



Programmers Manual

KD Gantt

The contents of this manual and the associated KD Gantt software are the property of Klarälvdalens Datakonsult AB and are copyrighted. Any reproduction in whole or in part is strictly prohibited without prior written permission by Klarälvdalens Datakonsult AB.

KD Gantt and the KD Gantt logo are trademarks or registered trademarks of Klarälvdalens Datakonsult AB in the European Union, the United States, and/or other countries. Other product and company names and logos may be trademarks or registered trademarks of their respective companies.



Table of Contents

| | |
|--------------------------------------|----|
| 1. Introduction | 1 |
| What You Should Know | 1 |
| Structure of This Manual | 1 |
| What's next | 2 |
| 2. Anatomy of a Gantt Chart | 3 |
| Definition of a Gantt Chart | 3 |
| Gantt Chart Examples | 3 |
| What's Next | 4 |
| 3. Your First Own Gantt Chart | 5 |
| Procedure | 5 |
| Examples | 5 |
| What's Next | 9 |
| 4. Basic Usage | 10 |
| Working With Items | 10 |
| Working With Constraints | 15 |
| Working With the Grid | 17 |
| User Interaction | 18 |
| Next Chapter | 18 |
| 5. Advanced Usage | 19 |
| Working With The GraphicsView | 19 |
| Creating Your Own ItemDelegate | 22 |



List of Figures

| | |
|--|----|
| 2.1. A Basic Gantt Chart | 3 |
| 2.2. An Extended Gantt Chart | 4 |
| 4.1. The Different Items | 10 |
| 4.2. Customizing the Brush | 13 |
| 4.3. Customizing Lines | 14 |
| 4.4. Customizing Start and End Times | 15 |
| 4.5. A Simple Constraint | 16 |
| 4.6. Using Day Scale | 17 |
| 4.7. Using Hour Scale | 17 |
| 4.8. Changed Day Width | 17 |
| 4.9. Customized Grid | 18 |
| 5.1. Only Using GraphicsView | 19 |
| 5.2. Custom Item Painting | 22 |
| 5.3. Custom Constraint Drawing | 24 |
| 5.4. Using Your Own Items | 26 |

List of Tables

| | |
|----------------------------|----|
| 4.1. Item data table | 11 |
|----------------------------|----|

Chapter 1. Introduction

KD Gantt is Klarälvdalens Datakonsult AB's package for creating Gantt charts in Qt applications. KD Gantt features a large number of configuration options to tailor these Gantt charts to suit your needs. Since all configuration settings have reasonable defaults you can usually get by with setting only a handful of parameters and relying on the defaults for the rest.

This is the KD Gantt Programmer's Manual. It will get you started with creating your charts and provides lots of pointers for the advanced features. Besides this manual, there are two other documents:

- In the `doc` directory you will find an `Install` file explaining how to install KD Gantt on your platform or how to build it from the source code.
- KD Gantt also comes with an extensive Reference Manual that has been generated directly from the source code.

You should refer to these in conjunction with this Programmer's Manual.

▶ What You Should Know

You should be familiar with writing Qt applications and have a working knowledge of C++. If you are in doubt about how a Qt class mentioned in this Programmer's Guide works, please check the Qt reference documentation or a good book about Qt.

▶ Structure of This Manual

This manual starts with an introduction to Gantt charts in Chapter 2, *Anatomy of a Gantt Chart* where you will learn how a Gantt chart is constructed. We will then have a closer look at how to create a basic Gantt chart in KD Gantt, in Chapter 3, *Your First Own Gantt Chart*.

Following the introduction, we will delve deeper into the various parts of KD Gantt, these are collected in Chapter 4, *Basic Usage*. This chapter explains how you can modify the different components and gives some helpful pointers.

In the final chapter, we will have a closer look at some of the more advanced concepts in KD Gantt, such as how to create your own items and customize the appearance of the existing ones. You will find this information in Chapter 5, *Advanced Usage*.



What's next

In the next chapter you will learn about the components of a Gantt chart...

Chapter 2. Anatomy of a Gantt Chart

In this chapter, we will have a look at some different charts and explain their components.

Definition of a Gantt Chart

A Gantt chart is a horizontal bar chart used to schedule and track different tasks, for example, a software project.

It is constructed with a horizontal axis showing a timeline which can be divided into several smaller parts, such as hours, days, weeks, etc. A Gantt chart also has a vertical axis containing the different tasks included in the project. These tasks can be sub-items to other tasks or simply stand-alone tasks.

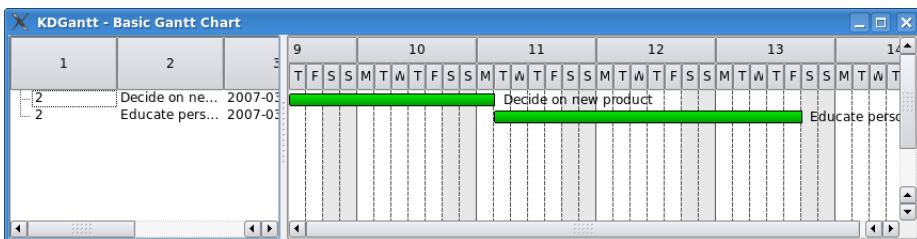
Gantt Chart Examples

The following two Gantt charts show a few of the possible uses of KD Gantt. A more detailed description of the parts shown in these examples can be found in later chapters.

The complete source code for these two examples will be shown in the next chapter, where we describe more closely how they work.

A Basic Gantt Chart

Figure 2.1. A Basic Gantt Chart



This is a screenshot of an application running KD Gantt. As you can see, the main widget is separated into two views — one view containing the tasks, and the other view containing the actual schedule with the timeline.

The left view in the screenshot above contains, as stated, all the scheduled tasks. All tasks have a name, type, a start date and an end date. Some tasks, such as milestones, only have

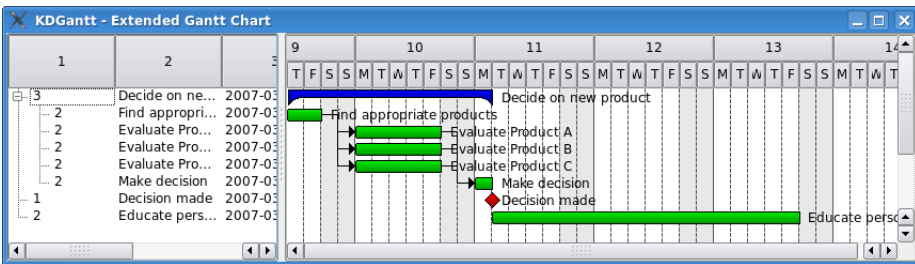
one date, because these are not actionable tasks, but rather a note on certain events, such as a delivery.

In the right view, we have the actual schedule. In this simple chart, we only have two tasks which are performed sequentially.

A Gantt chart is often better the more verbose it is. In the following section, we will extend the simple chart we have seen so far to show more details about the performed tasks.

An Extended Gantt Chart

Figure 2.2. An Extended Gantt Chart



In the above screenshot, you can see examples of subtasks as well as task links. We have also changed the type of the first task to a `KDGantt::TypeSummary`. Different item types are described further in Section , “Working With Items”.

The task links used in the Gantt chart above show dependencies between the tasks. You cannot evaluate the different products until you know which products to evaluate, and that is discovered during the "Find appropriate products" task. Also, you cannot decide on which product to use until all alternatives are evaluated. The different usage alternatives for task links are shown in more detail in Section , “Working With Constraints” where we will even show a few code examples on how you use them.

We have added an event item as well, showing when the actual decision event should take place. This is an object of type `KDGantt::TypeEvent`, with the start time set to the end time of the "Make decision"-task.

▶ What's Next

In the next chapter we will have a look at how to create your first Gantt chart...

Chapter 3. Your First Own Gantt Chart

In this chapter, we will have a look at how to create a basic Gantt chart, first outlining the general procedure, then later showing a code example.

► Procedure

The creation of a Gantt chart is a very easy and straightforward procedure:

1. Create an object of type `KDGantt::View`. This is the actual Gantt view, consisting of the tree view to the left and the schedule on the right. From this object you can access, apart from the tree view and the schedule, the underlying model that stores the data, as well as the constraint model and the grid.
2. Create a model, and add items to this model. These items are the entries in the Gantt chart, consisting of a type, name, start date, end date and, optionally, completion percentage. There are five different types to choose among: `KDGantt::TypeNone`, `KDGantt::TypeEvent`, `KDGantt::TypeTask`, `KDGantt::TypeSummary` and `KDGantt::TypeUser`. These five types are described in more detail in Section , “Working With Items”. You will need to set the start and end times of these items, to have them displayed in the Gantt view. How to do this will be explained in the following examples.

► Examples

To make the discussion more practical, we will now show you a few code examples. The output of these examples is shown in Figure 2.1 and Figure 2.2.

A Basic Gantt Chart

The following code is also available in a complete example in `step01a.cpp`.

```
1
  #include <QApplication>
  #include <QStandardItemModel>

5  #include <KDGanttView>
  #include <KDGanttDateTimeGrid>

  class MyStandardItem : public QStandardItem {
  public:
10   MyStandardItem( const QVariant& v ) : QStandardItem()
```

```

    {
        setData( v, Qt::DisplayRole );
    }
    MyStandardItem( const QString& v ) : QStandardItem()
15  {
        setData( v, Qt::DisplayRole );
    }
};

20 int main( int argc, char* argv[] )
    {
        QApplication app( argc, argv );

        QStandardItemModel model;❶
25  model.appendRow( QList<QStandardItem*>()❷
        << new MyStandardItem( QString( "Decide on new product" ) )
        << new MyStandardItem( KDgantt::TypeTask )
        << new MyStandardItem( QDateTime( QDate( 2007, 3, 1 ) ) )
        << new MyStandardItem( QDateTime( QDate( 2007, 3, 13 ) ) )
30  << new MyStandardItem( QString::null ) );

        model.appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Educate personel" ) )
        << new MyStandardItem( KDgantt::TypeTask )
35  << new MyStandardItem( QDateTime( QDate( 2007, 3, 13 ) ) )
        << new MyStandardItem( QDateTime( QDate( 2007, 3, 31 ) ) )
        << new MyStandardItem( QString::null ) );

        KDgantt::DateTimeGrid grid;❸
40  grid.setDayWidth( 16 );

        KDgantt::View view;❹
        view.setGrid( &grid );
        view.setModel( &model );

45  view.setWindowTitle( "KDgantt - Basic Gantt Chart" );
        view.show();

        return app.exec();
50 }

```

- ❶ The model is created, and in the following lines the data is added to the model. You can of course use your own model for this instead of using `QStandardItemModel`.
- ❷ The data is added to the model, row by row. As you can see, the first column of the row contains the type of entry, in this case a `KDgantt::TypeTask`, and the following columns contains the caption, start date, end date, and completion percent, in that order.
- ❸ As we are dealing with quite long tasks we will need to modify the grid in order to be able to see the entire schedule without scrolling. In this case, we set the day width of the grid to 16 pixels. This modifies the lower scale on the schedule, while the higher scale is calculated automatically.
- ❹ Last, we create the actual view, set it up with the grid, supply the model, give it a title and show it.

An Extended Gantt Chart

The following example is a more verbose version of the Gantt chart above. It can also be found in `step01b.cpp`.

```
1
#include <QApplication>
#include <QStandardItemModel>

5 #include <KDGanttView>
#include <KDGanttDateTimeGrid>
#include <KDGanttConstraintModel>

class MyStandardItem : public QStandardItem {
10 public:
    MyStandardItem( const QVariant& v ) : QStandardItem()
    {
        setData( v, Qt::DisplayRole );
    }
15    MyStandardItem( const QString& v ) : QStandardItem()
    {
        setData( v, Qt::DisplayRole );
    }
};

20 int main( int argc, char* argv[] )
{
    QApplication app( argc, argv );

25    QStandardItemModel model;

    QStandardItem* topitem
        = new MyStandardItem( QString( "Decide on new product" ) );

30    topitem->appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Find appropriate products" ) )
        << new MyStandardItem( KDGantt::TypeTask )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 1)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 3)) )
35        );

    topitem->appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Evaluate Product A" ) )
        << new MyStandardItem( KDGantt::TypeTask )
40        << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
        );

    topitem->appendRow( QList<QStandardItem*>()
45        << new MyStandardItem( QString( "Evaluate Product B" ) )
        << new MyStandardItem( KDGantt::TypeTask )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
        );

50    topitem->appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Evaluate Product C" ) )
        << new MyStandardItem( KDGantt::TypeTask )
```

```

55         << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
        );

topitem->appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Make decision" ) )
60         << new MyStandardItem( KDGantt::TypeTask )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 12)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
        );

65 model.appendRow( QList<QStandardItem*>()
        << topitem
        << new MyStandardItem( KDGantt::TypeSummary ) );

model.appendRow( QList<QStandardItem*>()
70         << new MyStandardItem( QString( "Decision made" ) )
        << new MyStandardItem( KDGantt::TypeEvent )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
        );

75 model.appendRow( QList<QStandardItem*>()
        << new MyStandardItem( QString( "Educate personel" ) )
        << new MyStandardItem( KDGantt::TypeTask )
        << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
80         << new MyStandardItem( QDateTime(QDate(2007, 3, 31)) )
        << new MyStandardItem( QString::null ) );

KDGantt::ConstraintModel cmodel;
QModelIndex pidx = model.index( 0, 0 );
85 cmodel.addConstraint(
        KDGantt::Constraint( model.index( 0, 0, pidx ),
                             model.index( 1, 0, pidx ) ) );
cmodel.addConstraint(
        KDGantt::Constraint( model.index( 0, 0, pidx ),
90                             model.index( 2, 0, pidx ) ) );
cmodel.addConstraint(
        KDGantt::Constraint( model.index( 0, 0, pidx ),
                             model.index( 3, 0, pidx ) ) );
cmodel.addConstraint(
95         KDGantt::Constraint( model.index( 1, 0, pidx ),
                             model.index( 4, 0, pidx ) ) );
cmodel.addConstraint(
        KDGantt::Constraint( model.index( 2, 0, pidx ),
                             model.index( 4, 0, pidx ) ) );
100 cmodel.addConstraint(
        KDGantt::Constraint( model.index( 3, 0, pidx ),
                             model.index( 4, 0, pidx ) ) );

KDGantt::DateTimeGrid grid;
105 grid.setDayWidth( 16 );

KDGantt::View view;
view.setGrid( &grid );
view.setModel( &model );
110 view.setConstraintModel( &cmodel );

```

```
view.setWindowTitle( "KDGantt - Extended Gantt Chart" );
view.show();

115     return app.exec();
    }
```

- ❶ To gather several tasks in one task, we use a `KDGantt::TypeSummary` entry, and add five rows of child entries to it. A summary entry does not need a start date and end date as these are automatically calculated from the children.
- ❷ To indicate that a deadline has been reached at a certain date, we use a `KDGantt::TypeEvent` in the Gantt chart. This has the same start date and end date and is shown as a diamond shape in the chart.
- ❸ The `KDGantt::ConstraintModel` class is used to set up constraints between entries. One single entry can depend on many entries, and vice versa—several entries can depend on one entry. This is done on the following lines, adding constraints between the `find` entry and the `evaluate` entries, and also from the `evaluate` entries to the `decision` entry. More on constraints can be found in Section , “Working With Constraints”.



What's Next

In the next chapter, we will go through the basic usage of KD Gantt including adding and customizing items and constraints...

Chapter 4. Basic Usage

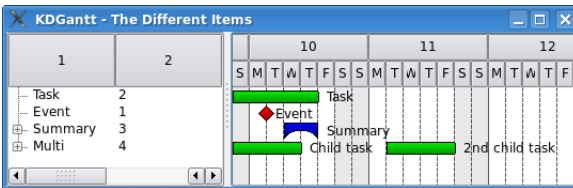
This chapter will focus on using KD Gantt with the `KDGantt::View` class. For more advanced concepts, such as working with the `KDGantt::GraphicsView` please see Chapter 5, *Advanced Usage*.

▶ Working With Items

Introduction to Items

There are three different items available in KD Gantt: Task, Event, and Summary items, represented by the enumeration values `KDGantt::TypeTask`, `KDGantt::TypeEvent`, `KDGantt::TypeSummary` and `KDGantt::TypeMulti`. There are also two special values, `KDGantt::TypeNone` and `KDGantt::TypeUser`. You can see a typical example of item usage in Figure 4.1.

Figure 4.1. The Different Items



These items are all created very similarly, it is just a matter of setting the correct type of the item. Each item is then given a name and, in the case of tasks and events, a start date and end date. Summary items do not have any dates set, as these are calculated from their children. Summary items are also not visible unless they have any visible children, such as tasks or events.

When adding several items at once, which could be the case when a file is loaded, redrawing the Gantt view after every addition might become very slow. To avoid this, you can use `KDGantt::View::setUpdatesEnabled(false)`, which will cause the update of the Gantt view to be suspended until you set the value to `true` again.

It is also possible to change the appearance of the different items, details and code examples are available in Section , “Customizing Items”.

Tasks

Task items are used to model the planned tasks in a Gantt chart. By default, they are drawn as a green rectangle which spans from the start date to the end date. If the time is not

specified, or if the start time equals the end time, nothing will be drawn in the schedule view, however the entry will still be visible in the tree view.

Events

Events are used to schedule certain events in, for example, a software project. Events need both a start time and an end time, but only the start time will be used for the drawing. The item itself is drawn as a red diamond shape, but both the brush and pen used to draw it can be changed, as is explained further in Section , “Customizing Items”.

Summaries

Summary items are used to summarize several tasks into one, and has calculates its time from its children's start/end times. Because of this, a summary item takes only a caption, but not a start date or an end date. If a summary item is added to the model, but never receives any children, it will not be visible in the schedule view, only in the tree view. Its children can be of any type, such as a task, an event or a summary.

By default, summary items are blue and drawn as a line with one triangle in each end.

Multi items

Multi items are similar to summary items as they both have several tasks as children and the start and end dates are calculated automatically. The difference is when using multi items, all the tasks are drawn on the same line in the graphics view.

Adding Items

The data for KD Gantt is stored in a model, and initially given to the view by using `KDGantt::View::setModel()`. When working with `KDGantt::View`, each row in the model represents an item in the Gantt chart, and the different columns are bound to store one type of data, as seen in Table 4.1 which, as a Gantt chart, would look similar to that in Figure 2.1.

Table 4.1. Item data table

| Caption | Item Type | Start Date | End Date | Completion Percentage |
|-----------------------|-------------------|---------------------|---------------------|------------------------------|
| Decide on new product | KDGantt::TypeTask | 2007-03-01 08:00 | 2007-03-13 17:00 | 20 |
| Educate staff | KDGantt::TypeTask | 2007-03-14 08:00 | 2007-03-31 17:00 | 0 |

The following code example comes from `step02a.cpp`, and the result is shown in Figure 4.1.

```
class MyStandardItem : public QStandardItem {❶
public:
MyStandardItem( const QVariant& v ) : QStandardItem()
{
setData( v, Qt::DisplayRole );
}
MyStandardItem( const QString& v ) : QStandardItem()
{
setData( v, Qt::DisplayRole );
}
};

int main( int argc, char* argv[] )
{
QApplication app( argc, argv );

QStandardItemModel model;
model.appendRow( QList<QStandardItem*>()❷
<< new MyStandardItem( QString( "Task" ) )
<< new MyStandardItem( KDantt::TypeTask )
<< new MyStandardItem( QDateTime( QDate( 2007, 3, 4 ) ) )
<< new MyStandardItem( QDateTime( QDate( 2007, 3, 9 ) ) ) );

model.insertRow( 1 );
model.setData( model.index( 1, 0 ), QString( "Event" ) );❸
model.setData( model.index( 1, 1 ), KDantt::TypeEvent );
model.setData( model.index( 1, 2 ), QDateTime( QDate( 2007, 3, 6 ) ) );
model.setData( model.index( 1, 3 ), QDateTime( QDate( 2007, 3, 6 ) ) );

// ...
}
```

- ❶ `MyStandardItem` is simply used for convenience when adding a row in one single call, instead of having to call `setData()` on five `QStandardItem` objects.
- ❷ Here, we add one entire Gantt chart item in one single call, by creating a list of `MyStandardItem` objects, each representing a specific data entry in the model.
- ❸ Another method of adding data would be to call `QAbstractItemModel::setData()` for each of the indices.

If you are adding child items to a summary item or multi item, simply use the model index of the summary or multi item as the parent index in the call to `QAbstractItemModel::index()`, instead of the default value. Or, if you are constructing your items from `QStandardItem` objects, use the object representing the summary or multi item, and call `QStandardItem::appendRow()`.

Customizing Items

There are two ways of customizing the appearance of items in KD Gantt, either using the already present `ItemDelegate` or implementing your own custom `ItemDelegate`

subclass. In this section we will cover the former; implementing your own custom `ItemDelegate` is covered in Section , “Creating Your Own `ItemDelegate`”.

Changing the Brush

Different types of items have different default colors. The initial default color is set to red for event items, green for tasks and blue for summaries.

The basic way of changing the color of an item type is to use a different `QBrush` when drawing it. This `QBrush` is then supplied to `KD Gantt` using `KDGantt::ItemDelegate::setDefaultBrush()` , which takes two arguments: the type for which you wish to change the brush, and the actual brush.

This will change the brush, and hence the color, for all items of a certain type, including already created items.

This code example comes from `step02b.cpp`, and the result is shown in Figure 4.2.

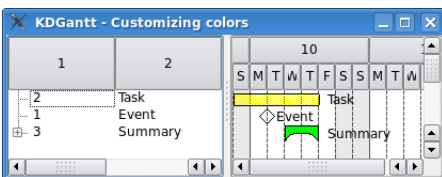
```
// Added a task, an event and a summary item, with a child

QBrush myTaskBrush( Qt::yellow, Qt::Dense3Pattern );❶
QBrush myEventBrush( Qt::NoBrush );
QBrush mySummaryBrush( Qt::green );

view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeTask,❷
myTaskBrush );
view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeEvent,
myEventBrush );
view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeSummary,
mySummaryBrush );
```

- ❶ Setting up the brushes for customizing the different items. The tasks will be yellow, with a moderately dense pattern, the events will be transparent while the summary items will be green.
- ❷ Setting the default brush on the `ItemDelegate`. As mentioned, this will change the brush for all existing items of this particular type.

Figure 4.2. Customizing the Brush



Changing the Pen

Just as with item colors, you can change the line style for each item type, simply by providing `KDGantt::ItemDelegate` with a `QPen` object. The default pen for all items is black, with a width of 1 pixel.

To change the pen for an item type, use `KDGantt::ItemDelegate::setDefaultPen()`. This method takes two arguments: the item type and the `QPen` object.

Note

Changing the pen for an item not only changes the line color in the schedule view, it also changes the color of the caption text in the schedule view, as these are drawn using the same pen.

This code example comes from `step02c.cpp`, and the result is shown in Figure 4.3.

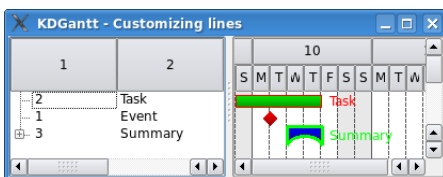
```
// Added a task, an event and a summary item, with a child

QPen myTaskPen( Qt::red ); ❶
QPen myEventPen( Qt::NoPen );
QPen mySummaryPen( Qt::green, 3 );

view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeTask, ❷
myTaskPen );
view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeEvent,
myEventPen );
view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeSummary,
mySummaryPen );
```

- ❶ Setting up the pens for customizing the different items. The tasks will be drawn with a red line, the events will be drawn without line while the summary items will have a green line with a width of 3 pixels.
- ❷ Setting the default pen on the `ItemDelegate`. As you can see in the screenshot below, this also affects the caption in the schedule view for each of the items.

Figure 4.3. Customizing Lines



Start and End Time

Changing the start and end time of an item is simply a matter of modifying the data within the model, specifically in the third and fourth column. This is done by using

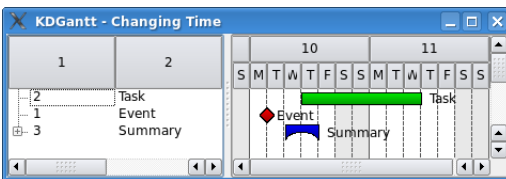
`QAbstractItemModel::setData()`, and giving it the index for the start or end date entries, as well as the new date.

As mentioned earlier in this manual, the start date and end date for summary items are calculated automatically and should not be set. Also, events only take the start date into consideration when drawing, although they still need a valid end date. Hence, the only items that require both a start date and an end date are task items.

The complete source code for this example is in `step02d.cpp`. The result can be seen in Figure 4.4, and as seen compared to Figure 4.1 the task item has been moved and extended.

```
// A task added to the first row of the model
model.setData( model.index( 0, 2 ), QDateTime( QDate( 2007, 3, 8 ) ) );
model.setData( model.index( 0, 3 ), QDateTime( QDate( 2007, 3, 15 ) ) );
```

Figure 4.4. Customizing Start and End Times



Working With Constraints

Constraints are most often used to show dependencies between tasks. We saw an example of this in the introductory chapters, in Figure 2.2. Using constraints makes it easier for the reader to see quickly which tasks need to be completed before the next task can be started.

In this section, we will try to explain this a bit further, by means of a few code examples on how you can use constraints.

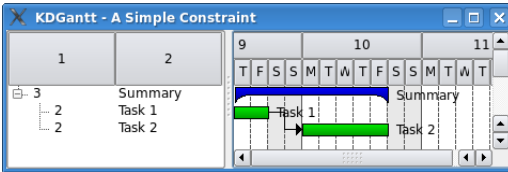
Introduction To Constraints

A constraint is an object of `KDGantt::Constraint`, containing references to the indices of the two tasks it is linking together. One `KDGantt::Constraint` can only link two tasks, however, several `KDGantt::Constraints` can reference the same tasks at the same time. This enables one task to be dependant on several others, and several tasks to be dependant on one task.

To have KD Gantt observe a constraint, the `KDGantt::Constraint` object must be added to a `KDGantt::ConstraintModel`.

The example in Figure 4.5 shows links between items of the type `KDGantt::TypeTask` but it works just the same with items of the types `KDGantt::TypeEvent` and `KDGantt::TypeSummary`.

Figure 4.5. A Simple Constraint



Adding Constraints

As mentioned in the introduction, constraints are objects of `KDGantt::Constraint` added to a `KDGantt::ConstraintModel` using `KDGantt::ConstraintModel::addConstraint()`. This constraint model is then passed into the view by calling `KDGantt::View::setConstraintModel()`.

The `KDGantt::Constraint` object is made up of two `QModelIndex`'s, each representing an item in the Gantt chart with the second index depending on the first index.

The complete source code for this example is in `step02e.cpp`. The result can be seen in Figure 4.5

```
QStandardItemModel model;

// Added summary on index 0,0

// Added two tasks on index 0,0 and 0,1 as the children of the summary

KDGantt::ConstraintModel cmodel;
QModelIndex pidx = model.index( 0, 0 );
cmodel.addConstraint( KDGantt::Constraint( model.index( 0, 0, pidx ),
model.index( 1, 0, pidx ) ) );

view.setModel( &model );
view.setConstraintModel( &cmodel );
```

Customizing Constraints

To customize the look of the constraints you will need to implement your own `KDGantt::ItemDelegate` and override `paintConstraintItem()`. More on how to do this is covered in Section , “Creating Your Own ItemDelegate”.

Working With the Grid

There are a number of options you can change in the grid, the most important being the scale. There are three different values for the scale, `ScaleAuto`, `ScaleHour` and `ScaleDay`. Changing the scale does not change the width of the days, but rather the second row in the header. As can be seen in Figure 4.6 and Figure 4.7 the width stays the same although we changed the scale.

Figure 4.6. Using Day Scale

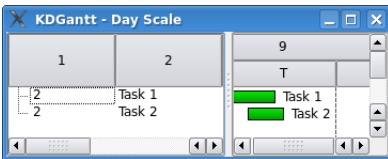
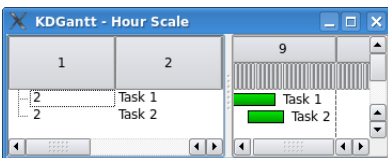


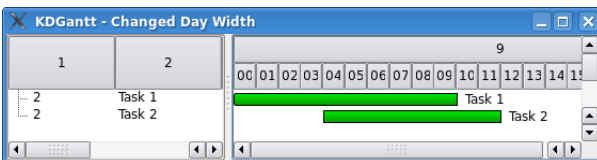
Figure 4.7. Using Hour Scale



Changing the width of the days would enable the user to see the items in the chart with finer granularity than before. Instead of looking at an item spanning 2 hours of a day from a view where you can see an entire week, you can, by increasing the day width, see exactly during which hours the item takes place. This is done by calling `KDGantt::DateTimeGrid::setDayWidth()`, giving it a value representing the actual pixel width of a day.

The screenshot shown in Figure 4.8 is based on the same data as the one in Figure 4.7, and is also using the hour scale. However, the day width has been changed to 500 pixels, making it possible to see at which hours the tasks begin and end.

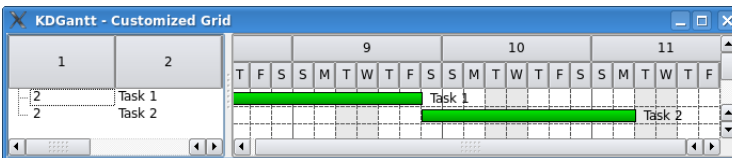
Figure 4.8. Changed Day Width



You can also modify the week shown in the grid, both by setting the start day of the week as well as setting which days are free. The former is done

by calling `KDGantt::DateTimeGrid::setWeekStart()`, and the latter by calling `KDGantt::DateTimeGrid::setFreeDays()`. The start day of the week will have a solid left line drawn in the grid, instead of a dashed line which is the case for all other week days. Also, quite naturally, the week scale will use it as week delimiter. The free days will be drawn with a grey background instead of the default white.

Figure 4.9. Customized Grid



The following code example can be found in `step02f.cpp`, and the results can be seen in Figure 4.9.

```
// ... set up the model, added two tasks

KDGantt::DateTimeGrid grid;
grid.setScale( KDGantt::DateTimeGrid::ScaleDay );
grid.setDayWidth( 20 );
grid.setWeekStart( Qt::Sunday );
grid.setFreeDays( QSet<Qt::DayOfWeek>() << Qt::Tuesday << Qt::Wednesday );
grid.setRowSeparators( true );

// ... set up the view
```

User Interaction

KD Gantt has a set of built-in user interactions. For example, a user can click and drag an item in the chart to move it, or grab either end of it to change the start date or end date. It is also possible to create a constraint between two items by clicking-and-dragging.

Besides these default implementations, you can of course add your own user interactions by connecting to various signals in `KDGantt::GraphicsView` and `KDGantt::GraphicsScene`. Please refer to the reference manual for details on each class.

Next Chapter

In the next chapter, we will go into the internals of KD Gantt with a look at `GraphicsView`, as well as on how to implement your own `ItemDelegate` for greater control over how the items look.

Chapter 5. Advanced Usage

This chapter will give you a more in-depth knowledge of KD Gantt, and how you can work with it to fit your needs. We will also introduce a new way of adding items to the Gantt chart, rather than using `KDGantt::View`.

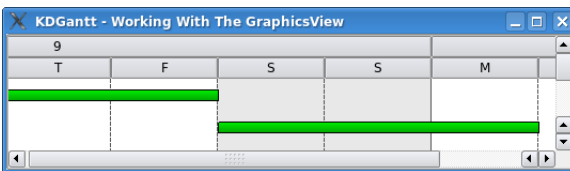
Furthermore, this chapter will show you how you can modify all items in the Gantt chart more extensively than offered by the available API.

Working With The GraphicsView

The `KDGantt::GraphicsView` is the component visible to the right in all screenshots so far, where all the tasks, events and summaries are drawn. You are able to use this directly instead of using `KDGantt::View`, but this introduces some new concepts when it comes to adding items to the Gantt chart. You also have greater responsibility to make sure everything needed is initialized, as this was handled by the `KDGantt::View` earlier.

As described in Chapter 4, *Basic Usage*, when adding items to the view you need to make sure the correct data is put into the correct column of the model. This is not the case when using `KDGantt::GraphicsView` directly, now all data goes to index 0,0 but with different data item roles. The item roles used are `ItemTypeRole`, `StartTimeRole`, `EndTimeRole` and `TaskCompletionRole` for each type of data.

Figure 5.1. Only Using GraphicsView



The following code example is also available in `step03a.cpp`. The result can be seen in Figure 5.1

```
1
#include <QApplication>
5 #include <QStandardItemModel>
#include <QPointer>
#include <KDGanttGraphicsView>
#include <KDGanttDateTimeGrid>
#include <KDGanttAbstractRowController>
10
class MyRowController : public KDGantt::AbstractRowController {
private:
    static const int ROW_HEIGHT ;
```

```

    QPointer<QAbstractItemModel> m_model;
15
public:
    MyRowController()
    {
    }
20
    void setModel( QAbstractItemModel* model )
    {
        m_model = model;
    }
25
    /*reimp*/int headerHeight() const { return 40; }

    /*reimp*/ bool isRowVisible( const QModelIndex& idx ) const
    { return true;}
30
    /*reimp*/ KDGantt::Span rowGeometry( const QModelIndex& idx ) const
    {
        return KDGantt::Span( idx.row()*ROW_HEIGHT, ROW_HEIGHT );
    }
    /*reimp*/ int maximumItemHeight() const {
35
        return ROW_HEIGHT/2;
    }

    /*reimp*/ QModelIndex indexAt( int height ) const {
40
        return m_model->index( height/ROW_HEIGHT, 0 );
    }

    /*reimp*/ QModelIndex indexBelow( const QModelIndex& idx ) const {
        if ( !idx.isValid() ) return QModelIndex();
        return idx.model()->index(
45
            idx.row()+1, idx.column(), idx.parent() );
    }
    /*reimp*/ QModelIndex indexAbove( const QModelIndex& idx ) const {
        if ( !idx.isValid() )return QModelIndex();
        return idx.model()->index(
50
            idx.row()-1, idx.column(), idx.parent() );
    }

};

55 const int MyRowController::ROW_HEIGHT = 30;

int main( int argc, char* argv[] )
{
    QApplication app( argc, argv );
60
    QStandardItemModel model;

    QStandardItem* item = new QStandardItem();
    item->setData( KDGantt::TypeTask, KDGantt::ItemTypeRole );
    item->setData( QString( "Decide on new product" ) );
65
    item->setData(
        QDateTime( QDate( 2007, 3, 1 ) ), KDGantt::StartTimeRole );
    item->setData(
        QDateTime( QDate( 2007, 3, 3 ) ), KDGantt::EndTimeRole );
70
    QStandardItem* item2 = new QStandardItem();

```

```

    item2->setData( KDGantt::TypeTask, KDGantt::ItemTypeRole );
    item2->setData( QString( "Educate personel" ) );
    item2->setData(
75     QDateTime( QDate( 2007, 3, 3 ) ❸ ), KDGantt::StartTimeRole );
    item2->setData(
        QDateTime( QDate( 2007, 3, 6 ) ), KDGantt::EndTimeRole );

    model.appendRow( item );
80    model.appendRow( item2 );

    MyRowController rowController;
    rowController.setModel( &model );

85    KDGantt::GraphicsView graphicsView;
    graphicsView.setRowController( &rowController );
    graphicsView.setModel( &model );

    graphicsView.setWindowTitle(
90     "KDGantt - Working With The GraphicsView" );
    graphicsView.show();

    return app.exec();
}
95

```

- ❶ One important part of setting up the `KDGantt::GraphicsView` is that you need to implement your own row controller as a subclass of `KDGantt::AbstractRowController`. The row controller is used by `KDGantt::GraphicsView` to navigate through the data model and also to determine the row geometries. For more information on each method you will need to override, please refer to the reference manual.
- ❷ Two new items are created and added to the model. As you can see, we only need to create one `QStandardItem` object per item in the Gantt chart with all the data in different roles, compared to when using `KDGantt::View` where we needed four or five `QStandardItem` objects for each item in the Gantt chart.
- ❸ We then create our `KDGantt::GraphicsView` and set the required member variables on it—the model containing the items to be drawn as well as the custom row controller.

Setting Up The Grid

Working with the grid on a standalone `KDGantt::GraphicsView` is no different from what we have described in Section , “Working With the Grid” , except that instead setting it up with `KDGantt::View::setGrid()`, you have to use `KDGantt::GraphicsView::setGrid()`.

Warning

You should never need to call either

```
KDGantt::AbstractGrid::setModel() or
```

```
KDGantt::AbstractGrid::setRootIndex() from client code. These  
are for internal use only. Doing so will cause an assert in client code.
```

A complete code example where the grid is used with a standalone `KDGantt::GraphicsView` can be found in `step03b.cpp`.

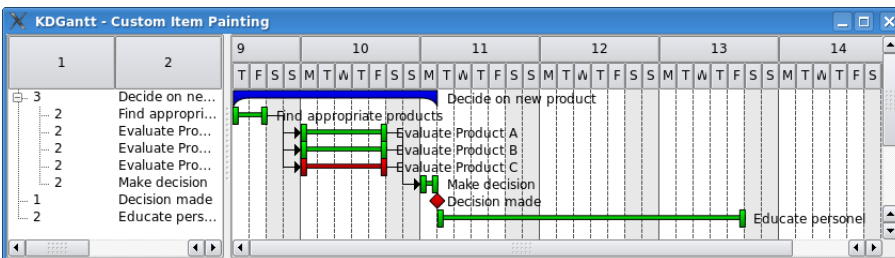
Creating Your Own `ItemDelegate`

In order to customize items and constraints more extensively than offered by the basic API you will need to implement your own custom `ItemDelegate`. This is then installed on the graphics view by using `KDGantt::GraphicsView::setItemDelegate()`. In this section, we will have a closer look on exactly what you can change with your own `ItemDelegate`, as well as show some code examples doing just that.

Customizing Items

In Section , “Customizing Items”, we have shown how you can change the brush and pen for all the items in the Gantt chart. By overriding `KDGantt::ItemDelegate::paintGanttItem()` you can customize the look of your items even further as you are then responsible for painting them.

Figure 5.2. Custom Item Painting



In the following code example, we change the look of tasks, while events and summaries are painted by the base class. For the complete code, please refer to `step03c.cpp`. The result can be seen in Figure 5.2.

```
using namespace KDGantt;  
class MyCustomItemDelegate : public ItemDelegate {  
void paintGanttItem( QPainter* painter, ❶  
const StyleOptionGanttItem& opt,  
const QModelIndex& idx )  
{
```

```

painter->setRenderHints( QPainter::Antialiasing );
if ( !idx.isValid() ) return;
ItemType type = static_cast<ItemType>(
idx.model()->data( idx, ItemTypeRole ).toInt() );
QString txt = idx.model()->data( idx, Qt::DisplayRole ).toString();
QRectF itemRect = opt.itemRect;
QRectF boundingRect = opt.boundingRect;
boundingRect.setY( itemRect.y() );
boundingRect.setHeight( itemRect.height() );

QBrush brush = defaultBrush( type );
if ( opt.state & QStyle::State_Selected ) {
QLinearGradient selectedGrad( 0., 0., 0.,
QApplication::fontMetrics().height() );
selectedGrad.setColorAt( 0., Qt::red );
selectedGrad.setColorAt( 1., Qt::darkRed );

brush = QBrush( selectedGrad );
}

painter->setPen( defaultPen( type ) );
painter->setBrush( brush );
painter->setBrushOrigin( itemRect.topLeft() );

switch( type ) {
case TypeTask:
if ( itemRect.isValid() ) {
QRectF r = itemRect;
r.translate( 0., r.height() / 3. );
r.setHeight( 2. * r.height() / 9. );

painter->translate( 0.5, 0.5 );
painter->drawRect( r );

QRectF leftRect = itemRect;
leftRect.setWidth( 5 );
painter->drawRect( leftRect );

QRectF rightRect = itemRect;
rightRect.setWidth( 5 );
rightRect.translate( r.width() - 5, 0 );
painter->drawRect( rightRect );

Qt::Alignment ta;
switch( opt.displayPosition ) {
case StyleOptionGanttItem::Left: ta = Qt::AlignLeft; break;
case StyleOptionGanttItem::Right: ta = Qt::AlignRight; break;
case StyleOptionGanttItem::Center: ta = Qt::AlignCenter; break;
}
painter->drawText( boundingRect, ta, txt );
}
break;
default:
KDGantt::ItemDelegate::paintGanttItem( painter, opt, idx );
break;
}
};

```

- ❶ As we are changing the look of Gantt items, we need to override `paintGanttItem()`. If we were to change the appearance of constraints, we would have to override `paintConstraintItem()` instead.
- ❷ One thing that we changed compared to the default painting of the task items, is that selected items are filled with a red gradient instead of having the line width doubled.
- ❸ The second thing we changed was the actual look of the task items. By default, they are drawn as a rectangle spanning from the start date to the end date. This is still the case, however we have decreased the height of the rectangle, and added two vertical rectangles at the beginning and end of the task, to symbolize the fact that you, as a user, can click and drag the edge of the rectangle to change the start date and end date.

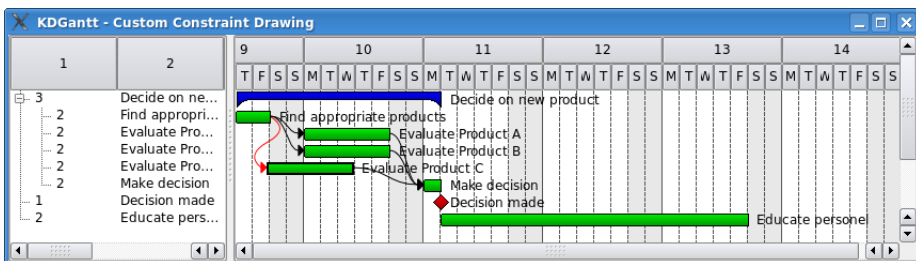
The process of customizing the appearance of summary and event items are of course similar, it is up to your imagination what they can look like.

Customizing Constraints

As mentioned in the previous section, it is also possible to change the appearance of the constraints. This is achieved by overriding `KDGantt::ItemDelegate::paintConstraintItem()`, providing your own code for drawing it.

The default look of a constraint is an arrow drawn from the end of one item to the beginning of another, as seen in e.g. Figure 5.2. In the following example, we will change the way the connection is drawn from straight lines to a Bezier curve. The complete code example is available in `step03d.cpp` and the results can be seen in Figure 5.3. Please note that the "Evaluate Product C" task has been moved manually to demonstrate what the constraint looks like when the start and end times overlap.

Figure 5.3. Custom Constraint Drawing



```
class MyCustomItemDelegate : public ItemDelegate {
    static const qreal TURN = 10.;

    void paintConstraintItem( QPainter* painter, ❶
        const QStyleOptionGraphicsItem& opt,
        const QPointF& start, const QPointF& end )
    {
```

```

Q_UNUSED( opt );
painter->setRenderHints( QPainter::Antialiasing );
qreal midx = ( end.x()-start.x() )/2. + start.x();
qreal midy = ( end.y()-start.y() )/2. + start.y();

if ( start.x() <= end.x() ) {
painter->setPen( Qt::black );
painter->setBrush( Qt::black );
} else {
painter->setPen( Qt::red );
painter->setBrush( Qt::red );
}
if ( start.x() > end.x()-TURN ) {
QPainterPath path( start );❷
path.quadTo( QPointF( start.x() + TURN * 2., ( start.y() + midy ) / 2. ),
QPointF( midx, midy ) );
path.quadTo( QPointF( end.x() - TURN * 2., ( end.y() + midy ) / 2. ),
QPointF( end.x() - TURN / 2., end.y() ) );

QBrush brush = painter->brush();
painter->setBrush( QBrush() );
painter->drawPath( path );
painter->setBrush( brush );

QPolygonF arrow;
arrow << end
<< QPointF( end.x()-TURN/2., end.y()-TURN/2. )
<< QPointF( end.x()-TURN/2., end.y()+TURN/2. );
painter->drawPolygon( arrow );
} else {
painter->setBrush( QBrush() );
QPainterPath path( start );❷
path.cubicTo( QPointF( midx, start.y() ),
QPointF( midx, end.y() ),
QPointF( end.x()-TURN/2., end.y() ) );
painter->drawPath( path );

painter->setBrush( Qt::black );

QPolygonF arrow;
arrow << end
<< QPointF( end.x()-TURN/2., end.y()-TURN/2. )
<< QPointF( end.x()-TURN/2., end.y()+TURN/2. );
painter->drawPolygon( arrow );
}
};

```

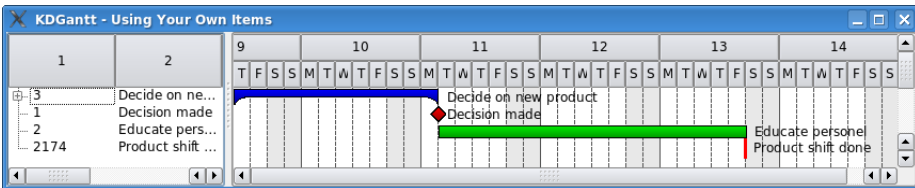
- ❶ When customizing the appearance of constraints, you will need to override `KDGantt::ItemDelegate::paintConstraintItem()`.
- ❷ What we have changed here was the drawing of the actual constraint link. We have set it up for two situations: when the start time and end time of the two tasks overlap, and when they do not. The painter path is then simply drawn using `QPainter::drawPath()`.

Creating Your Own Items

Besides changing the appearance of items and constraints, implementing your own `ItemDelegate` also has the advantage that you can add your own custom items to the Gantt chart. This is done similarly to customizing the items, but instead of changing the way an already existing item type is drawn, you simply look for your own type. This type must have a value greater than `KDGantt::TypeUser`, as the values below are reserved for KD Gantt.

The following example shows how you can create a fourth item type, `Deadline`, used for showing a deadline in the Gantt chart. The complete code example can be found in `step03e.cpp` and the results can be seen in Figure 5.4.

Figure 5.4. Using Your Own Items



```
enum MyItemType {  
TypeDeadline = KDGantt::TypeUser + 1174  
};  
  
using namespace KDGantt;  
class MyCustomItemDelegate : public ItemDelegate  
{  
void paintGanttItem( QPainter* painter,  
const StyleOptionGanttItem& opt,  
const QModelIndex& idx )  
{  
if ( !idx.isValid() ) return;  
ItemType type = static_cast<ItemType>(  
idx.model()->data( idx, ItemTypeRole ).toInt() );  
QString txt = idx.model()->data( idx, Qt::DisplayRole ).toString();  
QRectF itemRect = opt.itemRect;  
QRectF boundingRect = opt.boundingRect;  
boundingRect.setY( itemRect.y() );  
boundingRect.setHeight( itemRect.height() );  
  
switch( type ) {  
case TypeDeadline:  
if ( opt.boundingRect.isValid() ) {  
QPen pen( Qt::red );  
pen.setWidth( 3 );  
painter->setPen( pen );  
QRect r = opt.rect;  
QLineF line( 0., 0., 0., r.height() );  
painter->drawLine( line );  
}}
```

```

painter->setPen( Qt::black );
Qt::Alignment ta;
switch( opt.displayPosition ) {
case StyleOptionGanttItem::Left: ta = Qt::AlignLeft; break;
case StyleOptionGanttItem::Right: ta = Qt::AlignRight; break;
case StyleOptionGanttItem::Center: ta = Qt::AlignCenter; break;
}
painter->drawText( boundingRect, ta, txt );
}
break;
default:
KDGantt::ItemDelegate::paintGanttItem( painter, opt, idx );
break;
}
}
};

```

- ❶ We declare our new type which is later used to distinguish between this type and the types supplied by KD Gantt.
- ❷ When we discover that the painting code for our item is called, we simply draw it—in this case a 3 pixel thick vertical red line to show a deadline in the Gantt chart. The line takes up all the vertical space on the row it is set.