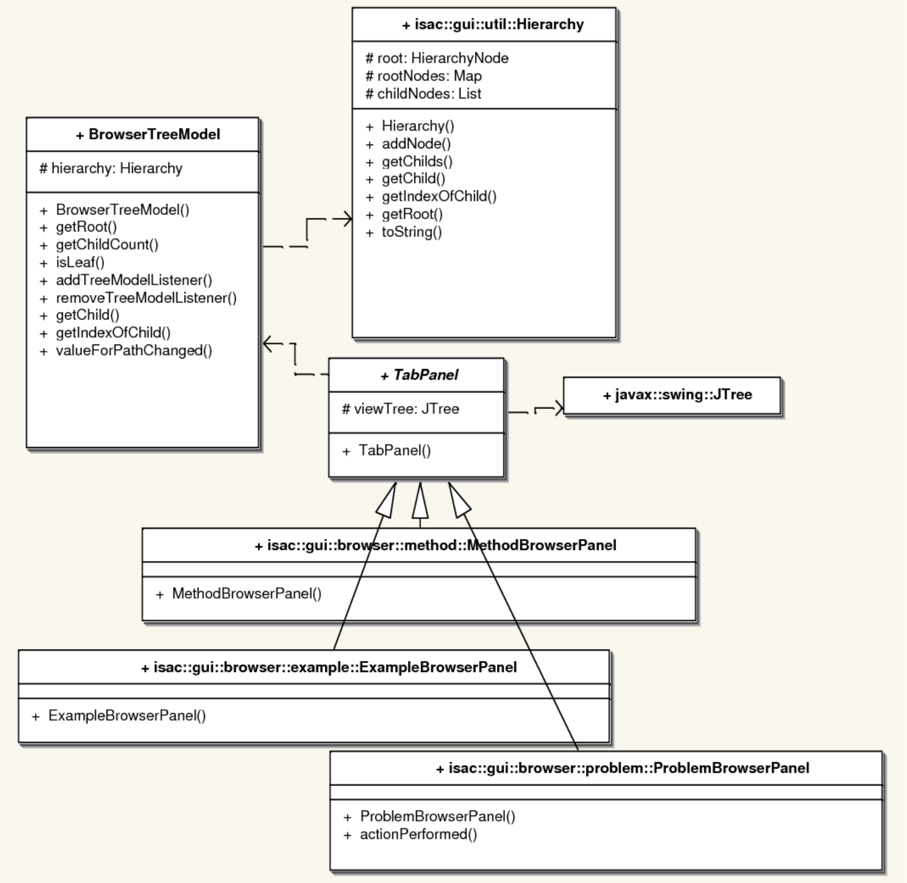


Figure 26.2: HierarchyBrowser using a more conventional MVC (model-view-controller) architecture

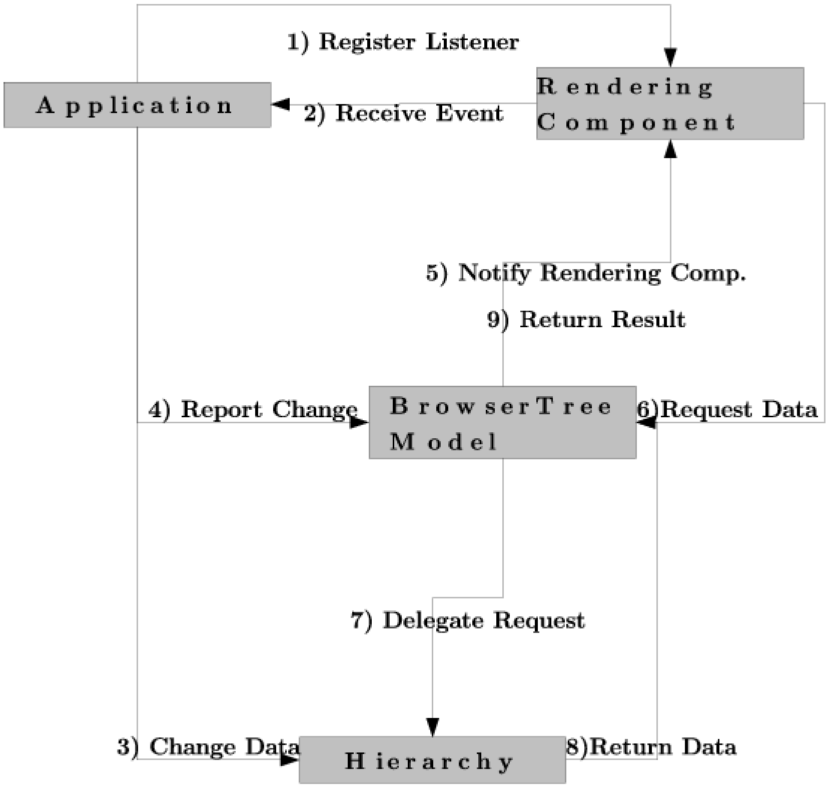


Figure 26.3: Communication flow that is necessary to propagate changes in the structure of the data to the rendering component.

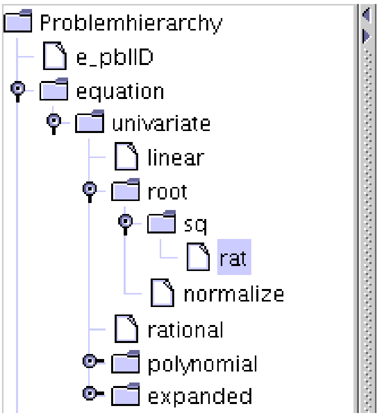


Figure 26.4: The problem hierarchy converted from XML representation and rendered by `JTree`

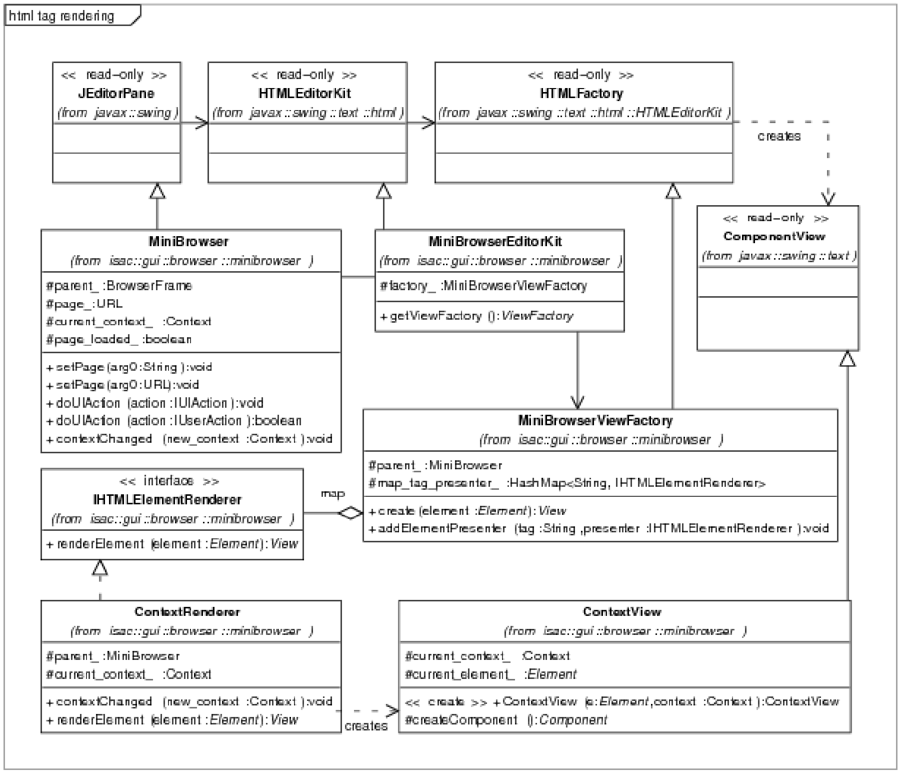

html tag rendering
<< read–only >>
JEditorPane
(from javax::swing)
<< read–only >>
HTMLEditorKit
(from javax::swing::text::html)
<< read–only >>
HTMLFactory
(from javax::swing::text::html::HTMLEditorKit)
creates
<< read–only >>
ComponentView
(from javax::swing::text)
MiniBrowser
(from isac::gui::browser::minibrowser)
#parent_ :BrowserFrame
#page_ :URL
#current_context_ :Context
#page_loaded_ :boolean
+setPage (arg0:String):void
+setPage (arg0:URL):void
+doUIAction (action:IUIAction):void
+doUIAction (action:IUserAction):boolean
+contextChanged (new_context :Context):void
MiniBrowserEditorKit
(from isac::gui::browser::minibrowser)
#factory_ :MiniBrowserViewFactory
+getViewFactory ():ViewFactory
MiniBrowserViewFactory
(from isac::gui::browser::minibrowser)
#parent_ :MiniBrowser
#map_tag_presenter_ :HashMap<String, IHTMLElementRenderer>
+create (element :Element):View
+addElementPresenter (tag :String ,presenter :IHTMLElementRenderer):void
<< interface >>
IHTMLElementRenderer
(from isac::gui::browser::minibrowser)
+renderElement (element :Element):View
map
ContextRenderer
(from isac::gui::browser::minibrowser)
#parent_ :MiniBrowser
#current_context_ :Context
+contextChanged (new_context :Context):void
+renderElement (element :Element):View
creates
ContextView
(from isac::gui::browser::minibrowser)
#current_context_ :Context
#current_element_ :Element
<< create >> +ContextView (e:Element,context :Context):ContextView
#createComponent ():Component

Figure 26.5: The Use of an EditorKit to Manipulate the Representation

Chapter 27

Bridge Java – SML

27.1 Klassen und Methoden

Die Hauptschnittstelle zum Frontend wird durch die Klassen *MathEngine*, mit der eine Verbindung zur *Bridge* hergestellt werden kann und neue *CalcTrees* angefordert werden können. Die Klasse *CalcTree* repräsentiert eine Berechung, in die neue Formeln/Taktiken eingefügt, geändert oder automatisch vom Kernel vervollständigt werden können. Um die vorhandenen Elemente im *Calctree* der Reihe nach abzuarbeiten, ist eine Klasse *CalcIterator* vorgesehen, die außerdem dazu dient, einzelne Elemente zu referenzieren, die z.B. geändert werden sollen.

27.1.1 BridgeMain

Dies ist die Hauptklasse der Bridge, die die anderen Teilkomponenten (SML-Thread, RMI-Verbindung, Logger, ...) startet. Sie erzeugt ein GUI-Fenster, in dem die Kommunikation mit dem Kernel, Verbindungen von außen, sowie Fehlermeldungen mitgeloggt werden. Diese Logging-Information wird zusätzlich in eine Datei geschrieben, wofür die Klasse BridgeLogger zuständig ist. Diese Klasse benötigt folgende Parameter beim Aufruf:

- Den Pfad, wo sich die INI-Datei befindet, in der weitere Parameter der Bridge spezifiziert werden können.
- Hostname sowie
- Port, an dem der Socket, der auf Verbindungen wartet, geöffnet wird.
- Pfad zur DTD-Datei, mit der die XML-Ausgaben des Kernel validiert werden.

Die INI-Datei hat folgendes Aussehen:

transportMode=[string—mathml] Zur Zeit werden mathematische Formeln zur Vereinfachung nur als Strings bermittelt, in einer spteren Projektphase solle jedoch die Verwendung von MathML mglich sein.

socketPort=xx[Integer] Der Port an dem ein Socket geffnet wird, der auf Verbindungen von Objekten wartet, die mit dem SML-Kernel kommunizieren mchten.

path=xx[String] Der Pfad zu dem Verzeichnis, in dem Ausgaben der Bridge (Logging und Debug-Information) geschrieben werden sollen.

waitMillis=xx[Integer] Dieser Wert gibt an, wie lange (in Millisekunden) auf eine Reaktion des Kernels gewartet wird. Verwendet wird diese Variable in der Klasse TimeCheckerThread, die fr das Erkennen einer Zeitberschreitung zustndig ist. Sollte der Kernel in dieser Zeit nicht reagieren, kann angenommen werden, dass er in eine Endlosschleife gelangt ist oder fehlerhafter SML-Code ausgefhrt wurde. Sobald dieser Zustand erkannt wurde, ist es die Aufgabe der Bridge, dafr zu sorgen, dass der Kernel wieder reagiert (im konkreten Fall wurde dies durch einen Kernel-Neustart realisiert) und den letzten konsistenten Zustand des Kernels wiederherzustellen.

kernelExec=xx[String] Der (Kommandozeilen-)Befehl, der den Kernel startet. Mit diesem externen Prozess kommuniziert die Bridge im folgenden durch Umleitung der Ein- und Ausgabestrme in Java-Objekte. In der aktuellen Phase des Projektes die SML-Variante Poly/ML zusammen mit Isabelle 2002 als Mathematikbasis verwendet.

Der SML-Prozess kann auch auf einem exteren Server aufgerufen werden, wie das momentan im ISAC-Projekt der Fall ist. Die Remote-Kommunikation erfolgt dann ber das *Secure Shell* Kommando *ssh*. Beispielsweise heit der Server, auf dem SML am IST installiert ist, *damson*, der SML-Prozess kann daher von einem beliebigen Rechner am IST folgendermaen aufgerufen werden:

```
ssh damson exec /usr/local/Isabelle2002/bin/isabelle HOL-Real-Isac
```

kernelArgs=xx[String] Optionale Parameter, die beim Aufruf des obigen Befehls mitgegeben werden knnen.

Methoden

27.1.2 BridgeRMI

Diese Klasse ist fr die Kommunikation mit dem Frontend ber RMI zustndig. Alle Objekte am Frontend, die mit dem Kernel kommunizieren, delegieren ihre Methodenaufrufe, in denen eine Anfrage an den Kernel erfolgt, an das Objekt MathEngine, das sich noch in einer gemeinsamen Java-Umgebung mit diesen Objekten befindet. Die Remote-Kommunikation erfolgt also nur zwischen den Instanzen von MathEngine und BridgeRMI.

Methoden

27.1.3 MathEngine

Dies ist die zentrale Klasse des für das Frontend sichtbaren Teils der Bridge. Da für diese Klasse das Singleton-Pattern verwendet wurde, gibt es nur eine Objekt-Instanz dieser Klasse. Die Verbindung über RMI zu den hinteren Teilen der Bridge wird von dieser Klasse im Konstruktor vorgenommen. Wenn ein Benuzter eine Rechnung startet, werden Methoden in dieser Klasse aufgerufen, diese Aufrufe werden dann über RMI an eine Instanz der Klasse BridgeRMI weitergeleitet. Dort wird der entsprechende SML-Code an den Kernel geschickt. Die Ergebnisse werden in Java-Objekte geparst und als Rückgabewert der anfangs aufgerufenen Methoden an das Frontend zurückgeschickt. Die Methoden der Klassen CalcTree, CalcHead und CalcIterator, die eine Interaktion mit dem Kernel benötigen, werden an diese Klasse delegiert, die die entsprechenden Methoden bei BridgeRMI aufruft. In dieser Klasse ist außerdem das Laden und Speichern von Berechnungen realisiert.

Methoden

27.1.4 BridgeLogger

Diese Klasse dient dazu, die Kommunikation der Bridge mit dem SML-Kernel mitzuloggen und in einer Datei auf der Festplatte für spätere Analyse-Zwecke aufzuzeichnen. Neben den gesammten Ein- und Ausgaben vom und zum Kernel werden auch Initialisierungsschritte und Statusänderugen in der Bridge, Verbindungsaufbauten und Trennungen von außen sowie Debug- und Fehlerinformationen geloggt.

Datenfelder

int minLogLevel: Schranke, die die niedrigeste Wichtigkeitsstufe von Nachrichen angibt, die noch in der Datei mitgeloggt werden sollen. Der Wert dieser Schranke wird mit 0 initialisiert, was bedeutet, dass alle Nachrichten beachtet werden. Der Wert kann von außen verändert werden, um die Anzahl der mitgeloggten Nachrichten einzudämmen.

Methoden

`public void log(int level, String msg)`
Die Nachricht msg der Wichtigkeitsstufe level (je höher die Zahl desto wichtiger) wird in der Log-Datei angehängt, wenn level größer gleich der Schranke minLogLevel ist.

27.1.5 ClientList

Hier wird eine Verbindung zwischen der ClientID eines Objektes, das eine Verbindung zur Bridge herstellt hat, und einem Objekt der Klasse PrintWriter, durch das Daten an das Objekt geschickt werden können. Hierzu wird ein Container mit dem Interface Map verwendet, in dem ein Schlüssel (die ClientID) und ein Wert (PrintWriter-Objekt) aufeinander abgebildet werden.

Methoden

`public void addClient(int id, PrintWriter pw)` Fügt einen neuen Client zur Liste hinzu.
`public PrintWriter getPrintWriterOfClient(int id)` Holt den PrintWriter, der mit der mitgegebenen ID des Clients korrespondiert.

27.1.6 SMLThread

Das Starten des externen SML-Prozesses und die Umleitung der Ein- und Ausgabeströme erfolgt in dieser Klasse, die in einem eigenständigen Thread läuft. SML wird mit dem Kommando, das als Parameter beim Starten der Bridge mitgegeben wurde, aufgerufen. Zur Kommunikation mit dienen Objekte vom Typ InputStream bzw. OutputStream, die mit dem SML-Prozess verbunden sind. Diese Objekte werden von denjenigen Bridgekomponenten, die sich mit der Kernel-Ein- und Ausgabe befassen, verwendet.

Methoden

27.1.7 TimeCheckerThread

Es wurde schon öfters in dieser Arbeit die Problematik angesprochen, dass der SML-Kernel unter Umständen nicht mehr reagiert und neu gestartet werden muss. Um einen solchen Zustand erkennen zu können, wird nach jeder Eingabe an den Kernel geprüft, ob innerhalb einer bestimmten Zeitspanne auf die Anfrage reagiert wurde, d.h. eine entsprechende Ausgabe vom Kernel generiert wurde. Hierzu wird die Klasse TimeCheckerThread verwendet. Eine Instanz dieser Klasse wird beim Starten der Bridge angelegt und mit der Variable waitMillis aus der INI-Datei initialisiert. Um ein Überschreiten des Zeitlimits festzustellen, wird jedes Mal, wenn eine Anfrage an den Kernel geschickt wird, ein Zeitstempel in einer LIFO-Queue, die sich in der Klasse BridgeMain befindet, eingetragen, der wieder entfernt wird, sobald die Antwort des Kernels erfolgt. Sollte jedoch in der definierten Zeitspanne keine Rückmeldung eintreffen, dann leitet diese Klasse die notwendigen Schritte zum Kernelneustart ein.

Methoden

27.1.8 CalcTree

Diese Klasse repräsentiert einen Berechnungsbaum im SML-Kernel.

Methoden

`public void addClient(int id, PrintWriter pw)` Fügt einen neuen Client zur Liste hinzu.

27.1.9 CalcIterator

27.1.10 ClientOutputWorker

27.1.11 Clients2KernelServer

27.1.12 Kernel2ClientsServer

27.2 XML-Parser Digester

Als Parser für die vom Kernel ausgegebenen XML-Daten wurde der in Apache Projekt beheimatete Parser *Digester* gewählt. Dieser Abschnitt befasst sich mit der Verwendung dieser Klasse im Java-Code.

27.2.1 Arbeitsweise des Parsers

Der Digester Parser ist in der Lage, aus XML-Code fertige Java-Objekte zu erzeugen, die während des Parse-Vorgangs dynamisch angelegt und mit Inhalten aus dem XML-Code befüllt werden können, wobei die Information, was an welcher Stelle in der Hierarchie des XML-Codes zu geschehen hat, dem Parser über sogenannte *Regeln* mitgegeben wird, z.B. eine Regel zum Erzeugen eines Objektes einer bestimmten Klasse. Dieses Objekt wird vom Parser auf einen internen Stack abgelegt.

Um eine Methode dieses Objektes aufzurufen gibt es eine weitere Regel. Die Methode wird mittels des *Reflection*-Mechanismus von obersten Objekt des Parser-Stacks aufgerufen. Als Parameter der Methode wird der Inhalt des entsprechenden XML-Elements mitgegeben.

Weiters gibt es eine Regel, die es ermöglicht, das oberste Stack-Objekt in das darunter liegende einzufügen, und so auf beliebig vielen Ebenen verschachtelte Objektbäume zu erzeugen. Als Ergebnis des Parse-Vorgangs des Digester Parsers wird das am Ende zuoberst am Stack liegende Objekt zurückgeliefert.

27.2.2 Einrichten und Initialisieren

Die folgenden Packages des Apache-Jakarta-Projektes (zu finden auf der Jakarta-Homepage[1]) werden für das Funktionieren des Digester Parsers benötigt: *commons beanutils; commons logging; commons digester; commons collections.* Nachdem diese Packages lokal vorhanden und dem Klassenpfad des Java-Projekts hinzugefügt wurden kann der Digester im Java-Code verwendet werden. Die erforderliche

---

[1] `http://jakarta.apache.org/commons/digester/`

Import-Anweisung lautet:
`import org.apache.commons.digester.Digester;`
Danach kann mit dem Befehl `Digester digester = new Digester();` eine neue Instanz der Klasse Digester angelegt werden.

27.2.3 Regeln

Der Digester bentigt Regeln, die angeben, was beim Parsen zu geschehen hat. Diese Regeln werden dem Parser in der Initialisierung ber bestimmte Methoden (eine fr jede Regelart) bergeben. Diese Methoden besitzen das Namensmuster `add`*`Regelart`*. Folgende Regeln wurden bei der Implementierungsarbeit bentigt:

addObjectCreate(String elem, Class cl) veranlasst den Parser, jedesmal ein neues Java-Objekt der Klasse *cl* anzulegen, sobald im XML-Code das Element mit dem Namen *elem* gefunden wird. Dieses neue Objekt wird auf der obersten Position des Objekt-Stacks des Parsers abgelegt.
Beispiel:

```
digester.addObjectCreate("*/CALCHEAD", CalcHead.class);
```

addCallMethod(String elem, String meth, int args) veranlasst den Parser, die Methode *meth* beim obersten Objekt des Stacks aufzurufen, sobald im XML-Code das Element mit dem Namen *elem* gefunden wird. Als Argument der Methode *meth* wird der Inhalt des Elements als String mitgegeben. Der Parameter *args* gibt die Anzahl der zustzlichen Argument von *meth* an (Bei den fr die Verwendung mit der Brigde-Komponente bentigten Regeln hat dieser Parameter immer den Wert 0).
Beispiel:

```
digester.addCallMethod("*/CALCID", "setID", 0);
```

Der Methodenaufruf geschieht ber den Reflection-Mechanismus, d.h.der Name der aufzurufenden Methode wird als String bergeben. Dieser Umstand fhrt dazu, dass Fehler in den Regeln oft erst zur Laufzeit entdeckt werden, z.B.wenn der Methodenname falsch geschrieben wurde, oder am Stack ein Objekt liegt, auf das diese Methode nicht angewandt werden kann.

addSetProperties(String elem, String attr, String prop) veranlasst den Parser, ein Property-Feld mit dem Name *prop* des obersten Objekt des Stacks mit Wert des Attribut *attr* des XML-Elements *elem* zu befllen. Anders ausgedrckt, wird die Methode `setProp(String val)` des obersten Objekts aufgerufen mit dem Wert von *attr* als Parameter.
Beispiel:

```
digester.addSetProperties("*/SIMPLETACTIC", "name", "name");
```

Wenn z.B.XML-Code das Element `<SIMPLETACTIC name="Add_Given">` vorkommt, so wird die Methode `setName("Add_Given")` des obersten Objektes aufgerufen.

addSetNext(String elem, String meth) die Methode *meth* beim *zweitobersten* Objekt des Stacks aufzurufen, sobald im XML-Code das Element mit dem Namen *elem* gefunden wird. Als Parameter wird das oberste Objekt des Stacks mitgegeben, welches daraufhin vom Stack entfernt wird.
Beispiel:

```
digester.addSetNext("*/POSITION", "setPosition");
```

addSetRoot(String elem, String meth) die Methode *meth* beim *untersten* Objekt des Stacks (Wurzelobjekt) aufzurufen, sobald im XML-Code das Element mit dem Namen *elem* gefunden wird. Als Parameter wird das oberste Objekt des Stacks mitgegeben, welches daraufhin vom Stack entfernt wird.
Beispiel:

```
digester.addSetRoot("*/CALCID", "setCalcID");
```

27.3 Starten der Bridge-Komponente

Chapter 28

Implementation Details

28.1 Packages

ISAC's software design document [iT02a] defines a subset of the overall functionality (described in section **??**) that has to be implemented within the prototype in order to limit the programming effort as the complete functionality for the user interface goes far beyond the confines of a thesis.

The implementation of the predetermined functionality follows the non-functional requirements described in section **??**; therefore, the Java programming platform and the Java programming language were chosen for the implementation tasks. The Java programming platform was also chosen because there are a lot of (open source) tools like *Ant*[1] or *Log4J*[2] that make it possible to concentrate on the programming tasks and are easy to integrate into the Java programming platform.

To provide a well-structured and easily readable source code, the implementation follows the *Code Conventions for the Java Programming Language*[3] that are published by SUN.

Related Java classes are grouped into Java packages to provide a hierarchy with logically connected classes. The Java classes that were implemented within this thesis or are connected to this thesis are grouped into the following packages:

- `isac.gui`
  Classes that represent the graphical user interface (WindowApplication) and *windows* that are displayed in the graphical user interface. Also classes that handle the user events and the alignment of the windows.

- `isac.gui.browser`
  All classes that are needed by the HierarchyBrowser as well as the HierarchyBrowser itself.

---

[1] http://ant.apache.org/

[2] http://logging.apache.org/log4j/docs/index.html

[3] ftp://ftp.javasoft.com/docs/codeconv/CodeConventions.pdf and also available on the CD accompanying this thesis.

- `isac.gui.browser.example`
  Classes that are connected to the Example HierarchyBrowser as well as the Example HierarchyBrowser itself.

- `isac.gui.browser.method`
  Classes that are connected to the Method HierarchyBrowser as well as the Method HierarchyBrowser itself.

- `isac.gui.browser.problem`
  Classes that are connected to the Problem HierarchyBrowser as well as the Problem HierarchyBrowser itself.

- `isac.gui.calcheadviews`
  Classes that represent the different views and therefore the detail level that a CalcHead can have.

- `isac.gui.treetable`
  Classes that represent the calculation tree as well as classes that can modify the representation of the calculation tree.

- `isac.gui.util`
  Helper (Utility) classes for the GUI-like classes that perform GUI-related work in a dedicated thread or classes that are responsible for the layout of graphical components in a window.

- `isac.util`
  Helper (Utility) classes that provide functionality which can be helpful within the whole $\mathcal{ISAC}$ architecture.

- `isac.session`
  Classes that are responsible for the session handling like the SessionDialog.

- `isac.wsdialog`
  The DialogGuide (Worksheet Dialog) and classes that have a relation to the DialogGuide. These are classes that for example record the user history or build up the connection to the Bridge.

Part V

Usecases

This part of the document serves the following purposes: it

1. relates the isac-docu to the code and vice versa: i.e. the use cases to the test cases (JUnit tests, etc.). For that reason the code uses exactly the complete labels within this document, e.g. “`\label{UC:cas-input}`”.
2. documents internal discussions about crucial design questions during the early design phases
3. establishes data for test cases at the end of the implementation phase
4. provides for entry-points to understand *ISAC* for collegues newly joining the *ISAC*-team.

Thus for the most important usecases there are more details (including interaction diagrams) and testdata in the software design document (SDD), part IV: each testcase is based on a usecase described here.

All these details in the SDD are referenced at the respective usecases below and presented in the subsequent part VI.

Chapter 29

Visit an *ISAC* site

The 'user' in this chapter is a visitor.

UC 29.0.0.1 ***The contents of an ISAC site is inspected using a standard browser***
Anybody dropping into an *ISAC*-site more or less intentionally first wants to know what is offered at this site. The visitor may never had heard anything about *ISAC*, and thus should get information on the purpose and the features *ISAC*. Or somebody may want to know in detail about the contents of the knowledge base and/or the example collection of this particular site.

1. The Visitor is reaching the entry-page of the *ISAC*-site. This site containes a welcome-message from the host of the system as well as a brief sumary of its content. The entry-page acts as doorway for
    - the *ISAC*-knowledge (i.e. decorated knowledge including explanations) as it contains links to the problems, methods and theories
    - the example collection
2. The user might select one of the three main branches of the knowledge base (Fig.29.1. on p.186) and browse them as described in 29.1, or jump directly to an example-collection.

29.1 Browse the knowledge:

*ISAC*s mathematics knowledge is separated from the knowledge interpreter (the 'mathematics engine'). The knowledge is described such that it can be read by human users *and* can be interpreted by the mathematics engine in *one and the same* format. Thus the knowledge base is one entry point of interactive learning (besides the example collection).

For technical reasons we subsume the example collection here: Handling and representation of knowledge and examples has been unified so far. There are

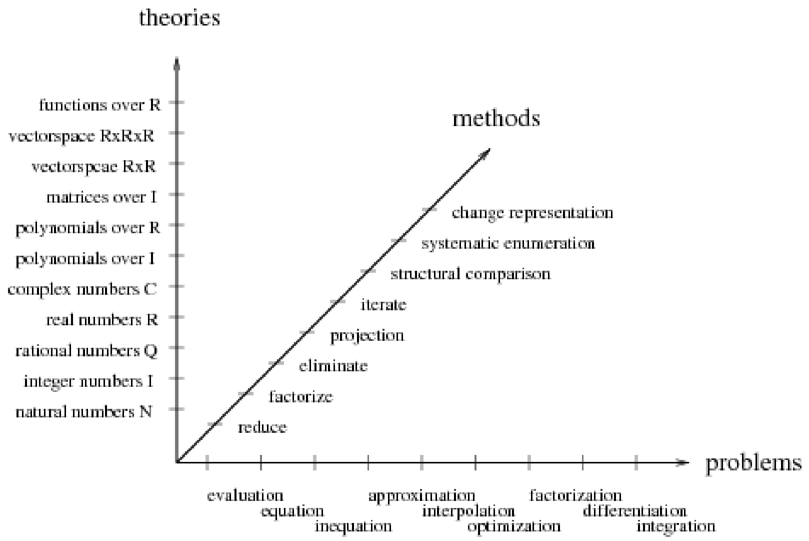


Figure 29.1: The 3 dimensions of ISACs knowledge base

several terms concerning knowledge, see [iT02b] 'List of Terms Used in the ISAC Project': knowledge, decorated knowledge (i.e. knowledge plus explanations), KE-store (i.e. decorated knowledge plus example collection).

UC 29.1.0.2 *User browses through the knowledge hierarchies*

A student uses a WEB-Browser to gain an overview of ISACs problem hierarchy, for instance, see p.186.

1. The user decides for 'problems' (and not for theories, methods or examples; this descision may be put on another KE-store browser or on a worksheet).
2. The browser-window of the problem-browser shows the root problem
3. The browser-window of the problem-browser shows the hierarchy of problems (at least the next lower level of problems).
4. After selecting one of the problems this one is displayed, and all the explanations contained are shown. (These explanations are optional informations which have been added by a course designer, depending on the special course: typical examples, illustrations etc).
5. Eventually the navigation tool for the hierarchy is updated (nodes can be expanded and collapsed when needed)

Further navigational help: If the user has changed the page with the contents of a problem, the survey on the hierarchy might not be up to date. Thus there is button 'where am I ?' showing the position of the current page within the hierarchy.

29.2 Browse the example collection

ISAC is a web-based system as the mathematics knowledge (Isabelle theories, problems, methods) and the example-collection of an *ISAC*-site can be browsed using a standard web-browser — if the owner of the *ISAC*-site is willing to do so (and there are many reasons to make the offers of a specific site visible to the public, at least a part of it).

The math power of an *ISAC*-site actually can be experienced by doing the calculations given in the respective example-collection. However, doing the calculation in the interactive way (which is *ISAC*'s outstanding feature !) requires software which is too complext to be handled by the Applet-technology available presently; for interactive calculation you need to download the *ISAC* Tutoring System.

But one may want to get at least a glimpse of the math power of an *ISAC*-site, or one might be interested only in the completed calculation, not in the interactive construction of a calculation: this for is the *ISAC* Web Reader — see `http://www.ist.tugraz.at/projects/isac/www/content/isac-reader.html`.

UC 29.2.0.3 ***Select and display a single example***
The user selects an example from the example-hierarchy (i.e. a leaf of the tree), and the description (text, figure, formulae) of the example is displayed.

UC 29.2.0.4 ***Execute a single example***
Executes an example displayed in the example browser by hitting some hotspot (preferably the number uniquely identifying each example).

UC 29.2.0.5 ***Select and display a page of examples***
Select a page of examples like in a textbook.
Hint (to be deleted from here!): Each node of the example hierarchy has a ¡DESCRIPTION¿ which can be used for a page of examples. Design such that a page is a node exactly one level above the leaves, and the leaves are exactly the examples contained in the respective page. The difference and challenge is with the following ...

UC 29.2.0.6 ***Execute an example from a page***
Execute an example (preferably by clicking the number identifying the example) contained in a page displayed on the example browser.

.

The last usecase does not address the visitor, but an example author. It is kept here until other UCs for the example author come up.

UC 29.2.0.7 ***An author selects a single example***
This extends UC.29.2.0.7 by displaying the formalizations of the example, which is hidden from the learner.

Chapter 30

Learn interactively with *ISAC*

The 'user' in this chapter is a learner.

Some of the actions below may be blocked depending on the specific example being solved, the user's preferences or restrictions in the user's rights. In this case, nothing happens, as if the action had not been requested.

The specific way how actions are requested is not specified. At the discretion of the designers of the GUI, this could happen by selecting from a menu bar or a context menu, by clicking buttons or by keyboard shortcuts.

30.1 Start a Calculation

30.1.1 Initialising Session and Dialog

UC 30.1.1.1 ***Identifying the User***
The user has to identify himself so *ISAC* can retrieve his personal preferences, his performance history and restrictions in his access rights.

On success: The user is identified and continues by choosing a starting point as described in 30.1.2.

On failure: The identification process starts over again.

Further Information: This use case shows no alternatives for incorrect user input. The user can cancel the login process at any time.

1. The system asks the user to authorize by username and password.
2. The user inputs username and password.
3. The system checks the inputs with the stored username and password pair and TODO shows the group(s) the user is member of.

4. TODO: If the user is assigned more than 1 course, the system asks for the choice of a course, and grants access.
5. Use case ends successfully.

Details in **??**.

UC 30.1.1.2 ***A subscribed user calls ISAC.***
The user has logged in as a registered user in another system (preferably a content management system of an educational institution) already; and within this system he followed a link to *ISAC*. In this case another login would be boring; thus *ISAC* tries to get the login-data immediately from the calling system. These login-data are the same data as in **UC.**30.1.1.1.

UC 30.1.1.3 ***Contacting a Math Engine***
The Math Engine does not run on the same machine as the Dialog Guide. The network address of the Math Engine has to be entered manually or read from a configuration file. This could happen before or after the user login.

On success: Continue with user identification (UC.30.1.1.1) or choosing a starting point (30.1.2).

On failure: Failure is critical, calculations cannot be done. The program has to be aborted. As an alternative, the user can be prompted for a network address even if he would not enter it manually under normal circumstances.

30.1.2 Choosing a Starting Point

UC 30.1.2.1 ***Doing an Example from an Example Collection***
An example from a collection of prepared examples is started for stepwise interactive calculation. This is analoguous to UC.29.2.0.4.

On success: The complete Model and Specification of the example are known to *ISAC*, so *ISAC* can offer help filling in the fields of Model and Specification.

Precondition: The complete model and specification of the example are known to *ISAC* (but not shown to the user), so *ISAC* can offer help filling in the fields of model and specification, which in turn is the prerequisite for *ISAC* to be able to calculate automatically the solution of the example.

Moreover, from the potential of solving *ISAC* derives the potential to automatically derive user guidance.

1. The system provides the user the example hierarchy.
2. The user selects an example from the hierarchy.

3. The system generates a CalcHead from the formalization hidden behind the example.
4. The system creates a CalcTree with the CalcHead.
5. The use case ends successfully.

Details in 33.1.2.

UC 30.1.2.2 *Starting a Calculation from Scratch*

The user can choose to start an example from scratch, entering the Model and Specification manually into an initially empty form.

On success: The user is free to do calculations not contained in the example collection. On the other hand, in this case *ISAC* cannot offer any help in completing the Model and Specification, only checks for completeness and consistency. See also UC.30.1.2.1 and UC.30.2.1.1.

Precondition: The user is free to do calculations not contained in the example collection. Help while filling in the fields of model and specification can only be based on matching from data entered so far. On the other hand, in this case *ISAC* cannot offer any help in completing the model and specification, but only checks for completeness and consistency.

1. The system provides the user with an empty model and specification.
2. The user *models* and *specifies* the problem (as described by use case **??**).
3. The system creates a calculation from the specification.
4. The use case ends successfully.

Solve an example for a particular problem:

Let us assume the user browses through the hierarchy of types of equations, finds some interesting type (i.e. 'problem'), eventually because of the method mentioned in this problem, and wants to solve an example of this type of equation by use of the method proposed.

1. The user has the equation-type ('problem') ["degree_5", "polynomial", "univariate", "equation"] in the problem-browser, and finds ["Newton", "univariate", "real"] as method solving this equation-type.
2. He hits the button ⟨transfer to worksheet⟩ which pops up a worksheet with an empty model, but with descriptions already inserted from the problem ["degree_5", "polynomial", "univariate", "equation"] (analoguous to UC.**??** below). This problem is also inserted in the calc-heads specification, already.

3. The user inputs an equation $x^2 + 3x + 4 = 0$ with bound variable x into 'given: *equality*' and 'given: *boundVariable*' respectively — yielding the status 'incorrect' for the precondition $has\_degree\_in\ 5\ (x^2 + 3x + 4 = 0)\ x$
4. If he has input an equation with a correct degree, and if he has selected the method (ev. from the method-hierarchy) and thus completed the calc-head, he can start solving.

Such situations are captured by the following use cases.

UC 30.1.2.3 ***Starting a Calculation from a Problem***
The user can choose to start an example beginning from a Problem, entering the Model and Specification manually. After choosing a Problem by browsing the Problem hierarchy, modelling and specifying will start with a form which is initially empty except for the Problem chosen.

In the future, *ISAC* could offer extra help by offering a choice of Methods known to match the Problem or by help texts specific to the Problem.

On success: The user is free to do calculations not contained in the example collection. On the other hand, in this case *ISAC* cannot offer any help in completing the Model and Specification, only checks for completeness and consistency. See also 30.1.2.1 and 30.2.1.1.

UC 30.1.2.4 ***Solve an example demonstrating a particular method***
Let us assume the user browses through the hierarchy of methods for solving equations, finds some interesting method (e.g. ["Newton", "univariate", "real"]), and wants to solve an example by use of the method selected.

1. The user has the method ["Newton", "univariate", "real"] displayed in the knowledge browser.
2. He hits the button ⟨transfer to worksheet⟩ which pops up a worksheet with an empty model, but with descriptions already inserted from the method ["Newton", "univariate", "real"]. This method is also inserted in the calc-heads specification, already.
3. The following steps and features are analoguous to UC.30.1.2 ...

UC 30.1.2.5 ***Input an example like into an algebra system***
The user calls such typical function like 'solve', 'simplify', 'differentiate' etc, presumerably for a quick solution.

1. The user gets a worksheet for input of just the *one* (head-)line (and no other parts of a (still empty) calc-head)
2. The user inputs $solve\ (x^3 + x^2 + x + 1 = 0,\ x)$
3. The dialog will guide the user immediately into the solve-phase (as above in UC.30.1.2.5 — or the user hits a button which immediately calculates the result.

30.2 Model and Specify an Example

As modeling and specifying is done ahead calculating a solution, and as these activities are rather different from calculating, these activities are done within an environment called CalcHead.

30.2.1 Edit the CalcHead

Let us assume, the user wants to exercise the example for demonstration in appendix A on p.219. This example is particularily interesting w.r.t. modeling (and not so much w.r.t. specifying). Furthermore, let us assume the CalcHead-Panel is already open and displays the empty model; i.e. without the contents of the boxes as shown on p.219, but containing the 'decriptions' (*Constants, Maximum,* etc.) for user-guidance.

1. The user inputs $r = 7$ in the appropriate field 'given: *Constants*' yielding the status 'correct' for this item.
2. The user inputs u to 'find: *Maximum*' and $A = 2uv - u^2$ to 'relate: *Relations*' yielding the states 'superfluous' (i.e. not known to the system) and 'incomplete' respectively.
3. If the user does not understand 'superfluous' (i.e. the system cannot relate the item to the example — the term 'superfluous' could be better ?!) he may ask the system for proposing the next step.
4. The system suggests 'Add-Find *Maximum A*'; accepting this suggestion yields 'correct'.
5. The user inputs $\frac{u}{2} = r\sin\alpha$ in 'relate', still yielding 'incomplete'. If the user may ask once more, the system suggests $\frac{v}{2} = r\cos\alpha\}$ pursuing the variants F_{II} and F_{III} on p.219.
6. However, the user finds out that he can solve the problem using the theorem of Pythagros, adds $(\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2$ to '*Maximum A*' and yields a 'complete' model. (i.e. after the tacit specification of theory, problem and method the calc-head is 'complete').

This situation is captured by the following use case.

UC 30.2.1.1 ***Editing the Model***
The user inpts items (i.e. adds formulae to the descriptions) to the fileds 'given', 'find' and 'relate' and gets feedback fron $\mathcal{ISAC}$ on each item with respect to the problem from the hidden specification. No fields of the Model or Specification will be changed.

On success: Fields offending criteria for completeness and correctness are marked and offer feedback on their status. The user can continue modelling or specifying. $\mathcal{ISAC}$ does not correct incorrect items; however, the user can remove the items marked as incorrect and let *ISAC* provide the correction as described in other use cases in this section.

On failure:

Precondition: The model and the specification are known to the system (because the user has selected the example from the collection, containing these data) but not shown to the user. The correct completion of the model and the specification is the prerequisite for *ISAC* to be able to calculate the solution of the example.

1. The user has to model the problem — input values for given, find, with, relate — and to specify the problem — input values for theory, problem and method.
2. The system checks the model with respect to the specification, in particular it matches the (root-) model with the problem.
3. The user accepts the calc-head (model and specification).
4. The use case ends successfully.

30.2.2 Obtaining Help from *ISAC*

Besides obtaining feedback on each item input into the model, the user can get help from *ISAC* in the following ways.

UC 30.2.2.1 ***Having ISAC Complete the CalcHead***
The user requests a CalcHead, containing some input, to be completed automatically.

On success: The Specification is complete and correct and ready for starting a calulation as in 30.2.4.

On failure: If the CalcHead contains some wrong items, a 'completion' cannot result in a correct CalcHead. The user can still continue specifying manually.

UC 30.2.2.2 ***Having ISAC Complete the Model***
When doing a prepared example, the user can have the Model (fields Given, Find, Relate) completed by *ISAC* with data from the hidden formalization.

On success: The Model is complete and correct.

On failure: If the example was started from scratch (UC.30.1.2.2), the data entered so far might not suffice to fill in all fields or find the best match. The user can still continue modelling manually.

UC 30.2.2.3 ***Having ISAC Complete the Specification***
When doing a prepared example, the user can have the Specification completed by *ISAC* with data from the hidden specification.

On success: The Specification is complete and correct and ready for starting a calulation as in 30.2.4.

On failure: If the example was started from scratch (UC.30.1.2.2), the data entered so far might not suffice to fill in all fields or find the best match. The user can still continue specifying manually.

UC 30.2.2.4 ***Having ISAC Complete One Field of the Model***
When doing a prepared example, the user can have one field of the Model completed by ISAC with data from the example collection.

On success: The selected field of the Model is complete and correct.

On failure: If the example was started from scratch (30.1.2.2), the data entered so far might not suffice to fill in the field or find the best match. The user can still continue modelling manually.

UC 30.2.2.5 ***Having ISAC Complete One Field of the Specification***
The user can have one field of the Specification (fields Theory, Problem, Method) completed by ISAC from data entered so far.

On success: The selected field of the Specification is complete and correct.

On failure: If the example was started from scratch (30.1.2.2), the data entered so far might not suffice to fill in the field or find the best match. The user can still continue specifying manually.

30.2.3 Contextual Access to the Knowledge

In many examples the interactive generation of the specification (theory, problem, method) is an opportunity for learning, e.g. determining the type of equation (i.e. the problem). For this task the user has to search the mathematics Knowledge Base, while ISAC uses contextual information from the calculation for guiding the search.

Three cases:

1. The user wants to calculate an existing example. In this situation the user has to choose the problem from a browser-window for problems (UC 29.1.0.2). But the user goes with the current model and specification on the CalcHeadPanel into the problem hierarchy. In order to give the user a hint for the problem the problem-browser is opened with the problem as specified (e.g. *problem [univariate, equation]* as provided by the generator of the example and suggested by ISAC on previous request of the user) — UC.30.2.3.1 below.
   Once the problem-browser is opened, any selection of a problem matches with the model still marked (i.e. the system remembers the active model. Thus, in order to browse *without* matching, there is a toggle-button $\langle match/no - match \rangle$ on the browser !) — UC.**??** below.

After some browsing the user may get lost in the large problem hierarchy. If he remembers the starting point of the search, he may select *problem [univariate, equation]* and hit the button ⟨*refine*⟩ on the browser (while $\langle match/no - match\rangle$ is activated) — UC.30.2.3.9.

2. The user is somewhere in the calculation and calls for <Problem>; then this problem is displayed in the problem browser which is specified in the parent-CalcHead of the Active Formula.
3. The user wants to start a new calculation that is unknown to the system (UC **??**).
   In this situation the user has also to choose the problem from a problem browser, but the system cannot give a hint to the user as the calculation is unknown to the system. Therefore, the browser window is opened with the root of the problem tree marked.

UC 30.2.3.1 ***Go into the problem hierarchy with a particular model***
The CalcHeadPanel contains some (or even no) input items in the model. These items are matched with the problem selected in the problem hierarchy.

Steps taken:

1. The user calls for the problem-browser.
2. The system provides the user the problem hierarchy.
3. The user selects a particular problem from the problem hierarchy which he wants to be matched.
4. The system validates the request and returns the matched model. Each model item (given, where, find, relate) contains a status.
5. The use case ends successfully.

For transferring the match to the actual worksheet see UC 33.2.4 on p.196.

UC 30.2.3.2 ***Go into the method hierarchy with a particular model***
Analoguous to UC 30.2.3.1.

UC 30.2.3.3 ***Go into the theory hierarchy with a particular theory***
Analoguous to UC 30.2.3.1.

UC 30.2.3.4 ***Having $\mathcal{ISAC}$ Refine the Problem.***
On request of the user, $\mathcal{ISAC}$ refines the problem to a best match of the data entered into the model so far. The root of the search is the currently selected problem.

Steps taken:

1. The system provides the user the problem hierarchy.
2. The user generates an action that tells the system to refine the selected problem.

3. The system validates the request and tries to find the corresponding problem for the model and returns the best matching problem. Each model item (given, where, find, relate) contains a status based on the matching.
4. The use case ends successfully.

For transferring the match to the actual worksheet see UC **??** on p.196. The transfer fails without error, if no worksheet is open (e.g. at the very beginning of a session).

Details in 33.2.3.

UC 30.2.3.5 ***Go into the problem hierarchy from a worksheet***
The user calculating (i.e. in the solve phase, some Active Formula) calls the problem hierarchy. This selects, matches and displays the problem specified in the parent-CalcHead to the Active Formula.

UC 30.2.3.6 ***Go into the method hierarchy from a worksheet***
analoguously.

UC 30.2.3.7 ***Go into the theory hierarchy from a worksheet***
analoguously.

UC 30.2.3.8 ***Transfer a selected problem to the worksheet***
Let us assume the user has found an appropriate problem P (with key *pblID*) in the problem-hierarchy (the matched model m' is on the problem-browser, too). Thus, the button $\langle match/no-match\rangle$ was activated to 'match'. Now she wants to transfer the specified problem P to the worksheet (and to the related calc-tree in the SML-kernel). For this purpuse the 'active formula' must have been somewhere on a modspec on pos p.

- The user hits a button $\langle transfer-to-worksheet\rangle$.
- The worksheet comes to the foreground again, and inserts model m' and updated *pblID* in the specification, all at position p (and all has been updated in the SML-kernel as well).

Details in 33.2.4.

UC 30.2.3.9 ***Toggle a problem from instantiated to non-instantiated***
The problem browser has been called from a worksheet, and shows the problem instantiated from the worksheet's context. A toggle-button switches to the plain problem without instantiation. And vice versa.

UC 30.2.3.10 ***Getting Background Information from the Knowledge Base***
Especially with Tactics, extra background information may be available from $\mathcal{ISAC}$'s Knowledge Base. In an extra window, $\mathcal{ISAC}$'s Knowledge Base can be browsed, with information on the current item as a starting point.

On success: A browser window is opened showing information on the current item.

On failure: If no information matching the current item can be found, browsing starts from the root of $\mathcal{ISAC}$'s Knowledge Base or from information on the current Problem.

30.2.4 Starting the Solving Phase

UC 30.2.4.1 ***Starting Interactive Calculation***
Having entered a Model and a Specification, the user can start solving the example interactively.

On success: In a worksheet window, the first formula in is displayed. Calculation continues as in 30.3.

On failure: If the Model and Specification are not error-free, complete and consistent, modelling and specifying continues. Status information on the fields of the Model and Specification is updated.

UC 30.2.4.2 ***Starting Automatic Calculation***
Having entered a Model and a Specification, the user can request the calculation being done by $\mathcal{ISAC}$.

On success: In a worksheet window, the final result is displayed. The user can still analyse and modify the calculation as described in 30.3.

On failure: If the Model and Specification are not error-free, complete and consistent, modelling and specifying continues. Status information on the fields of the Model and Specification is updated.

30.3 Calculating a Result

Let us assume, the calculation has been propagated for several steps until the following expression (the *current formula*) is on the worksheet (see x.x on page **??**)

$$\frac{d}{du}\left(4u\sqrt{r^2-\left(\frac{u}{2}\right)^2}-4\left(r^2-\frac{v}{2}\right)^2\right)=$$

Now, what are the choices for the users (i.e. the users) ?

Any of the following actions may be taken at any time during the calculation. The numbering below does not imply the order in which the actions are taken.

30.3.1 Moving the Active Formula

Most actions during the Solving Phase refer to a specific point in the course of calculation, the currently active formula. All editing takes place at the currently active formula or the Tactic being applied to it and calculation continues from that point. Most of the time, the last formula entered or calculated will be the active one.

UC 30.3.1.1 ***Moving the Active Formula***

In order to edit a part of the calculation, the user has to move the active formula to the desired location first.

On success: The referenced formula becomes the new active formula. Actions refer to this position now.

30.3.2 Taking Single Steps Interactively

UC 30.3.2.1 ***Entering a Tactic Manually***

The user can enter the Tactic, i.e. step of calculation, to be applied to the active formula manually. The Tactic is recognised by its name. The Tactic is not applied immediately, but will be applied the next time a step in a calculation (30.3.3) is requested.

On success: The Tactic is recorded and will be applied to the active formula the next time a step of calculation is requested.

On failure: If the Tactic entered is not known to $\mathcal{ISAC}$ or the Tactic is not applicable to the currently active formula, the user is notified but nothing else happens.

UC 30.3.2.2 ***Picking a Tactic from a List of Known Tactics***

The user can pick the Tactic to be applied to the active formula from a list of all Tactics known to $\mathcal{ISAC}$. The Tactic is not applied immediately, but will be applied the next time a step in a calculation (30.3.3) is requested.

On success: The Tactic is recorded and will be applied to the active formula the next time a step of calculation is requested.

On failure: If the Tactic is not applicable to the currently active formula, the user is notified but nothing else happens.

UC 30.3.2.3 ***Picking a Tactic from a List of Applicable Tactics***

The user can pick the Tactic to be applied to the active formula from a list of Tactics applicable to the current situation. This list is prepared by $\mathcal{ISAC}$. The Tactic is not applied immediately, but will be applied the next time a step in a calculation (30.3.3) is requested.

On success: The Tactic is recorded and will be applied to the active formula the next time a step of calculation is requested.

UC 30.3.2.4 ***Entering a Formula Manually***

The user can enter a proposal for the next formula in the course of calculation manually.

If a Tactic has been chosen for this step, the input must match the result of this Tactic.

If no Tactic has been chosen, there must be a sequence of Tactics which derive the formula entered from the currently active formula.

On success: The entered formula is entered into the calculation and becomes the currently active formula. Additional steps may be added automatically to reflect the derivation of the formula entered.

On failure: The formula is rejected if it is not the result of the Tactic chosen for this step or - if no Tactic has been chosen - no derivation confirming the correctness of the formula can be found. A new formula can be entered.

UC 30.3.2.5 *Editing and Replacing a Formula*

The user can edit and modify the active formula, which need not be the last formula in the calculation. Changing a formula potentially invalidates all subsequent steps, which are removed from the calculation. There must exist a sequence of Tactics which derive the formula entered from the preceding formula.

On success: The edited formula is entered into the calculation. The display is updated as to reflect the derivation of the entered formula.

On failure: The formula is rejected if no derivation confirming the correctness of the formula can be found. A new formula can be entered.

30.3.3 Automatic Calculation

UC 30.3.3.1 *Having ISAC Propose the Next Tactic*

In addition to choosing a Tactic manually (UC.30.3.2.1), ISAC can propose the next Tactic based on its knowledge of the Method solving the current Problem.

On success: A Tactic for the next step has been chosen.

UC 30.3.3.2 *Having ISAC Calculate the Next Formula*

In addition to entering a formula manually (UC.30.3.2.4), ISAC can calculate the next formula by applying the chosen Tactic to the active formula. If no Tactic has been chosen, ISAC uses its knowledge about the Method solving the current Problem to choose a Tactic (UC.30.3.3.1)

On success: The entered formula is entered into the calculation and becomes the currently active formula.

UC 30.3.3.3 *Having ISAC Calculate until the End of the Current Subproblem is Reached*

ISAC can calculate any steps needed to finish the current subproblem automatically.

On success: Intermediate steps have been added to the calculation to solve the current subproblem. The result of the subproblem becomes the currently active formula.

UC 30.3.3.4 ***Having ISAC Calculate until a Final Result is Reached***
ISAC can calculate any steps needed to finish the current calculation automatically.

On success: Intermediate steps have been added to the calculation to reach a final result. The result becomes the currently active formula.

30.3.4 Showing and Hiding Data

UC 30.3.4.1 ***Hiding a Category of Information***
The user can choose not to see certain categories of information (e.g. Tactics, Assumptions). In any case, formulas cannot be hidden and remain displayed.

On success: The respective category of information is removed from the worksheet.

On failure: The worksheet remains unchanged.

UC 30.3.4.2 ***Showing a Category of Information***
The user can choose to see categories of information currently hidden.

On success: The respective category of information is displayed on the worksheet.

On failure: The worksheet remains unchanged.

UC 30.3.4.3 ***Choosing the Displayed Nesting Depth of Details***
For the sake of better overview, the user can choose the depth of nested subproblems or subcalculations which are displayed. Hidden details are indicated by symbols to facilitate display on demand. If the currently active formula is below the displayed level of detail, it is moved up to the nearest visible formula.

On success: The respective nesting levels are removed from or added to the worksheet. Symbols indicate hidden levels of detail. After hiding levels, the active formula may have moved.

On failure: The worksheet remains unchanged.

UC 30.3.4.4 ***Hiding Parts of the Calculation***
For the sake of better overview, the user can choose to hide several steps of the calculation. Hidden steps are indicated by symbols to facilitate display on demand. If the currently active formula has been hidden, it is moved up to the nearest visible formula.

On success: The respective steps are removed from the worksheet. Symbols indicate hidden steps. After hiding steps, the active formula may have moved.

On failure: The worksheet remains unchanged.

UC 30.3.4.5 ***Hiding Tactics on Behalf of $\mathcal{ISAC}$***
$\mathcal{ISAC}$ can decide to hide steps of the calculation from the user. Such decisions can be based on records of the user's experience with specific Tactics or on the user's preferences.

On success: The respective steps are not displayed on the worksheet. Symbols indicate hidden steps.

UC 30.3.4.6 ***Showing Hidden Parts of the Calculation***
Parts of the calculation indicating hidden steps can be expanded, adding one nesting level of detail at a time. It does not matter whether the steps have been selected for hiding (UC.30.3.4.4) or have been hidden because of their nesting depth (UC.30.3.4.3) or have been hidden by $\mathcal{ISAC}$ (UC.30.3.4.5) in the first place.

On success: The respective steps are added to the worksheet.

On failure: The worksheet remains unchanged.

30.3.5 Obtaining Help and Extra Information

UC 30.3.5.1 ***Displaying the Assumptions Holding at a Specific Point in the Calculation***
The user can have the Assuptions (e.g. restrictions on possible values for a variable) holding at a specific point in the calculation displayed. Different from actions changing the calculation, the user need not move the active formula to the spot of interest.

On success: The Assumptions holding at the referenced formula are displayed.

UC 30.3.5.2 ***Displaying the Origin of Assumptions***
The user can have the origin of an Assumption, that is the Tactic creating the assumption, indicated on the worksheet.

On success: The Tactic where the assumption originated is indicated.

UC 30.3.5.3 ***Show the tactic applied to a formula***

UC 30.3.5.4 ***Show a list of tactics applicable to a formula***

UC 30.3.5.5 ***Show the intermediate steps leading to a formula***

UC 30.3.5.6 ***Show the CalcHead to a HeadLine on the worksheet***
The user marks a formula, which is the headline of a (sub-)problem, and requests to display the whole CalcHead.

30.3.6 Solving problems with subproblems

Let us assume, the user has finished the *model-phase* of a rootproblem or a subproblem in the reference example and has the following *model* on the screen:

x. *L = solve (Reals, [univariate, equation])*
   Model
      given: *equality* $-\frac{v}{\sqrt{r^2-\left(\frac{v}{2}\right)^2}}+2\sqrt{r^2-\left(\frac{v}{2}\right)^2}-2v=0,$
         boundVariable v*, errorBound* $\epsilon = 0.001$
      find: *solutions L*
   Specification
      Theory: *Reals*
      Problem: *[univariate, equation]*
      Method:

Now the user has to solve the equation. We further assume, *ISAC*'s knowledgebase contains a *method* for solving this particular equation; then it can be found either directly in the list of methods. If the user cannot find it directly, then the *problem hierarchy* assists in the search for an appropriate method by *refining*: the problem *[univariate,equation]* will *match* the model, apparently; and the problem hierarchy is structured such that the tactic *Refine_Problem [univariate,equation]* will find the appropriate problem (in this case: type of equation) in a search on the children of *[univariate,equation]* automatically — let us assume, the appropriate one is *[fraction,root,univariate,equation]*. Eventually the tactic *Refine_Problem* may be disallowed by the dialog guide (for the reason to push users searching the problem hierarchy themselves).

Chapter 31

Authoring

31.1 Author the knowledge base

UC 31.1.0.1 ***Build the structure of the knowledge base description***
All informations about the structure of the knowledge base are stored inside the SML-part of the system. This structure has to be known by the Broswers to enable the user to browse the systen. To reduce the load of the math-engine and to enhance the informations available for a problem, the informations of the mathengine are copied to the description-database. Here, the problem-descriptions can be enriched with additional informations using the *description editor*. The structure for the browsers is created through querying the Browsergenerators. This UC concernes about creating the structure inside the description-database and add additional informations.

① The structure-transfer is started by the description-administrator[1].

- The user starts the description-editor
- The description-editor starts the authentication-process which authentificates the user and results in an permission object which identifies the user[2].
- The *description-editor* Calls the *description-storage* to initiate the structure transfer.
- The *description-storage* starts the transfer if the permissions suffice.

authenticationmodule einfuehren - wo werden die userinfos gespeichert, kann man auch rechte auf einzelne knoten geben, wie werde diese rechte gespeichert...

② *browser-generators* are queried for the structure of the knowledge base. A XML Description of each node is created and stored within the XML-Description database.

---

[1] Access has to be controlled by the description-editor which is responsible for the user-authentication - only identifyed users may access

[2] The authentication process utilizes the Dinopolis framework - a detailed description can not be given yet

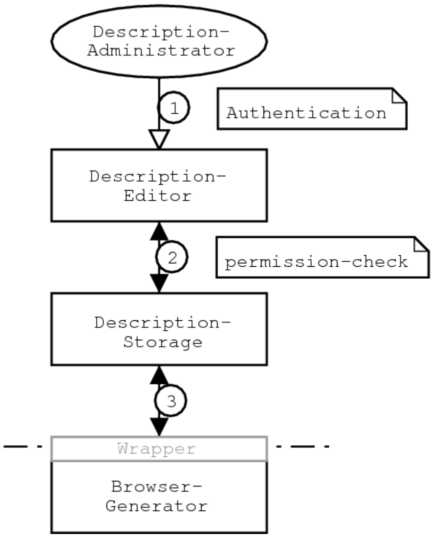


Figure 31.1: Structure generation

- the *browser-generator* is asked by the *description-storage* for the root-node of the problem hierachy[3].
- Afterwards, the *browser-generator* is asked for the mathematical informations of this problem. Mathematical informations contain:
  - subproblems
  - items of the problem
- A XML - Description is built for this informations. All informations are stored in dedicated fields which identify their type. Subproblems are stored as list of references to their XML-Description which are built resursively in the same way. A reference to the parent problem has to be added too.
- The XML-Descriptions are put into the storage. A reference to the root problem is set.

③ After creating the structure, the *description editor* can browse through the XML-Files and add additional information.

- The description editor asks the description-storage about the root problem
- The description-storage returns a so called *description-object*, which containes all necessary informations, to the editor. [4]

---

[3] This query is transferred through the wrapper which translates java-objects into a character stream which is used to communicate with the Math-Engine and vice versa - the syntax of the communication with the wrapper has to be defined within the ADD

[4] see ADD and SDD for more details

- The description containes fields which might be changed by the description editor through set-methods. Again, access to this methods is only given after checking the permission-object. A change of mathematical informations out of the math-engine is not allowed. Only additional informations like text, images and references to further informations (URLs, examples) are allowed. Changes have to be propagated to the description-storage and other users of the same informations (e.g. another description-editor or a browser).
- As the representation containes information about the subproblems, the editor also can navigate through the hierachy to edit other than the root-problem by loading their representation.

31.2 Author the example collection

UC 31.2.0.2 ***An author adds some examples***
Let us assume that some of the examples from a traditional mathematics textbook have been implemented in an ISAC example collection. Now an author wants to add some more examples from another section. Then he has to accomplish the following tasks.

1. **He opens the example browser** and zooms into the table of contents in order to locate the position where to insert the new section.
2. **He inserts the new section** by first selecting the section preceeding the new one. Then he selects the level of the new section (which may be the same as the preceeding one, or one level deeper) and inputs the headline of the new section. .
   Alternatives:
   (a) **The required headers are all present** already, and the author starts immediately with (3) editing an example. In this case the location of the examples label and text (or figures) is determined immediately.
   (b) **The new section is several levels deeper** in the hierarchy than the last preceeding section already input. In this case at least a label (and eventually a headline) for each level down to the level of the new section is input one after the other.

AG: vielleicht sollten wir das eigentliche example und den “verzeichnisbaum” (level/headline) voneinander trennen, damit man ein examle in verschiedenen beispielsammlungen verwenden kann

3. **He edits the first example** by declaring its label according to the textbook. Then he edits the examples text, the formulas inserted in the text, and eventually a figure — copying the layout presented in the textbook.
4. **He adds the 'hot spot' for calculation** of the example; now the description of the example as visible to the user is finished.

5. **He adds the formalization** of the example together with its specification necessary for automatically solving the example (which is the prerequisite for user guidance in tutoring).

UC 31.2.0.3 ***An author checks the example(s)***

1. **He checks the executability** of the examle by activating the 'hot spot' and thus calling the math engine, which first checks the syntax of the formalization, and then starts the calculation in a worksheet which pops up. (The calculagion usually will be done in an automated way chosen by the author.)
2. **He adds another example** after the first one has been finished by editing the examples label analogously to (3).
3. **The final check over all examples** newly input so far makes the author sure that the examples are ready to use for tutoring. The sequence of examples is determined by marking the first and the last one, and then it will take some time until all these examples have been calculated automatically. This 'batch mode' of calculation creates some protocol the author can inspect for the results: a list of the examples computed correctly, those not finished correctly, and those not terminating.
   Alternatives:
   (a) **The calculation of an example does not succeed,** which can be seen on the worksheet presenting the calculation. In this case the example author needs expertise for authoring math knowledge (given by himself or by some other person) and requires to start the authoring tools for the math knowledge base.

After adaption of the math knowledge base the respective formalization or even the text and figure of the example may be up to change. I.e. editing all elements of an example (including deletion) must be possible.

Part VI

System View of the UCs and Test Cases

Here some of the use-cases from part V are detailed down to a system view.

The the enumerations # within the use-cases are used for the numbering in the respective system views in this part, detailed into #.a, #.b,… if necessary. Numbers, where no comments additional to the UC is required, are left blank.

Some of the UCs are basis for test cases; these are also described in this part.

Chapter 32

Visit an $\mathcal{ISAC}$ site

Chapter 33

Learn interactively with $\mathcal{ISAC}$

33.1 Initializing the Session

33.1.1 Identifying the User, UC 30.1.1.1

This is a system view of usecase UC.30.1.1.1 on p.188.

1.

2.

3. The system checks the inputs with the stored username and password pair
    (a) login returns a session identifier (if the combination of username and password was correct)
    (b) create a browser-dialog using InformationProcessor's getBDialog method (there is *one* per session)
    (c) create a session-dialog using InformationProcessor's getSDialog (there eventually are *several* per session).
    (d) Call openDGuide from the session-dialog to open a new worksheet-dialog (DialogGuide).

These points are depicted in the interaction diagram on p.211.

33.1.2 Doing an Example from an Example Collection, UC 30.1.2.1

This system-view describes the internal steps of UC.30.1.2.1 where a learner decides to calculate an example. The system-view is depicted in Fig.33.2 We assume, that the user is already logged in and found the example to do.

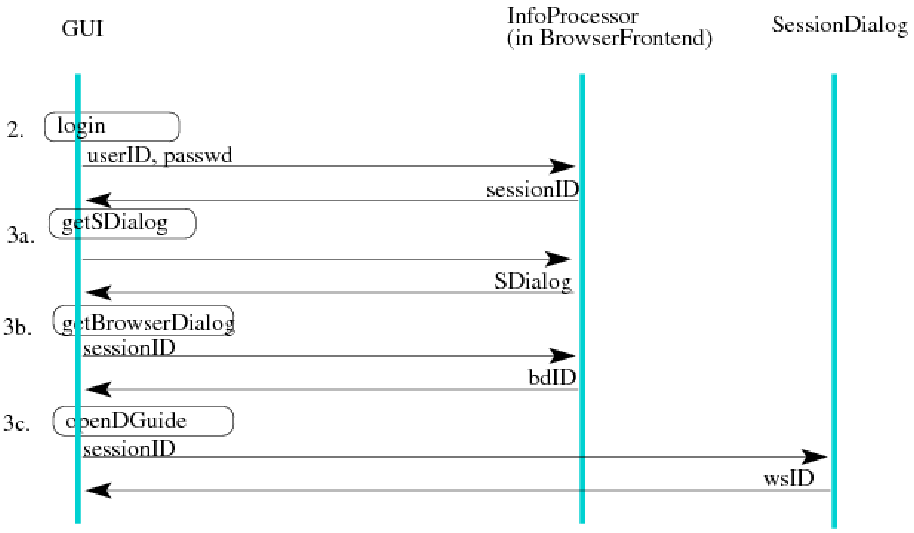


Figure 33.1: Usecase UC.30.1.1.1 identify user.

1.
2. (a) TODO: The user selects an example in the hierarchy-selection-tool, and hits the number of the example in the content-window. (Presently the examples are displayed separately, and thus can be selected in the hierarchy-selection-tool).
   (b) The browser-dialog checks the access-rights of the user (as a member of a group with specific pre-set, time-depending rights) and fetches the formalization of the example from the KE-basis.
   (c) The browser-dialog passes the formlization the the worksheet-dialog (which is responsible for all kinds of access to the SML-kernel)
   (d) The worksheet-dialog calls the SML-kernel to create a new calc-tree and an iterator, and to return a respective calcID.
   (e) Then the workseet-dialog opens a worksheet,
   (f) resets the iterator (to the head of the calc-tree) and fetches the calc-head ('modspec') from the calc-tree
   (g) The worksheet-dialog decides (depending on the user-model) on the presentation of the calc-head and transfers it to the worksheet.
3.
4.
5.

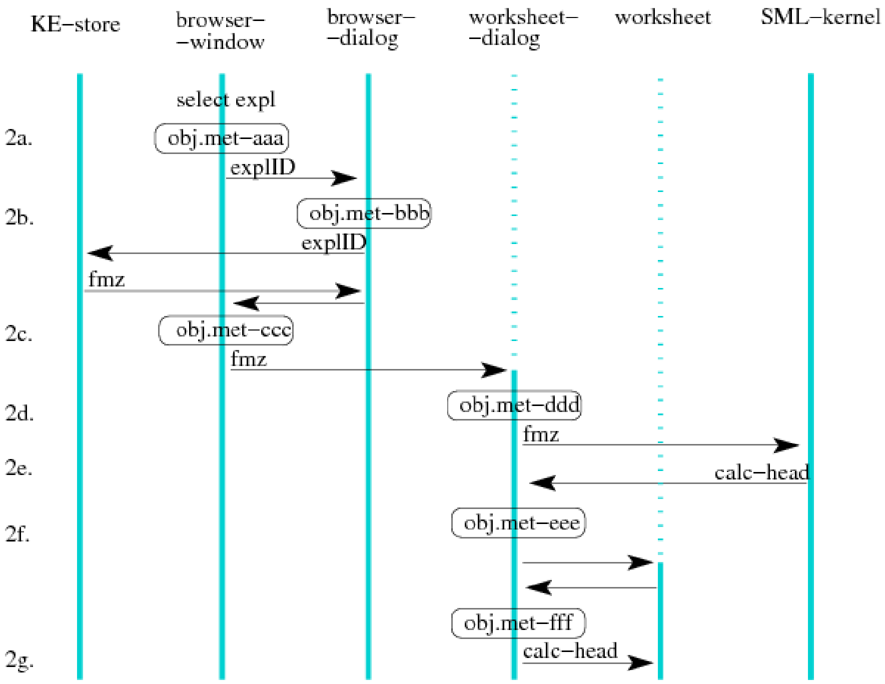


Figure 33.2: Usecase UC.30.1.2.1 start from expl-coll.

33.2 Initializing a Calculation

33.2.1 Go into the problem hierarchy with a particular model, UC 30.2.3.1

This UC on p.195 is detailed as follows and depicted in Fig.33.3:

1. (a) . . . button ⟨problem⟩ . . .
    (b) The worksheet-dialog fetches the calc-head from the SML-kernel and returns the problem-ID *pblID* to the worksheet.
    (c) ??? The worksheet opens a browser-window (or of already open, puts the window to the foreground).
    (d) The browser-window asks the browser-dialog for the content, in particular the model, which eventually has been completed by explanations from the KE-basis.
2. The browser-window displays *pblID* . . .

33.2.2 Matching the Problem, UC 30.2.3.1

The UC on p.**??** is detailed as follows and depicted in Fig.33.4:

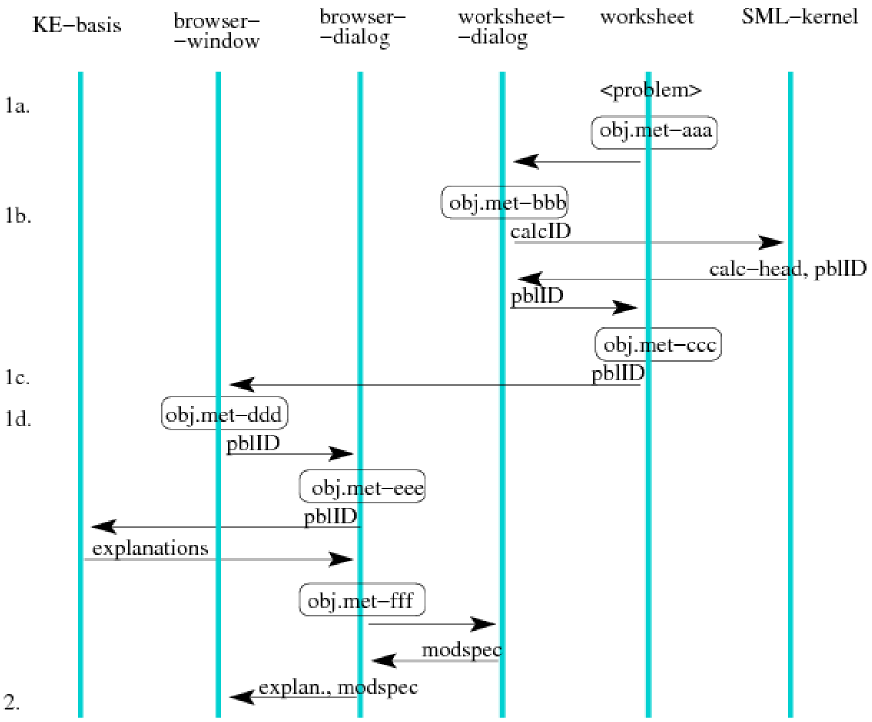


Figure 33.3: Usecase UC.30.2.3.1

1.

2.

3. (a) The key *pblID* of the selected problem P (e.g. *[rational,univariate,equation]*) is passed to the browser-dialog.
   (b) The browser-dialog accesses the KE-basis and filters the explanations according to the access-rights of the user.
   (c) The browser-dialog asks the worksheet-dialog for the data from the SML-kernel.
   (d) The worksheet-dialog passes the *pblID* (and the *calcID*) to the SML-kernel and fetches the modspec (after the problem P has been matched with the model m related to the 'active formula' still marked on the worksheet).

4. The model of m' is displayed in the browser-window together with the hierarchy-selection-tool (still displaying the selection from step 3. above.

5.

6.

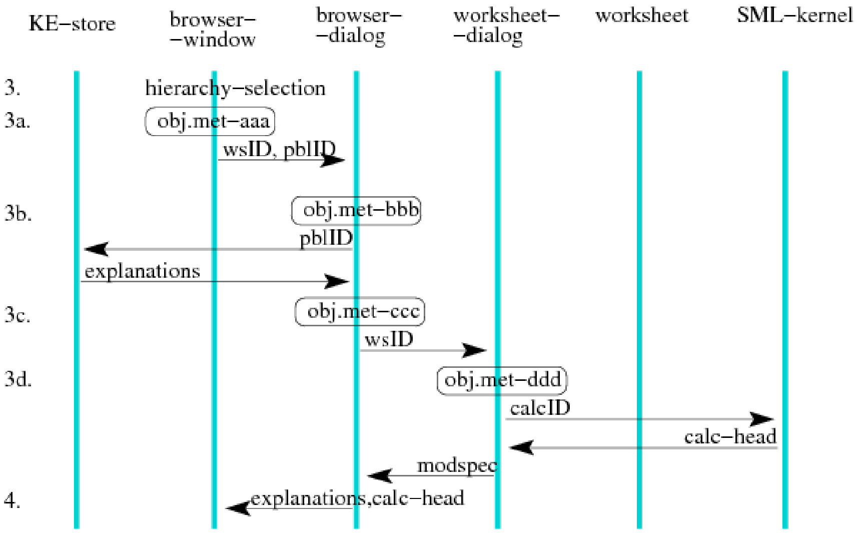


Figure 33.4: Usecase UC **??** match problem.

33.2.3 Having *ISAC* Refine the Problem, UC 30.2.3.4

This UC on p.30.2.3.4 is detailed as follows (the sequence of the function calls is exactly the same as in Fig.33.4, just the names (refine instead match) and arguments of some functions are different):

1.

2.

3. (a) The key *pblID* of the selected problem P (e.g. *[rational,univariate,equation]*) is passed to the browser-dialog.
   (b) The browser-dialog accesses the KE-basis and filters the explanations according to the access-rights of the user.
   (c) The browser-dialog asks the worksheet-dialog for the data from the SML-kernel.
   (d) The worksheet-dialog passes the *pblID* (and the *calcID*) to the SML-kernel and fetches the modspec (after the problem P has been refined to P' with the model m related to the 'active formula' still marked on the worksheet).

4. The model m' is displayed in the browser-window together with *pblID'* marked in the hierarchy-selection-tool. The $\langle match/no-match\rangle$ is set to $\langle match\rangle$.

33.2.4 Transfer a selected problem to the worksheet, UC 30.2.3.8

The system view — ALTERNATIVE A of UC.30.2.3.8 is detailed as follows and depicted in Fig.33.5. Design principle was to have the same functions as in UC.**??** . . . UC30.2.3.4:

1.

2. (a) The browser-window transfers the currently marked *pblID* (and the *wsID*) to the browser-dialog
   (b) The browser-dialog passes *pblID* and *wsID* to the worksheet-dialog in order to call for update of the related calc-tree.
   (c) The worksheet-dialog calls for update of the related calc-tree and receives the resulting modspec (with model m' and with *pblID* in the specification) from the SML-kernel.
   (d) The worksheet comes to the foreground again, and inserts the matched model m' and the updated *pblID* in the specification, all at position p in the worksheet.

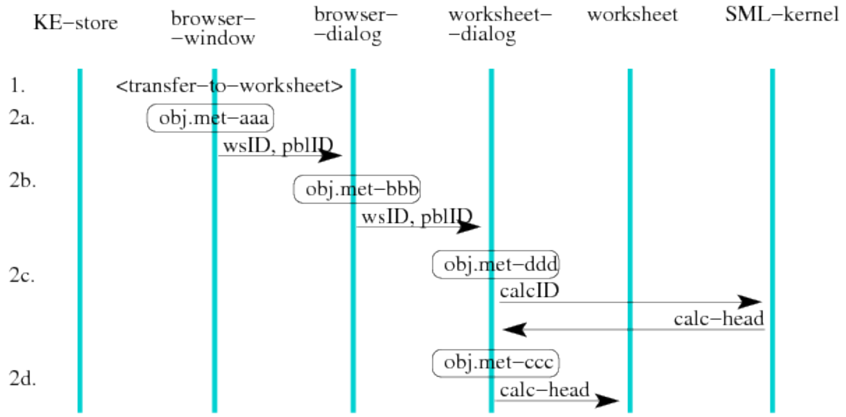


Figure 33.5: UC.30.2.3.8 transfer pbl to worksheet

The system view — ALTERNATIVE B of UC.30.2.3.8 is detailed as follows and depicted in Fig.33.5. Design principle was to avoid unnecessary datatransfer from the SML-kernel (the differences are in points 3 and 4 below):

1. The user hits a button ⟨*transfer_to_worksheet*⟩.
2. The browser-window transfers the currently marked *pblID* (and the *wsID*) to the browser-dialog

3. The browser-dialog passes *pblID*, *wsID* and the modspec to the worksheet-dialog.
4. The worksheet-dialog calls for update of the related calc-tree and receives 'calc-tree updated'.
5. The worksheet comes to the foreground again, and inserts the matched model m' and the updated *pblID* in the specification, all at position p in the worksheet.

Chapter 34

Authoring

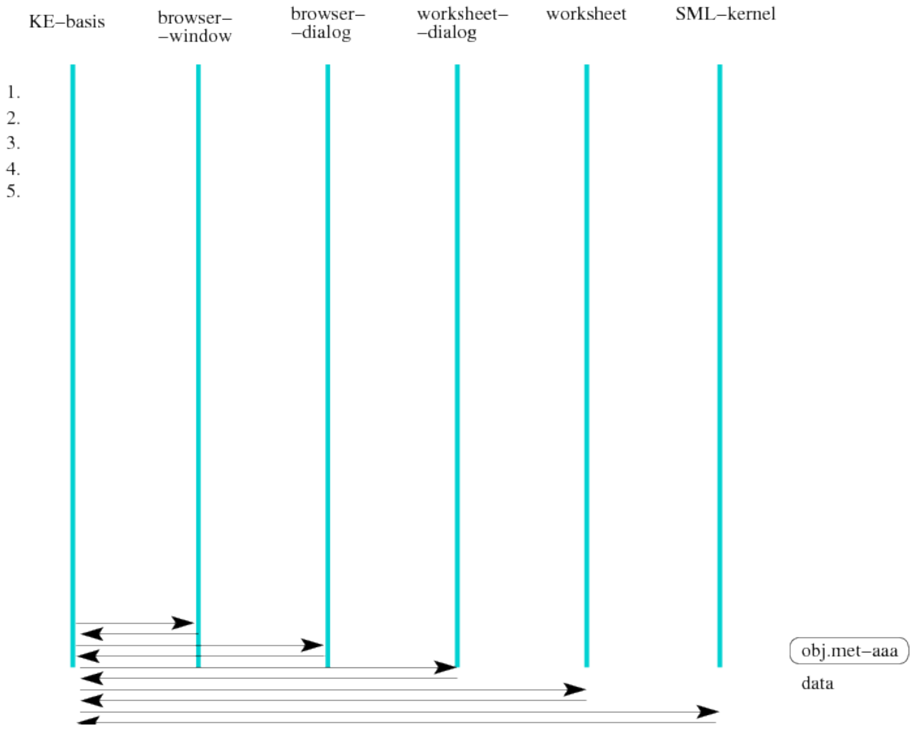


Figure 34.1: Usecase UC.??

Part VII

Appendices

Appendix A

The example for reference

This example intended to illustrate interaction with $\mathcal{ISAC}$ is quoted from [Neu99].

$\mathcal{ISAC}$ features a new kind of calculations in applied mathematics, and it is an issue to make the novel functionality of this software as clear as possible. In order to meet this issue, an example is given to be referenced by the use-cases within this document. The example is taken from a calculus course at highschools; thus it should not pose problems to understand the underlying mathematics. Nevertheless the example covers all major features offered by $\mathcal{ISAC}$.

A.1 Description, formalization and modeling phase

The **description** of an **example**[1] may consist of text, formulas and figures:

> *Given a circle with radius $r = 7$, inscribe a rectangle with length u and width v. Determine u and v such that the rectangles area A is a maximum.*

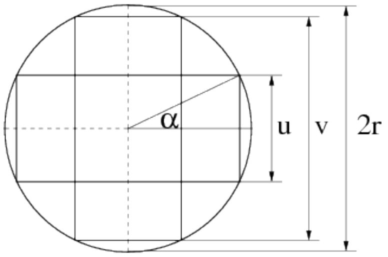


Figure A.1: Figure for the maximum example

---

[1] These **terms** are defined in the appendix of [iT02b] 'List of Terms used in the $\mathcal{ISAC}$-project'

The inital step in solving such an example is, to construct a **model** from the description. The respective model looks like this, if all items are input:

$$\begin{array}{ll}
\text{given :} & [\text{ Constants } \boxed{r=7}\ ] \\
\text{where :} & 0 \leq 7 \\
\text{find X:} & [\text{ Maximum } \boxed{A}, \\
 & \quad \text{AdditionalValues } \boxed{[u,v]}\ ] \\
\text{with :} & A = 2uv - u^2 \ \wedge\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2 \ \wedge \\
 & \qquad \forall\ A'\ u'\ v'.\ A' = 2u'v' - (u')^2 \wedge (\frac{u'}{2})^2 + (\frac{v'}{2})^2 = r^2 \\
\text{relate :} & [\ \boxed{A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2}\ ]
\end{array}$$

The boxes mark the items meant for input by the user, whereas the surrounding information is provided by the system and serves user guidance. If the model is perfect, *ISAC* can solve the example autonomously.

In order to provide userguidance already in the model phase, each example is accompanied by a **formalization** prepared by an author and normally hidden from the user:

$$\begin{array}{ll}
F_I \equiv & (\ \{r=7\},\ \ \{A,[u,v]\}, \\
 & \{0 \leq \frac{u}{2} \leq r,\ \ \{A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2\}\ )
\end{array}$$

$$\begin{array}{ll}
F_{II} \equiv & (\ \{r=7\},\ \ \{A,[u,v]\}, \\
 & \{0 \leq \frac{v}{2} \leq r,\ \ \{A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2\}\ )
\end{array}$$

$$\begin{array}{ll}
F_{III} \equiv & (\ \{r=7\},\ \ \{A,[u,v]\}, \\
 & \{0 \leq \alpha \leq \frac{\pi}{2},\ \{A = 2uv - u^2, \frac{u}{2} = r\sin\alpha, \frac{v}{2} = r\cos\alpha\})
\end{array}$$

In this case the formalization comprises three variants, F_I, F_{II}, F_{III}, which presumerably cover all possibilities students would consider in a particular course. All of such formalizations for *one* example together are called 'formalization' in the sequel.

Given such a formalization and a specification (see below) , *ISAC* can solve an example autonomously *and* in stepwise interaction down to the result.

If *ISAC* is being used to solve an example unknown to the system (i.e. without a hidden specification and formalization prepared by an author) *ISAC* cannot provide user guidance at the beginning of this phase (see **??**). In particular, the items *Constants*, *Maximum* and *AdditionalValues* have to be found in the theory *Descript.thy*.

A.2 Knowledgebase and specification phase

The knowledge base comprises three parts, theories, problems and methods.

Theories contain the knowledge deduced from axioms and definitions by formal proof (done by the interactive theorem prover Isabelle). For the example at hand knowledge is prepared like the following:

```
theory 'calculus'

consts
  d_d         :: "[real, real]=> real"

rules
  diff_sum    "d_d bdv (u + v) = d_d bdv u + d_d bdv v"
  diff_prod   "d_d bdv (u * v) = d_d bdv u * v + u * d_d bdv v"
  diff_quot   "Not (v = 0)     ==> (d_d bdv (u / v) =
                     (d_d bdv u * v - u * d_d bdv v) / v ^ 2)"
  ...
```

Problems capture the aspect of application of knowledge.

```
|
\__ equation
|     \__ univariate
|     |     \__ linear
|     |     \__ polynomial
|     |     \__ rational
|     |     \__ ...
\__ function
|     \__ make
|           \__ by_elimination
|           \__ by_new_variable
|     \__ differentiate
|     |     \__ for_maximum
|     |     |     \__ on_interval
|     |     \__ ...
|     \__ integrate
|           \__ ...
\__ optimization
|     \__ linear
|     |     \__ ...
|     \__ calculus
|           \__ maximum
|           | ...
```

The example at hand shall be described by the *problem [calculus, optimization]* (note the reverse order w.r.t. the hierarchy above, which seems more usual in many cases, e.g. *[linear, univariate, equation]*), and will be broken down into the subproblems *[make, function]*, *[on_interval, for_maximum, differentiate, function]* and *[tool, find_values]*. The root-problem of the example looks like

```
Solve_problem [maximum, calculus, optimization]
given  : [ Constants fix_ ]
where  : map ($ 0 ≤ $) fix_
find   : [ Maximum m_
           AdditionalValues vs_ ]
with   : let x_1 = {m_} ∪ {vs_} ∪ (Vars rs_);
                 x_2 = map primed x_1;
             in  map (op∧) rs_ $ ∧ $
                 ∀ $ x_2 $. ( λ $ x_1 $ . $ map (op∧) rs_ $ ) $ x_2 $
relate : rs_
```

This problem is **matched** with the formalization and yields the model shown above.

Methods describe the algorithms solving the problems. The method solving the example calls the subproblems mentioned:

```
Script Maximum_value (fix_::bool list) (m_::real) (rs_::bool list)
           (v_::real) (itv_::real set) (err_::bool) =
  (let
     e_ = (hd o (filter (Testvar m_))) rs_;
     t_ = (if #1 < Length rs_
           then (Subproblem (Reals,[make, funtion],no_met) [m_, v_, rs_])
           else (hd rs_));
     mx_ = Subproblem (Reals,[on_interval, for_maximum, differentiate,
           function], maximum_on_interval) [ t_, v_, itv_ ]
  in (Subproblem (Reals,[tool,find_values],find_values)
           [ mx_, (Rhs t_), v_, m_, (dropWhile (ident e_), rs_])))
```

An example is given a **specification** ('it is specified') by three pointers into each of the three parts of the knowledge base, i.e. a pointer to a theory, to a problem and to a method. For the example this is *(Differentiate, [calculus, optimization], Maximum_value)*.

A.3 Interaction on the worksheet and the browsers

Within a calculation the centre of interaction with the user is a so-called **worksheet**. At certain points the user may want to view the knowledge base in a so-called **browser-window** and/or select some knowledge, or the dialog presents such a window.

In the modeling phase the user inputs formulas on the worksheet, and on the worksheet they get most of the feedback. After a while the worksheet could look like this:

Solve_problem [maximum, calculus, optimization]
  Model
    given: [*Constants* $r = 7$]
    where: $0 \leq r$
    find: [*Maximum, AdditionalValues* $[u, v]$]
    relate: $[A = 2uv,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2,\ \frac{u}{2} = r \sin\alpha\ ]$

where *Solve_problem [maximum, calculus, optimization]*, *Constants*, *Maximum*, *AdditionalValues*, and $0 \leq r$ have been supplied by the system, and several items are marked: *Maximum* with 'missing', $A = 2uv$ with 'incorrect', and $\frac{u}{2} = r \sin\alpha$ with 'superfluous'. Regarding this kind of feedback users may successfully complete modeling; if they are not capable to do it, they just hit a 'go-on' button and *ISAC* gets the correct model from the hidden formalization and specification. For the following let us assume, that the model is completed correctly:

Solve_problem [maximum, calculus, optimization]
  Model
    given: [*Constants* $r = 7$]
    where: $0 \leq r$
    find: [*Maximum* , *AdditionalValues* $[u, v]$]
    relate: $[A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2, \frac{u}{2} = r \sin\alpha]$
  Specification
    Theory:
    Problem:
    Method:

The superfluous item $\frac{u}{2} = r \sin\alpha$ does not matter, below there are the **specification-fields** to be determined (i.e. input by the user) within the specification phase.

The specification phase requires data from the knowledge base in general. In the case of the examples model above the decision is required, which kind of problem the given model can be matched with: *[linear, optimization]*, or *[calculus, optimization]*, or some other problem. The information necessary for this decision can be found in the hierarchy of problems.

Learners can **browse** the hierarchy of problems (theories, methods) and they can **apply** the problem (theory, method); the latter transfers the problem (theory, method) to the respective field below the model on the worksheet.

For instance, applying the problem *[linear, optimization]* would cause this display on the worksheet:

Solve_problem [maximum, calculus, optimization]

Model
  given: $[\underline{Constants\ r = 7}]$
  where: $0 \leq r$
  find: $[Maximum\ ,\ \underline{AdditionalValues\ [u, v]}]$
  relate: $[\underline{A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2, \frac{u}{2} = r \sin \alpha}]$
Specification
  Theory:
  Problem: *[linear, optimization]*
  Method:

where some items in the model would be marked with some error-feedback. For instance, applying a theory even might cause the feedback 'syntax-error' when the model contains function constants not defined in the theory or in one of the theories parents.

Browsing problems (when started from a model on the worksheet) matches the problem selected with the model on the worksheet, i.e. on the browser-window the same feedbacks are given as for the model on the worksheet described on p.223.

A.4 The solving phase and subproblems

Before we describe $\mathcal{ISAC}$s features for the solving phase, we fix one correct model and one specification (out of several possible ones) as follows:

Solve_problem [maximum, calculus, optimization]
  Model
    given: $[Constants\ r = 7]$
    where: $0 \leq r$
    find: $[Maximum\ A,\ AdditionalValues\ [u, v]]$
    relate: $[A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2, \frac{u}{2} = r \sin \alpha]$
  Specification
    Theory: *Reals*
    Problem: *[maximum, calculus, optimization]*
    Method: *[make_fun, by_elimination]*

The following worksheet shows the whole calculation without the respective models and specifications; this short-presentation could stem from an interaction, where the user is merely interested in the result and let the system calculate autonomously:

Solve_problem [maximum, calculus, optimization]
  1. SubProblem (DiffAppl, [make, function])
    1. *solve_univariate* $\left(\left(\frac{u}{2}\right)^2 + \left(\frac{v}{2}\right)^2 = r^2\right)\ u$
    1'. $L = \left[u = 2\sqrt{r^2 - \left(\frac{v}{2}\right)^2},\ u = -2\sqrt{r^2 - \left(\frac{v}{2}\right)^2}\right]$
    2. $L_1 = \left[u = 2\sqrt{r^2 - \left(\frac{v}{2}\right)^2}\right]$

- *1'.* $A_1 = 2 \cdot 2\sqrt{r^2 - \left(\frac{v}{2}\right)^2} \cdot v - v^2$
- *2. SubProblem (DiffAppl, [on_interval, for_maximum, differentiate, function])*
  - *1.* $\frac{d}{dv}\left(2 \cdot 2\sqrt{r^2 - \left(\frac{v}{2}\right)^2} \cdot v - v^2\right) =$
  - *1'.* $A_1' = 2\sqrt{r^2 - \left(\frac{v}{2}\right)^2} - \frac{v^2}{2\sqrt{r^2-\left(\frac{v}{2}\right)^2}} - 2v$
  - *2. solve_univariate* $\left(2\sqrt{r^2 - \left(\frac{v}{2}\right)^2} - \frac{v^2}{2\sqrt{r^2-\left(\frac{v}{2}\right)^2}} - 2v = 0\right)\ v$
  - *2'.* $L = [v = 234.567]$
- *3. SubProblem (DiffAppl, [find_values, tool])*
- *3'.* $[u = 123.456, v = 234.567]$

$[u = 123.456, v = 234.567]$

Appendix B

$\mathcal{ISAC}$s tactics

Init_Proof_Hid (dialogmode, formalization, specification) transfers the arguments to the math engine, the latter two in order to solve the example automatically. The tactic is not intended to be used by the student; it generates a proof tree with an empty model.

Init_Proof generates a proof tree with an empty model.

Model_Problem problem determines a problemtype (eventually found in the hierarchy) to be used for modeling.

Add_Given, Add_Find, Add_Relation formula inputs a formula to the respective field in a model (necessary as long as there is no facility for the user to input formula directly, and not only select the respective tactic plus formula from a list).

Specify_Theory theory, Specify_Problem problem, Specify_Method method specifies the respective element of the knowledgebase.

Refine_Problem problem searches for a matching problem in the hierarchy below 'problem'.

Apply_Method method finishes the model and specification phase and starts the solve phase.

Free_Solve initiates the solve phase without guidance by a method.

Rewrite theorem applies 'theorem' to the current formula and transforms it accordingly (if possible – otherwise error).

Rewrite_Asm theorem is the same tactic as 'Rewrite', but stores an eventual assumption of the theorem (instead of evaluating the assumption, i.e. the condition)

Rewrite_Set ruleset similar to 'Rewrite', but applies a whole set of theorems ('ruleset').

Rewrite_Inst (substitution, theorem), Rewrite_Set_Inst (substitution, ruleset) similar to the respective tactics, but substitute a constant (e.g. a bound variable) in 'theorem' before application.

Calculate operation calculates the result of numerals w.r.t. 'operation' (plus, minus, times, cancel, pow, sqrt) within the current formula.

Substitute substitution applies 'substitution' to the current formula and transforms it accordingly.

Take formula starts a new sequence of calculations on 'formula' within an already ongoing calculation.

Subproblem (theory, problem) initiates a subproblem within a calculation.

Function formula calls a function, where 'formula' contains the function name, e.g. 'Function (solve $1 + 2x + 3x^2 = 0 \quad x$)'. In this case the modelling and specification phases are suppressed by default, i.e. the solving phase of this subproblem starts immediately.

Split_And, Conclude_And, Split_Or, Conclude_Or, Begin_Trans, End_Trans, Begin_Sequ, End_Sequ, Split_Intersect, End_Intersect concern the construction of particular branches of the prooftree; usually suppressed by the dialog guide.

Check_elementwise assumptions w.r.t. the current formula which comprises elements in a list.

Or_to_List transforms a conjunction of equations to a list of equations (a questionable tactic in equation solving).

Check_postcond: check the current formula w.r.t. the postcondition on finishing the resepctive (sub)problem.

End_Proof finishes a proof and delivers a result only if 'Check_postcond' has been successful before.

Appendix C

Development environment

Delopment environments: There are two environments, (1) SML for the knowledge interpreter and the meth knowledge base, which both are based on Isabelle written in SML, and (2) Java for networking, the dialog and the front-end.

Dev C.0.1 ***SML version*** is Standard ML of New Jersey, Version 110.0.7, September 28, 2000

Dev C.0.2 ***SML library for HTML*** is `smlnj-110.9.1/src/smlnj-lib/HTML`

Dev C.0.3 ***Isabelle version*** is "Isabelle99: October 1999"

Dev C.0.4 ***Java version*** is "1.3.0_01", Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0_01), Java HotSpot(TM) Client VM (build 1.3.0_01, mixed mode)

Dev C.0.5 ***Java applets*** ???

Dev C.0.6 ***Development environments*** are linux version . . . and Unix . . .

Standards and components: A major part of the components concern networking because the knowledge interpreter and the knowledge base reside on a server for two reasons: (1) for efficiency reasons: both need major computer resources, and (2) for administrative reasons: $\mathcal{ISAC}$ will be used in courses initially.

Access to the server is established by standard internet protocols, formats and browsers as much as possible; this is not trivial because importan standards are still under development.

Dev C.0.7 ***Standard for knowledge and example collections*** due to IEEE Learning Technology Standards Committee (LTSC), Learning Object Metadata, Working Draft Document 3 (approved 1999-11-27). TODO: OmDoc, LMML, ?IMS?

Dev C.0.8 ***The middle ware layer is realized by Dinopolis*** version . . .

Dev C.0.9 ***The Browser is Mozilla*** 1.2 - Released November 26, 2002. There should be as few additional software components within the browser as possible.

Dev C.0.10 ***Plugins must not be required by the browser.*** ???

Dev C.0.11 ***The SML-kernel*** comprising the knowledge interpreter and the knowledge base is under development. The interface to the SML-kernel is documented in `???isac/doc/interfaces/me-calc.dvi`.

Documentation and revision control: All documentations and sources are under one revision control.

Dev C.0.12 ***Doxygen*** is used for all java sources.

Dev C.0.13 ***cvs*** contains all documentation, sml sources, the config file of doxygen and the java sources. There are the following directories:

cvs/	doc			UseCases, URD-SRD, ADD-SDD, math-eng
	etc			settings
	lib/	icons		
		scripts		
		tools/		
	src/	dox-config		
		java/	. . . package 1/	. . . klasse 1
				. . . klasse . . .
			. . . package n/	. . . klasse 1
				. . . klasse . . .
			jars	
		sml		

Dev C.0.14 ***Directories*** for all files of the project are the following

bin		
cvs		. . . as above . . .
doxygen		
html/	thy	html-representation of Isabelles theories
	pbl	XML-representation of $\mathcal{ISAC}$'s hierarchy of problems
	met	XML-representation of $\mathcal{ISAC}$'s list of methods
	exp	XML-representation of $\mathcal{ISAC}$'s example collection

Appendix D

List of terms used in the $\mathcal{ISAC}$-project

Active formula (Aktive Formel) is the unique formula marked on the →worksheet, where the next step of calculation will be performed. If another item on the worksheet is marked, the formula closest (.. to be specified) to the marked item is the active formula. If the calculation is finished, the result of the calculation is the active formula. Also →context position.

Browser (Browser): There are browsers for →theories, →problems, →methods and →examples.

Browser dialog (Browser-Dialog) is the part of the →dialog guide which is concerned with the access to the →KE-store.

Browser-Window (Browser-Window) is the presentation of data generated by the →browser, which also handles the http requests generated by the browser window. (The browser-window is that what usually is called a 'browser').

Calc-head (Rechnungskopf) is the first (highly structured) element in a →calculation (i.e. a 'root-problem') and in a subproblem. It consists of a →headline, a →model and of a →specification. On the →worksheet it is represented only by the headline.

Calc-state (Rechenzustand) is given by an internal calc-tree (partially represented on the →worksheet) and a →active formula.

Calculation (Rechengang) leads from the →description of an →example via the →modeling phase, the →specification phase and the →solving phase to the result.

Calculator (Rechner) is that part of the →SML-kernel which does single steps of calculation without touching the →proofstate.

Course admin (Kurs-Administrator) is a person administering the use of *ISAC* for learning within a group of →learners.

Course designer (Kurs-Designer) edits the →example collection which can be solved by a given →math knowledge base (edited by a →mathematics author) and/or edits →explanations within the →math knowledge base.

Context (Kontext) is a relation between the →context position in a certain →calculation and a part of the →math knowledge.

Context position (Kontext Position) is a uniquely formula in a →calculation giving one side of a →context.

Decorated knowledge (Erweitertes Mathematik-Wissen) is the →mathemath knowledge(base) plus →explanations. Each element of the math knowledge can have $0 \ldots n$ explanations (usually specific for courses).

Description (Beschreibung) of an →example consists of →formulas, eventually of text and/or a figure. The →modeling phase transforms the description into a →model.

Dialog atom (Dialog-Atom) is a predefined, minimal unit of interaction between the →learner and *ISAC*. These atoms are symmetrical w.r.t. the two dialog partners.

Dialog author (Dialog-Autor) an expert in learning theory who adapts and extends the →dialog guide.

Dialog guide (Dialog-Komponente) is a component of *ISAC* which consists of the →worksheet-dialog and the →browser-dialogs.

Dialog mode (Dialog-Modus) is assembled from →dialog patterns and supports certain learning strategies, e.g. exploratory learning, written examniation etc. Dialog patterns are designed and implementd by a →dialog author.

Dialog pattern (Dialog-Muster) is assembled from →dialog atoms such that it can adapt to certain situations in a dialog, e.g. if the →learner produces many errors. Dialog patterns are designed and implementd by a →dialog author.

Dialog profile (Dialog-Profil) defines certain →dialog modes for examples in the example collection for a certain course. A dialog profile is defined by a →course designer and set/reset for a specified duration during a course by the →course admin.

Example (Beispiel) is a unit to be calculated and solved separated from others. In general, they are prepared by an author in an →example collection. It consists of an explanation (analoguous to →explanation of an element of the math knowledge), a →formalization and a →specification.

Example browser (Beispiels-Browser) is an interactive representation of the →example collection within the →front-end.

Example collection (Beispielsammlung) contains →examples, each of them consisting of formulas, of a hidden →formalization and →specification, and eventuallly of text and a figure.

Example profile (Beispiels-Profil) describes the structure of an→example collection; this structure provides data for the →dialog guide.

Explanation (Erklärung) is an optional addon (text, formulas, figures, movies, links, →examples and any combination of these) to elements of the →math knowledge base.

Formalization (Formalisierung) contains the formulas in a minimal structure necessary for automated generation of a →model of an example. Together with a →specification this information is sufficient for automatically solve the example.

Formula (Formel) consists of variables, constants and functions constants (for logical, algebraic etc. operators); all these parts, however are not yet structured as a (typed) →term.

Guard (Guard) of a →method: prevents the method's script to be applied to an inappropriate problem. The guard has the same structure as a →modelpattern (and thus sometimes is called a 'guardpattern').

Headline (Problem-Kopf) represents a →calc-head on the worksheet; it either looks like *Problem (Reals, [univariate,equations])* or an Algebrasystem function like *solve*($x^2 + x + 1 = 0,\ x$).

Interpreter (Interpreter) comprises the modules →math engine and the →calculator.

Isabelle is the name of one of the most successful interactive theorem provers; Isabelle provides the →theories containing the deductive part of *ISAC*'s knowledge base.

Item (Item) of a →model, which can be an input item (in the field 'given'), a precondition (in the field 'where'), an ouput item (in the field 'find') or a relation (in the field 'relate'). 'Given', 'find'and 'relate' may be input by the user, where 'where' is supplied by the system. An item consists of the →item-description and the →item-data.

Item-data (Item-Daten) are the formulas following the →item-description.

Item-description (Item-Beschreibung) is an identifier heading each →item in the fields 'given', 'find' and 'relate'. It indicates the kind of data to be input to the respective item by the users, serves typechecking of the data etc.

Item-status (Item-Status) gives feedback to each item of a →model with *one* of the following kinds of status: correct, true, false, missing, incomplete, superfluous, syntaxerror.

KE-store (KE-Basis) is the →decorated →math knowledge plus the →example collection.

Kernel (SML-Kern) comprises the →interpreter and the →knowledge base, all written in SML.

Knowledge base →mathematics knowledge base

Knowledge browser is one of the →theory brosers, →problem browser, →method browser.

Learner (Lernender) a user of $\mathcal{ISAC}$, who uses $\mathcal{ISAC}$ for learning and exercising, i.e. who calculates →examples by use of the →math knowledge base.

Learner model (Lernprofil) is an abstraction over all interactions of a certain learner with $\mathcal{ISAC}$ during a course; this abstraction serves to adapt $\mathcal{ISAC}$'s behaviour to the personal needs of the learner.

Match (Matchen): the →model of an example (or a subproblem) matches the →modelpattern of a problem, or not. This kind of matching is different from the matching-algorithm of symbolic computation: it checks if all →items are input, and evaluates the predicates in 'where'.

Math engine (Mathematik-Maschine) provides for all functions doing →calculations: for applying →tactics, for input →formulas, for calculating resulting formulas, for proposing the next tactic, and for doing calculations automatically; it maintains a →proofstate for each calculation.

Mathematics author (Mathematik-Autor) an expert in computer mathematics who adapts and extends the →mathematics knowledge base.

Mathematics kernel (Mathematik-Kern) replaced by →math engine; please, dont't use anymore !

Mathematics knowledge base (Mathematische Wissensbasis) is stored in three SML-datastructures, in an acyclic graph of →theories, in a hierarchy of →problems, and in a hierarchy of →methods. It is extensible by →math authors and can be both, read by →learners and interpreted by $\mathcal{ISAC}$. Short form is math knowledge. See also →decorated math knowledge.

Method (Methode) contains a →script describing the algorithm for calculating the result, and a guard structured like a →modelpattern in order to inhibit inappropriate application of the script.

Method browser (Methoden-Browser)

Model (Modell) is a part of the →calc-head. It consists of →items (as well as the →modelpattern).

Modelpattern (Modell-Pattern) is the part of a →problem.

Modeling phase (Modellierungs-Phase) is the initial phase in problem solving. In this phase either the system automatically transforms a →formalization of an example into a →model or the user inputs the →items into the model.

Parsing (Parsen) is the process of transforming an 'plain' formula into a typed term. Parsing requires the specification of a →theory containing information about infix position of operators etc.

Problem browser (Problem-Browser)

Problem (Problem) consists of a →modelpattern and some technical elements (→methods solving this problem, rule sets for evaluating the precondition in →matching etc.)

Proofstate (Beweiszustand) replaced by →calc-state; please, don't use anymore !

Rewriting (Rewriting) transforms a formula into a new one by application of a →theorem. $\mathcal{ISAC}$ provides conditional as well as ordered rewriting.

Script (Skript) describes the algorithm solving a particular problem; a script contains →tactics, expressions for guiding the flow of evaluation, and eventually subproblems.

Selection-tool (Auswahls-Tool) displays the contents of either the →example collection, or the dependency graph of →theories, or the hierarchy of →problems, or the hierarchy of →methods; and it allows to select a respective item for detailed display.

SML-kernel →kernel

Solving phase (Lösungs-Phase) is the final phase in problem solving, which generates the solution from the →model and the →specification; this phase may comprise all problem solving phases for one or more subproblems.

Specification (Spezifikation) relates a →model (or a →guard) of a calc-head to the →modelpattern (or guradpattern) of the respective →problem (or →method) while determining a →theory, →a problem and a →method.

Specification phase (Spezifikations-Phase) is the second phase in problem solving, which determines the →theory, →the problem and the →method. This phase can be done automatically and hidden from the user, if the →dialog guide decides to do so. Sometimes, if it is clear from the context, this phase also comprises the →modeling phase.

Step ((Rechen-) Schritt) propagates a →calculation and involves both partners once, i.e. the →learner and the →dialog guide. A step is represented by one of the →dialog atoms.

Tactic (Taktik) is applicable or not to the current →formula within the current proofstate, and generates a new formula accordingly.

Term (Term) is an Isabelle term (simple typed lambda calculus) generated from a →formula by →parsing.

Theorem (Theorem) is a predicate proven true by →Isabelle w.r.t. certain preconditions. Theorems are applied by →rewriting.

Theory (Theorie) is the part of the →math knowledge base which defines (function) constants and axioms. Within a theory usually the related →theorems are being proven by →Isabelle and stored.

Theory Browser (Theorie-Browser)

User (Benutzer) of $\mathcal{ISAC}$ may be one of the following: →visitor, →learner, →math author, →dialog author, →course designer, or →course admin.

Visitor (Besucher) a user of $\mathcal{ISAC}$, which occasionaly browses an $\mathcal{ISAC}$-site, i.e. the →knowledge base and the →example collection.

Worksheet (Arbeitsblatt) contains the →calculation of an →example eventually leading to a result.

Worksheet dialog (Arbeitsblatt-Dialog) is the part of the →dialog guide which is concerned with the interaction between learner (see chapter 1 on p.13) and →math engine.

Appendix E

Abbrevations

UR	user requirement
URD	user requirements document
SR	software requirement
SRD	software requirement document
UC	use case

ADD	architectural design document
SDD	software design document

Bibliography

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide.* Addison-Wesley, 2001.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[Fla99] David Flanagan. *Java Foundation Classes in a Nutshell.* O'Reilly, Sebastopol, CA, September 1999.

[Fla02] David Flanagan. *Java in a Nutshell.* O'Reilly, Sebastopol, CA, 4. edition, March 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gra04] Richard Gradischnegg. Eine java/sml–schnittstelle für *ISAC* auf basis von xml. Master's thesis, University of Applied Sciences, Dpt. Software Engineering, Hagenberg, Upper Austria, 2004. http://www.ist.tugraz.at/projects/isac/publ/da-gradischnegg.ps.gz.

[Gri03] Andreas Griesmayer. Architecture and Knowledge-Represenation of the Web-based Math-Learning-System *ISAC*. Master's thesis, University of Technology, Institute for Softwaretechnology, Graz, Austria, Oct 2003. http://www.ist.tugraz.at/projects/isac/publ/da-griesmayer.pdf.

[Hoc04] Mario Hochreiter. Design and implementation of a graphical user interface for the math-learning-system *ISAC*. Master's thesis, University of Applied Sciences, Dpt. Software Engineering, Hagenberg, Upper Austria, 2004. http://www.ist.tugraz.at/projects/isac/publ/da-hochreiter.ps.gz.

[iT02a] *ISAC* Team. *ISAC* – user requirements document, software requirements document, architectural design document, software design

document, use cases, test cases. Technical report, Institute for Softwaretechnology, University of Technology, 2002.
http://www.ist.tugraz.at/projects/isac/publ/appendices.ps.gz.

[iT02b] $\mathcal{ISAC}$ Team. $\mathcal{ISAC}$ appendices to the analysis and design documents. Technical report, Institute for Softwaretechnology, University of Technology, 2002.
http://www.ist.tugraz.at/projects/isac/publ/appendices.ps.gz.

[iT02c] $\mathcal{ISAC}$ Team. $\mathcal{ISAC}$ use cases. Technical report, Institute for Softwaretechnology, University of Technology, 2002.
http://www.ist.tugraz.at/projects/isac/publ/use.ps.gz.

[iT02d] $\mathcal{ISAC}$ Team. $\mathcal{ISAC}$, interfaces for developers of math knowledge and tools for experiments in symbolic computation. Technical report, IICM, Institute for Softwaretechnology, University of Technology, 2002.
http://www.ist.tugraz.at/projects/isac/publ/mat-eng.pdf.

[KÖ6] Robert Könighofer. Presentation of mathematical knowledge in the $\mathcal{ISAC}$-system. Telematik Projekt/Seminar and Bakk-Arbeit, Oct 2006. Graz University of Technology, Institute for Softwaretechnology.

[Kom07] Georg Kompacher. Context-based access to $\mathcal{ISAC}$s knowledge base. Software-Projekt und Bakk.-Arbeit B, Oct 2007. Graz University of Technology, Institute for Softwaretechnology.

[Kre05] Alan Krempler. Architectural design for integrating an interactive dialogguide into a mathematical tutoring system. Master's thesis, University of Technology, Institute for Softwaretechnology, Graz, Austria, March 2005.
http://www.ist.tugraz.at/projects/isac/publ/da-krempler.pdf.

[Neu99] Walther A. Neuper. Mathematics tutoring II: A mathematics-engine for guided interaction. technical report IST-TEC-99-15, IICM - Inst. f. Software Technology, Technical University, A-8010 Graz, August 1999.
http://www.ist.tugraz.at/projects/isac/publ/IST-TEC-99-15.ps.gz.

[Neu01] Walther A. Neuper. *Reactive User-Guidance by an Autonomous Engine Doing High-School Math.* PhD thesis, IICM - Inst. f. Softwaretechnology, Technical University, A-8010 Graz, 2001.
http://www.ist.tugraz.at/projects/isac/publ/wn-diss.ps.gz.

[Pfa85] G. Pfaff, editor. *Seeheim Workshop on User Interface Management Systems.* Springer, 1985.

[SBCO01] Timothy J. Sliski, Matthew P. Billmers, Lori A. Clarke, and Leon J. Osterweil. An architecture for flexible, evolvable process-driven user-guidance environments. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 33–43. ACM Press, 2001.

[Sch02a] Klaus Schmaranz. *Dinopolis - A Massively Distributable Componentware System*. Habilitation thesis, University of Technology, IICM - Inst. f. Softwaretechnology, A-8010 Graz, Austria, June 2002.

[Sch02b] Klaus Schmaranz. Dolsa - a robust algorithm for massively distributed, dynamic object-lookup services. 2002.