The TaskJuggler Project Management Software

Chris Schlaeger

Table of Contents

Introduction

TaskJuggler is a project planing tool for Linux and UNIX system-based operating systems. All tasks with their dependencies and other properties are edited by using a text editor. The file is then sent through TaskJuggler which in turn produces all sorts of reports in HTML or XML format. The HTML files can be viewed and printed with any web browser. The XML files are used by the Gantt- and PERT chart generators and the KDE Konqueror plug-in.

Since TaskJuggler is not constrained by the performance requirements of real-time editing it can offer a much broader set of features not found in any of the WYSIWYG (What You See Is What You Get) project planing tools. The the project description language is easy to learn and supports the user very effectively during the planing and tracking process.

TaskJuggler does not only honor the task interdependencies but also takes resource constrains and job prioritization into account. The multi-scenario support makes TaskJuggler a versatile tool for both what-if-analysis and plan/actual comparisons. Using TaskJuggler's powerful filtering and reporting algorithms the user can create a variety of task lists, resource usage tables, status reports, project calendars and project accounting statements.

Features and Highlights

- Automatic scheduling of interdependent tasks with resource conflict solver.
- Powerful project description syntax with macro support.
- Flexible working hours and vacation handling.
- Support for shifts.
- Multiple time zone support.
- Flexible resource grouping.
- Project accounting support.
- Task may have initial and finishing costs.
- Resource may have running costs.
- Support for profit and loss analysis.
- HTML and XML report generation.
- Gantt and PERT chart generators.
- Support for plan and actual scenario comparisons.
- Project tracking support.
- Groupware support by using a revision control system such as CVS or RCS on the project description files.
- Support for central resource allocation database.
- Support for cascaded and nested projects.
- Import and export of sub-projects.
- Unicode and localization support.

Basics

TaskJuggler uses one or more text files to describe a project. The main project should be placed in a file with the *.tjp* extension. This main project may contain several sub-projects. Such sub-projects should be placed in files with the *.tjsp* extension. These sub-projects are included in the main project during compile time.

TaskJuggler works like a compiler. The user provides the source files and TaskJuggler computes the contents and creates the output files.

Let's say `AcSo.tjp` is a valid TaskJuggler project file. It contains the tasks of the project and their dependencies. To schedule the project and create report files the user calls TaskJuggler to process it.

```
% taskjuggler AcSo.tjp
```

TaskJuggler will try to schedule all tasks with the specified conditions and generate the reports that were requested with the *htmltaskreport*, *htmlresourcereport* or other report attributes in the input file.

The TaskJuggler Syntax

To introduce the TaskJuggler syntax we create the project plan for a software development project. This example illustrates the basic features of TaskJuggler. The full source code of the example and the resulting reports can be found on the TaskJuggler Web Site [http://www.taskjuggler.org] at http://www.taskjuggler.org/example.php.

A project is always started with the *project* property.

```
project acso "Accounting Software" "1.0" 2002-01-16 2002-04-26 {
  now 2002-03-04
}
```

It tells TaskJuggler the default project ID, a short name for the project, a version number and a start and end date. The start and end dates don't need to be exact, but must enclose all tasks. It specifies the time interval the TaskJuggler scheduler will use to fit the tasks in.

All TaskJuggler properties have a certain number of fixed attributes and a set of optional attributes. Optional attributes are always enclosed in curly braces. In this example we use the optional attributes *now* to set the current day for the scheduler to another value than to the moment the user invokes TaskJuggler. We pick a day during the above specified project period. So we always get the same results of a TaskJuggler run, no matter when we process our first project.

Global Attributes

Besides finding suitable start and end dates for the tasks of the project, TaskJuggler can also do a simple profit and loss analysis. The user has to specify the default daily costs of an employee. This can be changed for specific employees later but it illustrates an important concept of TaskJuggler - inheritance of attributes. In order to reduce the size of the TaskJuggler project file to a still readable minimum, properties inherit many optional attributes from their enclosing scope. We'll see further down, what this actually means. Here we are at top-level scope, so this is the default for all following properties.

```
rate 310.0
currency "EUR"
```

The *rate* attribute can be used to specify the daily costs of resources. The *currency* attribute specifies the currency to be used.

Macros

Macros are another TaskJuggler feature to keep project files small. Macros are text patterns that can be defined once and inserted many times further down the project. A macro always has a name and the text pattern is enclosed by square brackets.

```
macro allocate_developers [
  allocate dev1
  allocate dev2 { load 0.5 }
  allocate dev3
]
```

To use the macro the user simply has to write *${allocate_developers}* and TaskJuggler will replace the *${allocate_developers}* with the pattern. We will use the macro further down in the example and then explain the meaning of the pattern.

Macros can also have parameters. Inside the macro they are referenced by their number.

```
macro foo [
  The first parameter is ${1}.
  And the second is ${2}.
]
```

Now the call

```
${foo "bar1" "bar2"}
```

would be expanded to

```
  The first parameter is bar1.
  And the second is bar2.
```

Using flags

A TaskJuggler feature that the user will probably make heavy use of is *flags*. Once declared the user can attach them to many properties. When generating reports of the TaskJuggler results, the user can use the flags to filter out information and limit the report to exactly those details that should be included.

```
flags team
```

Flags must be declared at global scope before they can be attached to other properties.

Declaring Resources

```
resource dev "Developers" {
  resource dev1 "Paul Smith" { rate 330.0 }
  resource dev2 "Sébastien Bono"
  resource dev3 "Klaus Müller" { vacation 2002-02-01 - 2002-02-05 }

  flags team
}
resource misc "The Others" {
  resource test "Peter Murphy" { maxeffort 0.8 rate 240.0 }
  resource doc "Dim Sung" { rate 280.0 }

  flags team
}
```

This snippet of the example shows the *resource* property. Resources always have an ID and a Name. IDs may only consist of ASCII characters, numbers and the underline character. All global TaskJuggler properties have IDs. They need to be unique within their property class. The ID is needed so that we can reference the property again later without having the need to write the potentially much longer name. Names are strings and as such enclosed in double quotes. Strings may contain any character, even non-ASCII characters. As seen above, resource properties can be nested. *dev* is a virtual resource, a team, that consists of three other resources.

dev1, alias Paul Smith earns more than the normal employee. So the declaration of *dev1* overwrites the inherited default rate with a new value of 330.0. The default value has been inherited from the enclosing scope, resource *dev*. Which in turn has inherited it from the global scope.

The declaration of the resource Klaus Mueller uses another optional attribute. With *vacation* the user can specify a certain time interval where the resource is not available.

Time and Date Specifications

It is important to understand how TaskJuggler handles time intervals. Internally TaskJuggler uses the number of seconds after January 1st, 1970 to store any date. So all dates are actually stored with an accuracy of 1 second. *2002-02-01* specifies midnight February 1st, 2002. Again following the TaskJuggler concept of needing as little info as necessary and extending the rest with sensible defaults, TaskJuggler adds the time 0:00:00 if nothing else has been specified. So the vacation ends on midnight February 5th, 2002. Well, almost. Every time the user specifies a time interval, the end date is not included in the interval. But the second before the date that has been specified is still part of the interval. So Klaus Mueller's vacation ends 23:59:59 on February 4th, 2002.

Peter Murphy only works 6.5 hours a day (actually 6.4 hours). So we use the *maxeffort* attribute to limit his daily working hours. We could also define exact working hours using the *shift* property, but we ignore this for now. *shift* enables the user to specify the exact working hours for each day of the week.

Note that we have attached the flag *team* after the declaration of the sub-resources to the team resources. This way, they flags don't get inherited by the sub-resources. If we would have declared the flags before the sub-resources, then they would have the flags attached as well.

Declaring Accounts

The use of our resources will create costs. For a profit and loss analysis, we need to balance the costs against the customer payments. So that we don't get lost with all the amounts, we declare 3 accounts to credit the amounts to. We create one account for the development costs, one for the documentation costs and one for the customer payments.

```
account dev "Development" cost
account doc "Dokumentation" cost
account rev "Payments" revenue
```

The *account* property has 3 fixed attributes, an ID, a name and a type. The type can either be *cost* or *revenue*. For the analysis TaskJuggler subtracts the total amount of all cost accounts from the total amount of all revenue accounts.

Specifying Tasks

Let's focus on the real work now. The project should solve a problem - the creation of an accounting software. Since the job is quite complicated we break it down into several sub tasks. We need to do a specification, develop the software, test the software and write a manual. In TaskJuggler syntax this would look like that:

```
task AcSo "Accounting Software" {
  task spec "Specification"
  task software "Software Development"
  task test "Software testing"
  task deliveries "Milestones"
}
```

Just like resources, tasks are declared by using the *task* keyword followed by an ID and a name string. All TaskJuggler properties have their own namespaces. This means, that it is quite OK to have a resource and a task with the same ID. Tasks may have optional attributes which can be tasks again, so tasks can be nested. In contrast to all other TaskJuggler properties, task IDs inherit the ID of their enclosing task as a prefix to the ID. The full ID of the *spec* task is *AcSo.spec*.TaskJuggler uses dots to concatenate the IDs of hierarchical tasks to absolute IDs.

To track important milestones of the project, we also added a task called Milestones. This task, like most of the other task will get some sub tasks later on. We consider the specification task simple enough that we don't have to break it into further sub tasks. So let's add some more details to it.

Task Durations

```
  task spec "Specification" {
    effort 20d
    ${allocate_developers}
    depends !deliveries.start
  }
```

The effort to complete the task is specified with 20 man days. Alternatively we could have used the *length* attribute or the *duration* attribute. *length* specifies the duration of the task in working days while *duration* specifies the duration in calendar days. Contrary to *effort* these two don't require to have a specification of the involved resources. If resources are specified they are allocated when available but they do not affect the total duration of the task.

Allocating Resources

Since *effort* specifies the duration in man days, we need to say who should be allocated to the task. The task won't finish before the resources could be allocated long enough to reach the specified effort.

Here we use the above mentioned macro *allocate_developers*. The expression

```
    ${allocate_developers}
```

is simply expanded to

```
  allocate dev1
  allocate dev2 { load 0.5 }
  allocate dev3
```

If it is necessary to allocate the same bunch of people to several task, the macro saves some writing. One could have written the *allocate* attributes directly instead of using the macro. Since the allocation of multiple resources to a task is a very common place for macro usage, we found it a good idea to use it in this example as well.

One more interesting thing to note is the fact that we like the resource *dev2* only to work 50% of the day on this task, so we use the optional attribute *load* to specify this.

For TaskJuggler to schedule a task it needs to know either the start and end criteria of a task, or one of them and a duration specification. The start and end criteria can either be fixed dates or relative dates. Relative dates are specification of the type "task B starts after task A has finished". Or in other words, task B depends on task A. In this example the *spec* task depends on a sub tasks of the *deliveries* tasks. We haven't specified it yet, but it has the local ID *start*.

Task Dependencies

To specify the dependency between the two task we use the *depends* attribute. The attribute must be followed by one or more task IDs. If more than one ID is specified, each ID has to be separated with a comma from the previous one. Task IDs can be either absolute IDs or relative IDs. An absolute ID of a task is the ID of this task prepended by the IDs of all enclosing tasks. The task IDs are separated by a dot from each other. The absolute ID of the specification task would be *AcSo.spec*.

Relative IDs always start with one or more exclamation marks. Each exclamation mark moves the scope to the next enclosing task. So *!deliveries.start* is expanded to *AcSo.deliveries.start* since *AcSo* is the enclosing task of *deliveries*. Relative task IDs are a little bit confusing at first, but have a real advantage over absolute IDs. Sooner or later the user wants to move tasks around in the project and then it's a lot less likely that one has to fix dependency specifications of relative IDs.

The software development task is still too complex to specify it directly. So we split it into sub tasks again.

Task Priorities

```
task software "Software Development" {
  priority 1000
  task database "Database coupling"
  task gui "Graphical User Interface"
  task backend "Back-End Functions"
}
```

We use the *priority* attribute to mark the importance of the tasks. By default all tasks have a priority of 500 unless the parent tasks specifies it differently. Setting the priority to 1000 marks the task as most important task, since the possible range is 1 (not important at all) to 1000 (ultimately important). *priority* is an attribute that is inherited to sub tasks if specified before the sub tasks declaration. So all sub tasks of *software* have a priority of 1000 as well.

```
  task database "Database coupling" {
    effort 20d
    depends !!spec
    allocate dev1
    allocate dev2
  }
```

The work on the database coupling should not start before the specification has been finished. So we use again the *depends* attribute to let TaskJuggler know about this. This time we use two exclamation marks for the relative ID. The first one puts us in the scope of the enclosing *software* task. The second one is to get into the *AcSo* scope that contains the *spec* tasks. This time we allocate resources directly without using a macro.

Comparing multiple Project Scenarios

```
  task gui "Graphical User Interface" {
    effort 35d
    actualeffort 40d
    depends !database, !backend
    allocate dev2
    allocate dev3
  }
```

TaskJuggler can schedule the project for two different scenarios. The first scenario is called the plan scenario. The other is referred to as the actual scenario. Many of the reports allow to put the values of both scenarios side by side to each other, so one can compare the two scenarios. The two scenarios must have the same task structure and the same dependencies. But the start and end dates of tasks as well as the duration and the resulting resource allocation may vary. In the example we have planed the work on the graphical user interface to be 35 man days. It turned out that we actually needed 40 man days. The *actualeffort* attribute can be used to specify this.

Tracking the Project Status

```
  task backend "Back-End Functions" {
    effort 30d
    complete 95
    depends !database, !!spec
    allocate dev1
    allocate dev2
  }
```

By default TaskJuggler assumes that all tasks are on schedule. Sometimes the user wants to generate reports, that show how much of a task has actually been completed. TaskJuggler uses the current date for this unless another date has been specified by using the *now* attribute. If a task is ahead of schedule or late this can be specified using the *complete* attribute. It specifies how many percent of the task have been completed up to the current date. In our case the back-end implementation is slightly ahead of schedule as we will see from the report.

Specifying Efforts

```
task test "Software testing" {

  task alpha "Alpha Test" {
    effort 1w
    depends !!software
    allocate test
    allocate dev2
  }

  task beta "Beta Test" {
    effort 4w
    depends !alpha
    allocate test
    allocate dev1
  }
}
```

The software testing task has been split up into an alpha and a beta test task. The interesting thing here is, that efforts can not only be specified as man days, but also man weeks, man hours, etc. Per default TaskJuggler assumes a man week is 40 man hours or 5 man days. These values can be changed using the *dailyworkinghours* attribute.

Crediting costs to Accounts

Let’s go back to the outermost task again. At the beginning of the example we stated that we want to credit all development work to one account with ID *dev* and all documentation work to the account *doc*. To achieve this, we use the attribute account to credit all tasks to the *dev* account.

```
task AcSo "Accounting Software" {

  account dev

  task software "Software Development" {
```

Since we specify the attribute for the top-level task before we declare any sub tasks, this attribute will be inherited by all sub tasks and their sub tasks and so on. Since the only exception is the writing of the manual, we need to change the account for this task again since it is also a sub task of *AcSo*.

```
  task manual "Manual" {
    effort 10w
    depends !deliveries.start
    allocate doc
    allocate dev3
    account doc
  }
```

Specifying Milestones

All task that have been discussed so far, had a certain duration. We did not always specify the duration explicitly, but we expect them to last for a certain period of time. Sometimes the user just wants to capture a certain moment in the project plan. These moments are usually called milestones since they have some level of importance for the progress of the project.

TaskJuggler has support for milestones as well. They are handled as special types of tasks. By using the optional attribute *milestone* for a task, this task is declared a milestone. Milestones have no duration, so it’s illegal to specify any duration criteria, or a non identical start and end date.

```
  task deliveries "Milestones" {

    account rev

    task start "Project start" {
      milestone
      start 2002-01-16
      actualstart 2002-01-20
      startcredit 33000.0
    }

    task prev "Technology Preview" {
      milestone
      depends !!software.backend
      startcredit 13000.0
    }

    task beta "Beta version" {
      milestone
      depends !!test.alpha
      startcredit 13000.0
    }

    task done "Ship Product to customer" {
      milestone
      # maxend 2002-04-17
      depends !!test.beta, !!manual
      startcredit 14000.0
    }
  }
}
```

We have put all important milestones of the project as sub tasks of the *deliveries* task. This way they show up nicely grouped in the reports. All milestone have either a dependency or a fixed start date. For the first milestone we have used the attribute *start* to set a fixed start date. All other tasks have direct or indirect dependencies on this task. Moving back the start date will slip the whole project. This has actually happened, so we use the *actualstart* attribute to specify the real start of the project 4 days later.

Every milestone is linked to a customer payment. By using the *startcredit* attribute we can credit the specified amount to the account associated with this task. Since we have assigned the *rev* account to the enclosing task, all milestones will use this account as well.

The line within the definition of the task *done* that starts with a hash is a comment. If TaskJuggler finds a hash it ignores the rest of the line. This way the user can include comments in the project. The *maxend* attribute specifies that the task should end no later than the specified date. This information is not used for scheduling but only for checking the schedule afterwards. Since the task will end later than the specified date, commenting out the line would trigger a warning.

Now the project has been completely specified. Stopping here would result in a valid TaskJuggler file that could be processed and scheduled. But no reports would be generated to visualize the results.

Generating Reports of the scheduled Project

TaskJuggler offers a number of report types. Probably the most popular ones are the HTML reports. The user can advice TaskJuggler to generate one or more HTML pages that contain lists of tasks, resources or accounts.

Before we start with the reports, we present another macro. We like to add a navigation bar to each HTML page that holds a number of buttons. Each button changes the page to another report. This way we can create a navigation bar that holds links to all reports. Since we have created a macro, we can add the navigation bar to all reports without much hassle. The navigation bar is constructed with HTML tags. If the user is not familiar with HTML this will look very strange but it is a cool feature we would like to demonstrate. Certainly the user can use TaskJuggler to it's full extend without having to learn HTML code.

The HTML code is injected into the reports using the *rawhead* attribute. This will put the HTML code close to the top of the HTML page right after the body started. As can be seen here, string parameters of attributes can be enclosed in single quotes as well. This is handy, if the string itself needs to contain double quotes.

```
macro navbar [
rawhead
  '<p><center>
  <table border="2" cellpadding="10">
  <tr>
    <td class="default" style="font-size:120%" rowspan="2">
    <a href="Tasks-Overview.html">Tasks Overview</td>
    <td class="default" style="font-size:120%" rowspan="2">
    <a href="Tasks-Details.html">Tasks Details</td>
    <td class="default" style="font-size:120%" rowspan="2">
    <a href="Staff-Overview.html">Staff Overview</td>
    <td class="default" style="font-size:120%" rowspan="2">
    <a href="Staff-Details.html">Staff Details</td>
```

```
   <td class="default" style="font-size:120%" rowspan="2">
   <a href="Accounting.html">Accounting</td>
   <td class="default" style="font-size:120%" rowspan="2">
   <a href="acso.eps">GANTT Chart (Postscript)</td>
  </tr>
  </table>
  </center></p><br>'
]
```

Generating HTML Task Reports

As the first report, we would like to have a general overview of all tasks with their computed start and end dates. For better readability we include a calendar like column that lists the effort for each week. The report shell consists mainly of a listing of the tasks in a table form. The property *htmltaskreport* generates exactly this, a list of all tasks in a table. The columns are flexible and can be specified with the *column* attribute. For this report we like to see the number, the name, the start and end date, a weekly calendar and the total effort in the table.

```
htmltaskreport "Tasks-Overview.html" {
  ${navbar}
  columns no, name, start, end, weekly, effort
  headline "Accounting Software Project"
  caption "This table shows the load for each task on a weekly basis.
  All values are man-days."
}
```

With the *headline* attribute we can specify a headline for the report. To have a little more info included as well, we use the *caption* attribute. Both of these attributes are followed by the string to be included into the report.

Now we like to generate a report that contains a lot more details about the task. The weekly calendar is replaced by a daily calendar. The weekly calendar had a column for each week. The daily calendar features a column for each day. The column includes the load for the task for the week or day and a colored background in case the task is active that day or week.

```
htmltaskreport "Tasks-Details.html" {
  ${navbar}
  columns no, name, start, end, daily
  headline "Accounting Software Project"
  caption "This table shows the load of each day for all the tasks.
  Additionally the resources used for each task are listed. Since the
  project start was delayed, the actual schedule differs significantly
  from the original plan."
  hideresource 0
  showactual
}
```

We also like to list all assigned resources right after each task. Normally resources are hidden in task reports but they can be enabled by using the *hideresource* attribute. The attribute is followed by a logical expression that specifies what resources to hide. The expression is evaluated for each resource and if the result is true (not 0) than the resource is hidden. Since we want to show all resources we put a 0 in, so it's false for all resources.

To add even more information to this report, we also turn on the reporting of values of the actual scenario by using the *showactual* attribute. This causes TaskJuggler to split the lines of the report into two where appropriate and report the actual value underneath the plan value.

Generating HTML Resource Reports

The previous report listed the resources per task. Now we want to generate a report the lists all resources. It's again a report with a weekly calendar. This time we use the attribute *loadunit* to report the load in hours instead of days.

```
htmlresourcereport "Staff-Overview.html" {
  ${navbar}
  columns no, name, weekly, effort
  showactual
  loadunit hours
  headline "Weekly working hours for the Accounting Software Project"
}
```

Now a similar report but with much more details. We want to include that tasks again, this time each resource should be followed by the tasks the resource is assigned to. In *htmltaskreports* resources are hidden by default while in *htmlresourcereports* tasks are hidden by default. To include tasks the attribute *hidetask* needs to be used. It is followed by a logical expression just like *hideresource*.

```
htmlresourcereport "Staff-Details.html" {
  ${navbar}
  columns name, daily, effort
  showactual
  hidetask 0
  hideresource team
  sortresources nameup
  loadunit hours
  headline "Daily working hours for the Accounting Software Project"
}
```

When specifying the resources we have grouped the resources into two teams by creating two pseudo resources that had the real employees as sub resources. We have attached the flag *team* to those pseudo resources. We now use this flag as logical expression for *hideresource*. So all resources that have this flag will be hidden in the report. For better readability we sort the resource list by name in ascending order. The attribute *sortresources* is taking care of this.

Generating HTML Account Reports

To conclude the HTML reports a report that shows how badly the project is calculated is generated. The company won't get rich with this project. Due to the slip, it actually needs some money from the bank to pay the salaries.

```
htmlaccountreport "Accounting.html" {
 ${navbar} columns no, name, total, monthly
 headline "P&L for the Accounting Software Project"
 caption "The table shows the profit and loss analysis as well as the
        cashflow situation of the Accounting Software Project."
 accumulate
 showactual
}
```

The *htmlaccountreport* property produces similar reports as the above ones, but it lists accounts instead of tasks or resources. The *total* column shows the value of the account at the end of the reported time interval. The *accumulate* attribute puts the calendar in accumulation mode. The monthly columns list the value of the account at the end of the month. Normally the amount that has been added or subtracted from the account would be listed.

Generating XML Reports

Finally we generate an XML report that contains all info about the scheduled project. This report will be used by tjx2gantt to create a nice GANTT chart of our project. The file can also be read by tools like tjGUI or the KDE Konqueror plug-in.

```
xmlreport "AccountingSoftware.tjx"
```

Conclusion

TaskJuggler is a substantially different approach to project planing and tracking tools. The powerful textual project description language allows the user to efficiently capture the properties of the project. Since the scheduler it is not hampered by the performance requirements of GUI based tools it offers many important features not found in GUI tools. For future versions we plan an optimizer that can find best results even with complex tasks and skill-based resource selections.