Trygve Reenskaug[1]

MVC and DCA
Demo program comments

VERY EARLY DRAFT, VERSION = DATE

Abstract. My MVC (Model-View-Controller) is a well known architecture for bridging the gap between the user's mental model and a computer. MVC supports the user illusion of working directly with the domain information (models) as seen in different perspectives and presented in different views. All I/O code is pulled out of the model programs; making them simpler and more readable. The MVC is particularly applicable to situations with complex information and knowledgeable users.

My DCA (Data-Collaboration-Algorithm) is a new architecture for creating simple solutions to complex problems. It uses a divide and conquer approach to hide object substructures within components that appear as regular objects in their environment. The DCA architecture further simplifies the internals of complex components; data and collaboration aspects are refined and made explicit in specialized data and interaction objects. The result is disentangled and more intelligible code with domain objects being purified to implement the essential algorithms only.

The *BabyProject-3 demo Applet* and the *BabyProject-3 demo code* can be found at
http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/

1 The BabyUML Project 3
2 The Demo Example 3
3 MVC: The Model-View-Controller paradigm 5
3.1 The Model 5
3.2 The View 6
3.3 The Controller 6
3.3.1 Discussion: The Recursive Controller6
3.3.2 Discussion: Smalltalk-80 Controller is different6
3.3.3 Discussion: Some "Controllers" are perverse7
3.4 The anatomy of the demo user interface 7
3.5 Controller coordinates selection 8
3.5.1 Discussion: Observer pattern or direct View access from Controller?10
3.5.2 Discussion: The panels could be subordinate controllers.10
4 DCA: The Data-Collaboration-Algorithm paradigm 10
4.1 The Model as a single object 10
4.2 The DCA Component; a well-structured monster object. 11
4.3 The demo Model as a DCA component 12
4.4 The Data part defined by its schema 13
4.5 Example 1: Panel layout 14
4.6 Example 2: Frontloading 15
4.6.1 FrontloadCollab, the frontloading collaboration16
4.6.2 FrontloadAlgorithm, implementing the frontloading interaction16
5 Summary and Discussion. 17
5.1 The MVC paradigm 17
5.2 The DCA paradigm 18
5.3 Where are we? 18
5.4 What's next? 19
6 Acknowledgements 20
7 References. 20
appendix 1The NetBase code 22
appendix 2The DependencyView class 24

---

1. Department of Informatics, University of Oslo, Norway

appendix 3The RankedCollab class 26
appendix 4The FrontloadAlgorithm class. 29
appendix 5The FrontloadCollab class 30

1 The BabyUML Project

Once in a while, I have written object oriented software that made me feel proud. But I always seem to be skating on thin ice. An object is embedded in a two-dimensional structure: It is connected to other objects by links that may be well hidden within its code. And the code is distributed between class and superclasses. There is no support for clustering so objects exist in a featureless landscape. I can study every detail, but I cannot see the whole. This is OK with small and insignificant programs, but my programming style does not scale and the complexity of major programs bogs me down.

The goal of the BabyUML project is to give me better confidence in my programs. I want to be able to write a piece of code and give it to a colleague so that she can audit it and take responsibility for its correctness. I want my code to be effectively chunked and self documenting so that other people can read it and grasp its architecture and its operation.

The BabyUML project slogan is a quote from Hoare's 1980 Turing award lecture:

> *The price of reliability is the pursuit of the utmost simplicity.*[Hoare]

The BabyUML project shall give me a high level programming discipline that facilitates compact and simple descriptions of major systems of collaborating objects. Simplicity is the overriding concern. Lines of code and even efficiency come second. Gerald Weinberg[Wein] argues that while low level code can be very efficient in the details, high level code can be superior because it is more readable and makes it easier to find efficient structures. Further, high level code is also easier to change to reflect new insights.

The essence of object orientation is that objects collaborate to reach a common goal. Important questions that need to be answered are "what are the objects", how are they interconnected and how do they interact. None of these questions are answered explicitly by low level object languages such as Simula, Smalltalk, and Java. In the BabyUML project, I search for constructs that remedy this deficiency.

The BabyUML project is essentially experimental. This note is a discussion of a particular Java experiment. Chapter 2 introduces the example problem. The problem solution combines two paradigms, MVC and CDA. MVC is the Model-View-Controller paradigm discussed in chapter 3. DCA is the Data-Collaboration-Algorithm paradigm discussed in chapter 4. I finally sum up what I have learned from the experiment in chapter 5.

2 The Demo Example

Project planning and control is frequently based on the idea of *activity networks*. A piece of work that needs to be done is described as an activity. The work done by an architect when designing a house can be broken down into activities. The work of erecting the house likewise. Example activities: digging the pit, making the foundation, erecting the frame, paneling the walls, painting these walls.

An activity is characterized by its *name*, its *duration*, its *earlyStart and earlyFinish times*, its *lateStart and finish times*, a set of *predecessor* activities, and a set of *successor* activities. Predecessors and successors are called *technological dependencies.* An activity can start when all its predecessors are finished, and a successors cannot start before the current activity is finished. There are more sophisticated forms of technological dependencies. For example, it is possible to start the painting of one wall before the panelling of all walls is finished. Such cases are catered for with various kinds of *activity overlap*.

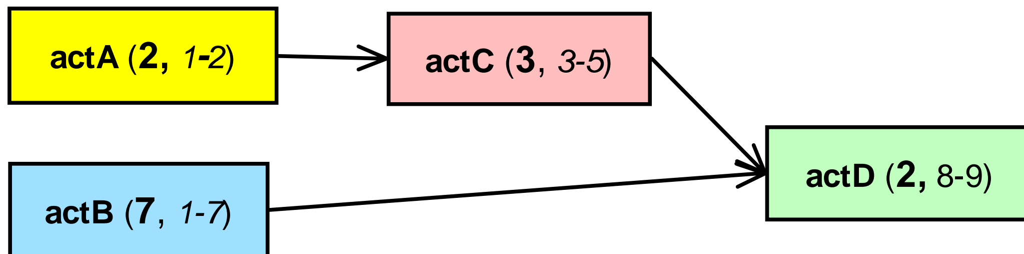
A *start activity* is an activity with no predecessors. *Frontloading* is the calculation of the *early start and finish times* of each activity given the start time of the start activities. Similarly, an *end activity* is an

activity with no successors. *Backloading* is the calculation of the *late start and finish times* of each activity given the end time for the end activities.

Activities may also be tied to *resources*. The creation of a design drawing requires some hours of work by an architect and a draftsman. The digging of the pit requires machinery and the efforts of some navvies. Resource allocation is to reserve resources for each activity. A scarce resource may delay the whole project. Resource allocation is an extension of the basic idea of activity networks. It is a non-trivial operation; one can easily end up with unimportant activities blocking the progress of critical ones. (We cannot dig the pit because the navvies are busy levelling the garden.)

The example chosen for this demo experiment is the rudimentary activity network shown in figure 1. The activity *duration*, *earlyStart* and *earlyFinish* times are shown in parenthesis. There is a single resource; say a pool of workers. It has unlimited capacity and an activity employs a single worker for its duration.

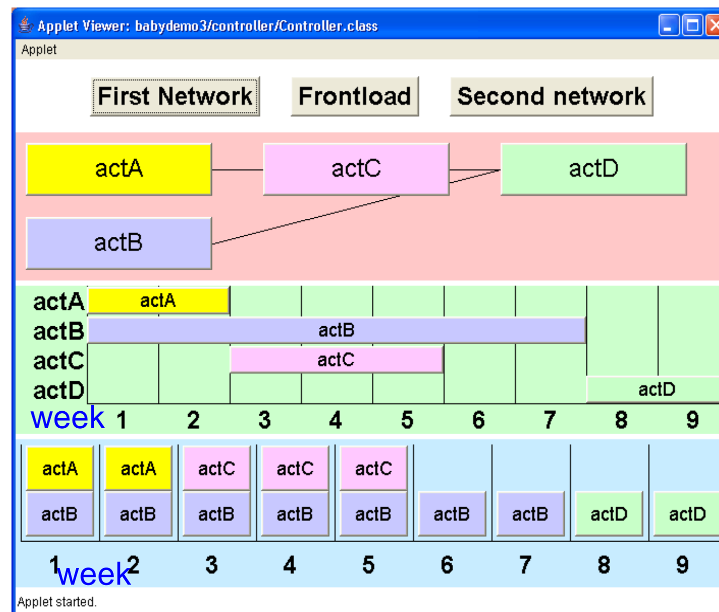
Fig. 1: The demo activity network.



The demo program GUI is shown in figure 2. It's partitioned into four strips. The top strip has three command buttons: Create *First network* (the one shown in figure 1). *Frontload* the network and allocate resources. Create *Second network*. The second strip shows the dependency network. The third strip is a gantt diagram showing when the different activities will be performed. The fourth and bottom strip shows how the activities are allocated to the resource.

This demo could be programmed in many different ways. I use it to illustrate the MVC and DCA paradigms, pretending that I'm working on a comprehensive planning system.

Fig. 2: The demo program user interface.



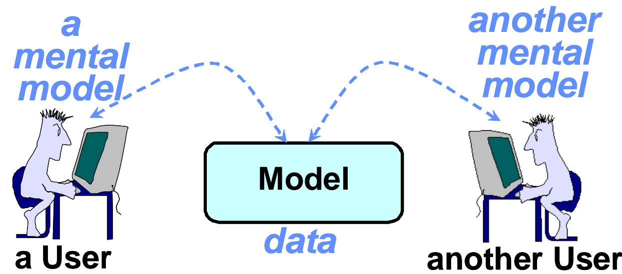
3 MVC: The Model-View-Controller paradigm

MVC was first conceived as a means for giving human users control of the computer resources. How can the user experience a system that feels like an extension of his own brain? How can we put the user in the driver's seat so that he can not only run the program but also understand and even modify its operation? How can we structure the system so that each user sees an image of the world that exactly corresponds to his own conception of it?

The first version of the MVC was created as a first step towards a solution. The domain was shipbuilding. The problem was project planning and control as described above. A manager was responsible for a part of a large project. His department had its own bottlenecks and its own considerations for resource allocations. Other departments were different; a pipe shop was very different from a panel assembly line. How could each manager have his own specialized part of the planning system while we simultaneously preserved the integrity of the system as a whole?

MVC was conceived to bridge the gap between the mind of the user and the information stored in the computer. The idea is illustrated in figure 3.

Fig. 3: Bridge the gap between the user's mind and the stored data.



3.1 The Model

The terms *data* and *information* are commonly used indiscriminately so that they are almost synonymous. In the stone age, IFIP defined them precisely in a way that I still find very fruitful when thinking about the human use of computers[IFIP]:

> *DATA. A representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process.*
>
> *Note: The representation may be more suitable either for human interpretation (e.g., printed text) or for internal interpretation by equipment (e.g., punched cards or electrical signals).*
>
> *INFORMATION. In automatic data processing the meaning that a human assigns to data by means of the known conventions used in its representation.*
>
> *Note: The term has a sense wider than that of information theory and nearer to that of common usage.*[IFIP]

So the user's mental model is *information*, information as defined does not exist outside the human brain. But *representation of information* can and do exist outside the brain. It is called *data*.

In the demo example, the Model is the data representing the demo activity network and the demo resource. The Model data may be considered *virtual* because it needs to be transformed to be observable.

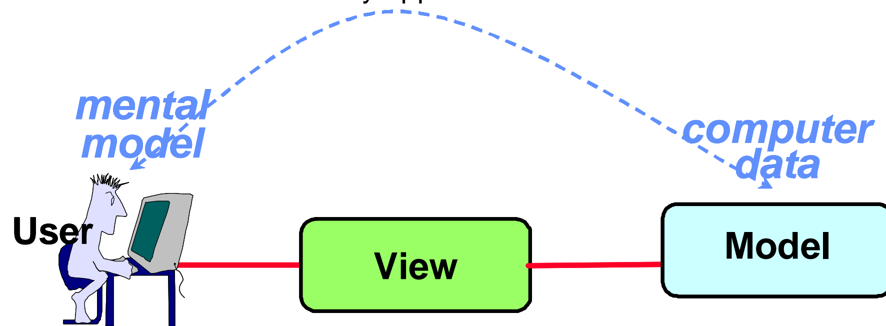
We will discuss the interaction between the Model and the View-Controller pair in section 4 on page 10.

3.2 The View

The View transforms the virtual Model data into a form that the human can convert into *information* as illustrated in figure 4.

If a common class for model and view is complex, the program will often be simpler if the model and view are separate classes. And if the users need to see the model data in different perspectives, then separating out several view classes is almost necessary.

Fig. 4: The View couples model data to the information in the user's brain so that they appear fused into one.

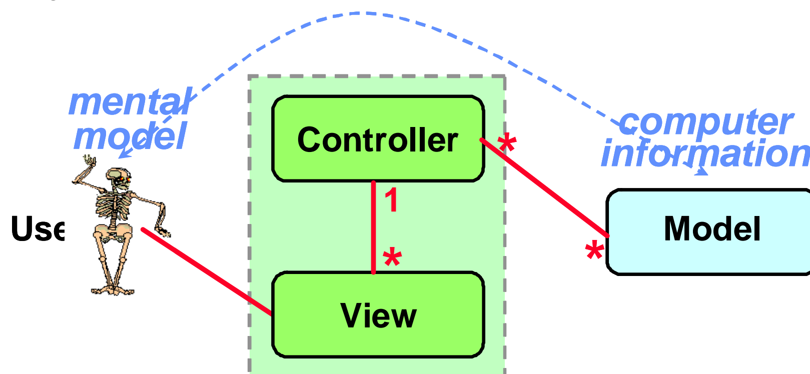


I will discuss our demo implementation in section 4.5 on page 14.

3.3 The Controller

The Controller is responsible for creating and coordinating a number of related Views. I sometimes think of it as a Tool that the user employs to work with the system's latent information.

Fig. 5: The Controller creates and coordinates multiple Views



3.3.1 Discussion: The Recursive Controller

A controller can also be a view. In this case, the three panels could be views under the main tool controller. They could also be controllers of their activityViews. We did not introduce such recursion here because it seemed unduly complex for this simple problem.

3.3.2 Discussion: Smalltalk-80 Controller is different

Note that the Smalltalk 80 Controller is responsible for input and thus different from the one discussed here.

3.3.3 Discussion: Some “Controllers” are perverse

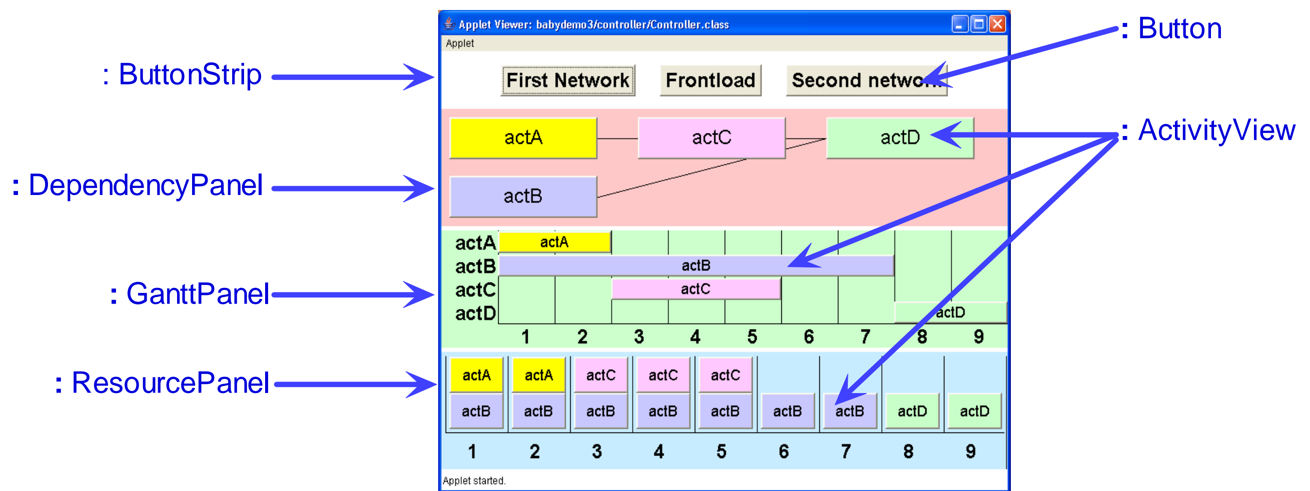
Also note that some so-called MVC structures let the controller control the user interaction and thus, the user. This whole idea is fundamentally different from MVC as described here. I want the user to be in control and the system appear as an extension of the user’s mind. In short, I want the “main program” of the interaction to be in the head of the user. In the alternative “MVC”, the computer is in control and the system appears as an enforcer of company procedures. In short, the “main program” of the interaction is in the computer.

3.4 The anatomy of the demo user interface

The demo GUI is shown in figure 6. We see that the tool is divided into 4 strips:

1. The top strip contains command buttons. They are not part of the MVC and will not be discussed further.
2. The second strip is the dependencyPanel; it is a view that shows the activities with their technological dependencies.
3. The third strip is the gantt panel; it is a bar chart showing the time period for each activity.
4. The fourth strip is a resource panel; it shows the activities that are allocated to the resource in each time interval.

Fig. 6: The anatomy of the MVC demo tool.
(: ButtonStrip means an instance of class ButtonStrip).

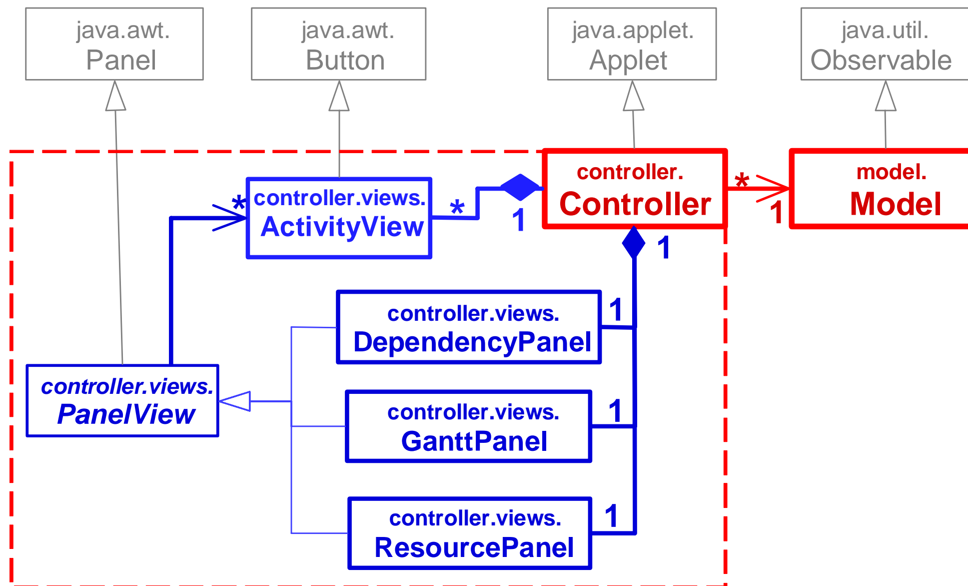


An overview of the implementation is shown in the class diagram of figure 7. We see the classes described above and their main associations.

In my traditional programming style, the views would all be associated with the model. In this implementation, I reduce the number of associations in order to get a simpler and cleaner structure. The views are now subordinated the controller by being enclosed in a controller-managed component. This is indicated by a dashed line in figure 7.

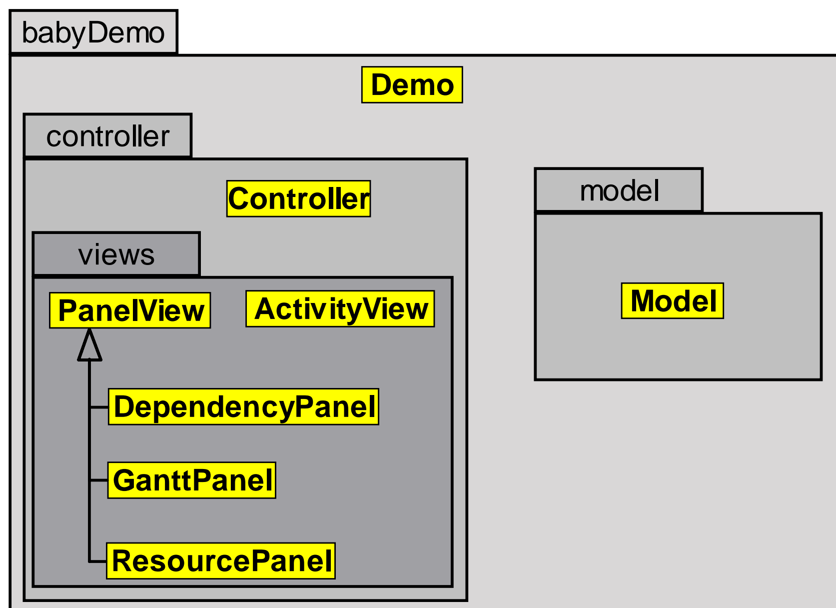
The Model and Controller are shown in red to indicate that they are the main collaborators in this implementation. The Views, being subordinate in this implementation, are shown in blue. The Java library superclasses are shown in gray along the top of the diagram.

Fig. 7: Demo class diagram



The component structure is reflected in the package structure as is illustrated in figure 8.

Fig. 8: The demo package diagram.



We will go into the details when we discuss the model internals and system behavior in section 4 on page 10.

3.5 Controller coordinates selection

We will now take selection as an example of how the controller coordinates the behavior of the views.

Fig. 9: actC is selected in all views where it appears.

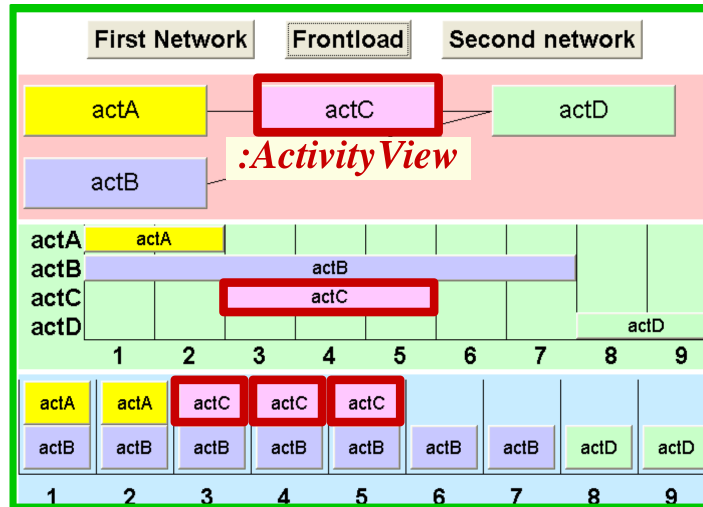
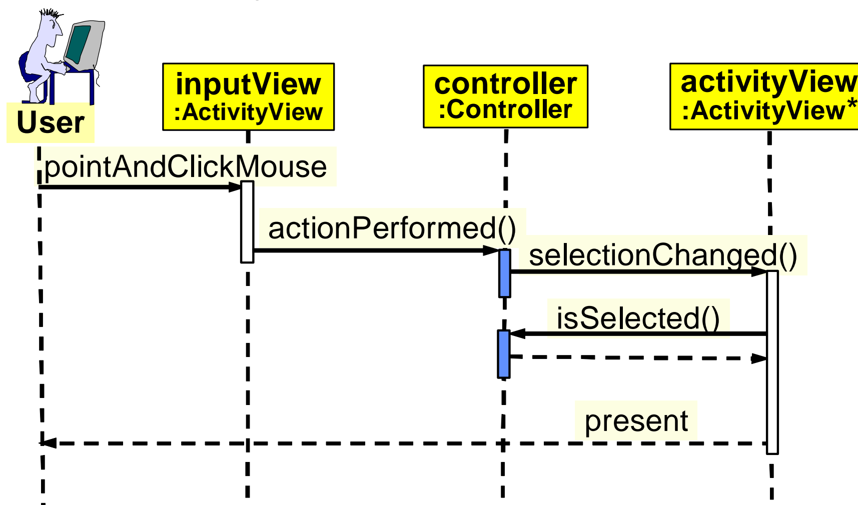


Figure 9 shows the window after the user has clicked on any of the actC activity symbols. The key to simplicity and generality is that the view being clicked only reports this event to the Controller. The Controller decides that this is indeed a selection command and that it shall be reflected in the appearance of all activity views. This behavior is illustrated in the sequence diagram of figure 10.

Fig. 10: The selection interaction.



The inputView role is shown to be some instance of class ActivityView. We see from figure 7 that ActivityView is an awt.Button, so it sends an actionPerfomed event to its actionListener. All activityViews are created to let their actionListener be the Controller:

```
Controller>>public void actionPerformed(ActionEvent e) {      (Java 1)
    ActivityView source = (ActivityView)e.getSource();        (Java 2)
    selection = source.activity();                            (Java 3)
    for (ActivityView view : activityViews) {                 (Java 4)
            view.selectionChanged();                          (Java 5)
                                                              (Java 6)
    repaint();                                                (Java 7)
}                                                             (Java 8)
```

The ActivityView now asks the controller if it is selected and then repaints itself appropriately:

```
ActivityView>>public void selectionChanged() {                (Java 9)
        if (controller.isSelected(activity)) {                (Java 10)
```

```
            setBackground(activity.color().darker());     (Java 11)
        } else {                                          (Java 12)
            setBackground(activity.color());              (Java 13)
        }                                                 (Java 14)
    }                                                     (Java 15)
```

and

```
    Controller>>public boolean isSelected(Activity act) {   (Java 16)
            return ( selection == act );                    (Java 17)
        }                                                   (Java 18)
```

3.5.1 Discussion: Observer pattern or direct View access from Controller?

The *Observer pattern*, also known as *Dependents*, *Publish-Subscribe, or Changed-update*, defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents are notified and updated automatically.[GOF]

A variant of the selection interaction use the observer pattern to let the controller alert the views about a changed selection. On the face of it, this is very flexible, extensible, and so on. But in this case, it would just be an obfuscator. The observer pattern is useful when the subject should be decoupled from its dependents. But here, the controller knows its views since it created them. The direct solution used here is the simplest and does not restrict flexibility and extensibility.

3.5.2 Discussion: The panels could be subordinate controllers.

We see from figure 7 on page 8 that the controller knows both panels and activityViews. An alternative could to let the controller know the panelViews only. Each panelView could then act as a local controller for its activityViews. The top-level programmer of the top level controller would then not need to know the inner workings of the panels. We did not choose this solution here because the responsibility for activity selection is anchored in the top level controller. The structure has to be known at the top, so there is no reason to complicate the code to attain a fictitious flexibility.

4 DCA: The Data-Collaboration-Algorithm paradigm

We now come to the Model. Seen from the Controller, it looks like an ordinary object. But a single object containing the activities and the resource would be a monster, so we have to give it some structure. The *DCA paradigm* tells us how to master this monster object; the Model becomes a DCA component. It looks like an object from the outside, but it has a well ordered and powerful object structure inside it.

4.1 The Model as a single object

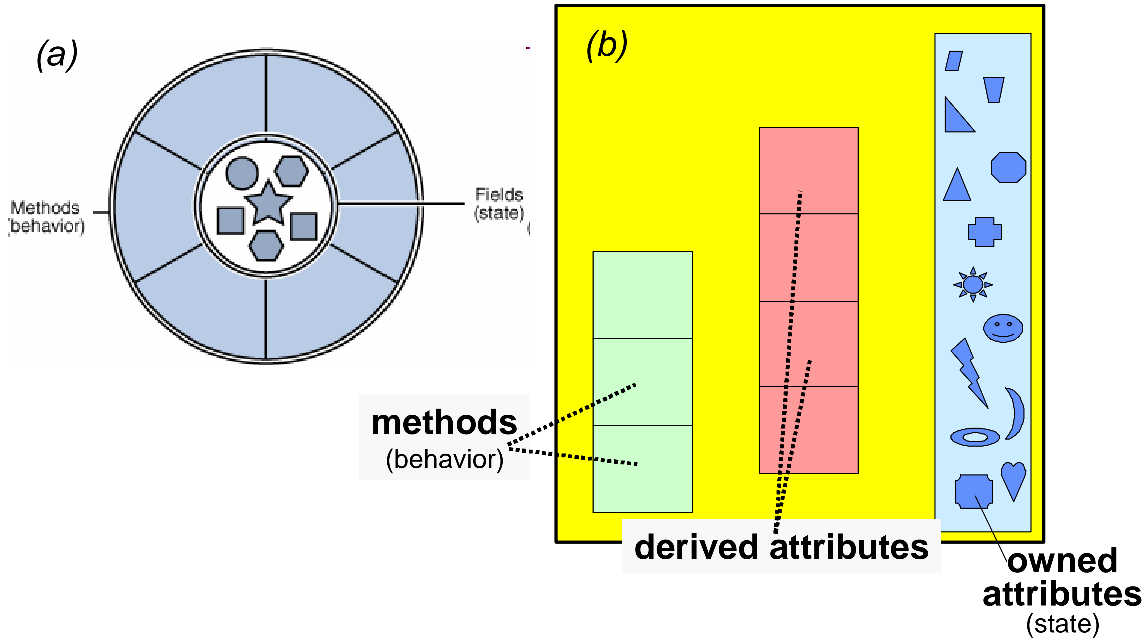
The Java tutorial[1] describes an object as a number of fields (state) surrounded by methods (behavior) as illustrated in figure 11(a). This is actually a better illustration of the Smalltalk object than the Java object. In Smalltalk, the fields ("instance variables") are invisible from outside the object; all access has to be through activating the methods by sending messages to the object. The Java object is different; the fields *are* visible from the outside. I write `x = foo.field1;` to access a field directly, and I write `x = foo.getField1();` to access it through a method.

Figure 11(b) shows an object model that we use as a starting point for discussing DCA. Borrowing terminology from UML, we use the term *owned attributes* to denote the state (fields, instance variables).

1.http://java.sun.com/docs/books/tutorial/java/concepts/object.html

We use the UML term *derived attributes* to denote attributes that are computed rather then stored. For example, a Person object may have birthDate as an owned attribute, while age() would be a derived attribute. The methods here implement the object's other operations.

Fig. 11: (The regular object is an instance of a class.
a) The object as depicted in the Java tutorial. (b) A more accurate object model.



4.2 The DCA Component; a well-structured monster object

There are (at least) two definitions of the term *object*. The simple one is that *an object is an instance of a class*. The other one is that an object is an entity that encapsulates state and behavior. The two definitions are synonymous in a Java program, but they are different in DCA.

Fig. 12: The DCA component.

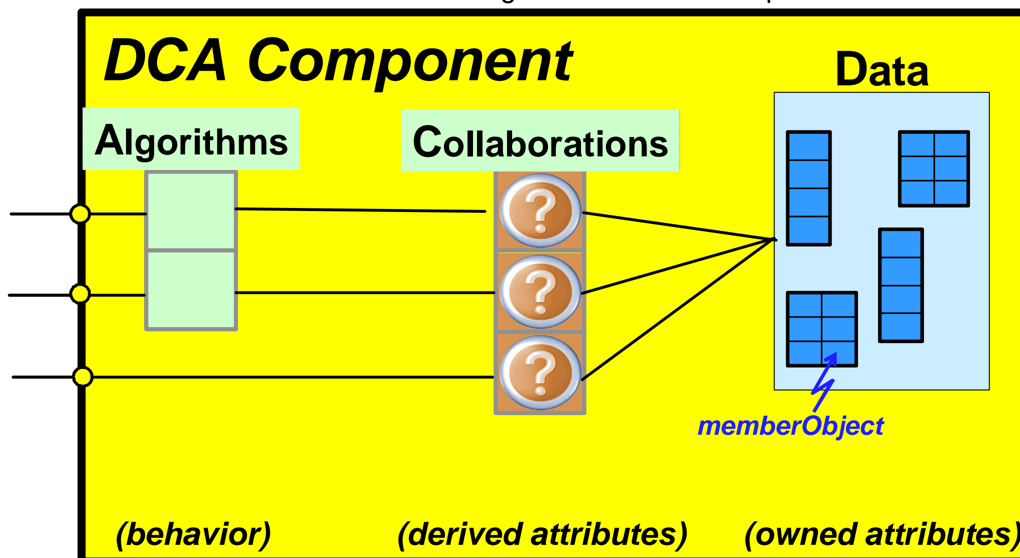


Figure 12 illustrates the DCA component as an object that encapsulates other objects. The DCA component looks like the object of figure 11 when seen from its environment. Inside, we find a number of specialized objects:

¤ ***Data***. This corresponds to the owned attributes of the regular object. The fields are replaced by a "baby database" that holds the domain objects that are encapsulated in the component. The term database is used in a restricted sense, we do not assume persistence, concurrency, access control, security, or any other goodie usually associated with databases. We call it the *base* for short.
  - The base is organized as a number of base relations in first normal form, ensuring referential integrity.
  - The values of these relations are the domain objects (including structure objects).
  - The domain objects are the objects that are encapsulated by the component.
  - Traditionally, the system structure is distributed as links in the domain objects. Here, the structure is in explicit relations. The domain objects are correspondingly simplified.
  - The code for base should ideally be declarative in the form of a *conceptual schema*, but we here rely on defining some Java classes with only getter and setter methods.

¤ ***Collaboration***. This corresponds to the derived attributes of the regular object. The base conceptual schema will normally not be ideal for the access requirements for the different uses of the data. The collaborations implement external schemas, each optimized for a particular usage of the base data.
  - UML defines " *a collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity and precise class of the actual participating instances are suppressed.*". Also "*a collaboration use represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration*."
  - In this demo, a *DCA collaboration* is a class that has the collaboration roles as fields and a number of database queries as methods. In its abstract form, it is one or more temporary relations that are derived from the base relations by queries.
  - The *DCA collaboration use* is an instance of the above class where the results of the queries are assigned to the role fields, thus binding roles to actual domain objects.
  - Objects using the collaboration see the base data in a perspective optimized for their use. Note that these user objects can be internal or external to the Model.

¤ ***Algorithm.*** The domain objects interact in order to "collectively accomplish some desired functionality". Traditionally, the code for this interaction is implicit by being distributed among the domain objects. In the DCA paradigm, the code controlling the interaction is pulled out and centralized in the component's algorithms. Thus, a DCA algorithm defines how the system accomplishes some desired functionality. The code becomes more readable because the interaction is explicit and the domain object classes are simplified.

4.3 The demo Model as a DCA component

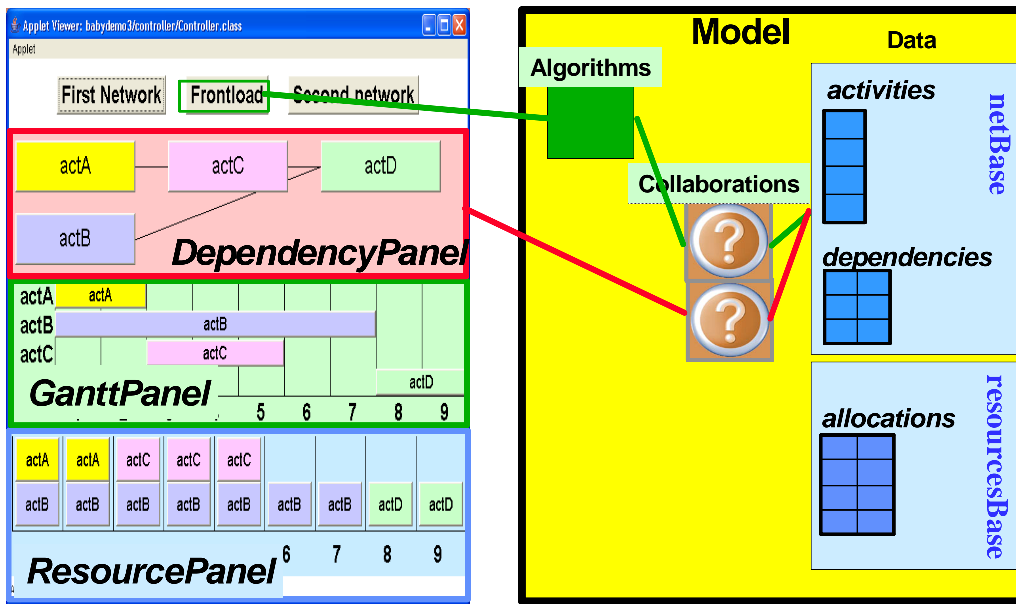
The demo Model is shown in figure 13. For illustrative purposes, the data is kept in two bases. The *netBase* holds the activity network in two relations: *activities* and *dependencies*, and the *resourceBase* has a single relation, *allocations*. (*allocations* has two attributes: *week* and *activity*).

There are five objects that uses the data, each seeing the data through a collaboration. Two uses are internal to the Model. The *frontload* algorithm that computes the early start and finish for each activity, and the resource allocation *algorithm*.

The three panelViews are external to the Model component, each needs to see the data in a unique perspective.

In the following, we will discuss the code DependencyPanel and the frontload algorithm with their tailored data access.

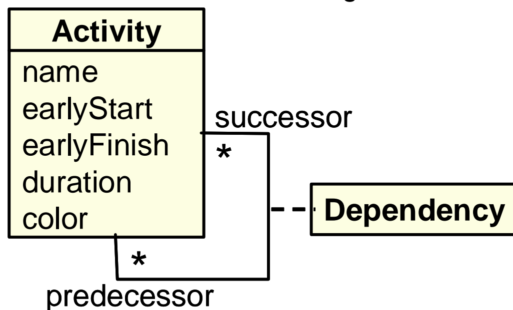
Fig. 13: The demo model is a DCA model.



4.4 The Data part defined by its schema

The data *bases* are defined by their schemas. Figure 14 shows the netBase schema expresses as a UML class diagram. (An ideal schema language would also name the relations, here *activities* and *dependencies*).

Fig. 14: The *netBase* schema as a UML class diagram.



The Java class declarations for the same are as follows:

```
public class Activity {                                          (Java 19)
    private Integer earlyStart, earlyFinish, duration;           (Java 20)
    private String name;                                         (Java 21)
    private Color color = Color.gray;                            (Java 22)
    ...                                                          (Java 23)
}                                                                (Java 24)
```

and:

```
public class MemberDependency {                                  (Java 25)
    private Activity predecessor, successor;                     (Java 26)
    ...                                                          (Java 27)
```

```
    }                                                                                    (Java 28)
```

The complete code is in appendix 1.

Comment: I personally prefer the graphical form because it is more compact and I can see at a glance what it is all about. Perhaps somebody will make an Eclipse solution to let something like Figure 14 *be* my schema declaration code. The diagram language has to be developed further if it shall be equivalent to the code in appendix 1.

4.5 Example 1: Panel layout

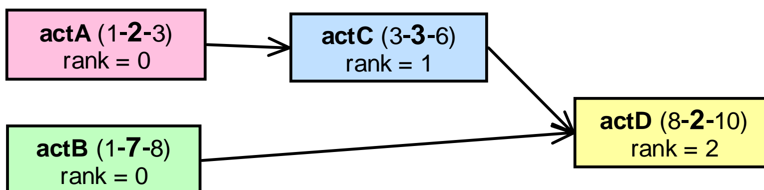
We now look at the DependencyPanel layout. We see from figure 13 that the DependencyPanel access model data through a collaboration.

The DependencyPanel layout method is as simple as can be. (Actually too simple, it will often lead to overlapping dependency lines.)

```
private void addActivityViews() {                                                        (Java 29)
38      Integer gridX = getSize().width / (rankedCollab.maxRank() + 1);                  (Java 30)
39      Integer gridY = getSize().height / rankedCollab.maxSizeActivitySets();           (Java 31)
40      Integer x0 = 10;                                                                 (Java 32)
41      Integer y0 = 10;                                                                 (Java 33)
42      Dimension buttonExtent = new Dimension(gridX-50 , gridY-20);                     (Java 34)
43      for (int rank=0; rank <= rankedCollab.maxRank(); rank++) {                       (Java 35)
44          Integer xPos = x0 + (gridX * rank);                                          (Java 36)
45          Integer yPos = y0;                                                           (Java 37)
46          for (Activity act : rankedCollab.activityListAtRank(rank)) {                 (Java 38)
47              ActivityView actView = new ActivityView(controller, act, 24) ;           (Java 39)
48              activityMapActivityViews.put(act, actView);                              (Java 40)
49              actView.setBounds(xPos , yPos , gridX-50 , gridY-20);                    (Java 41)
50              controller.addActivityView(actView);                                     (Java 42)
51              add(actView);                                                            (Java 43)
52              yPos = yPos + gridY;                                                     (Java 44)
53          }                                                                            (Java 45)
54      }                                                                                (Java 46)
```

Figure 15 illustrates that the unit on the horizontal axis is the activity *rank*; i.e., the max length of the activity's predecessor chain from the activity to the start of the network. Activities having the same rank are stacked vertically.

Fig. 15: The ranked activities



The above method accesses the activities through the rankedCollab, an instance of the RankedCollab class. This collaboration presents the data in a table with two columns: rank and activity. This table is accessed through the call to activityListAtRank() in line 46 above.

The corresponding code in the RankedCollab class:

```
public List<Activity> activityListAtRank(Integer rank) {                                 (Java 47)
    List<Activity> activityListAtRank = new ArrayList<Activity>();                       (Java 48)
```

```
    for (Activity act : netBase. activities()) {                    (Java 49)
        if (rankOf(act) == rank) {                                  (Java 50)
            activityListAtRank.add(act);                            (Java 51)
    }                                                               (Java 52)
}                                                                   (Java 53)


public List<Activity> activityListAtRank(Integer rank) {            (Java 54)
    List<Activity> activityListAtRank = new ArrayList<Activity>();  (Java 55)
    for (Activity act : netBase. activities()) {                    (Java 56)
        if (rankOf(act) == rank) {                                  (Java 57)
            activityListAtRank.add(act);                            (Java 58)
    }                                                               (Java 59)
}                                                                   (Java 60)
```

The complete RankedCollab code is given in Appendix 3:*The RankedCollab class* (page26).

4.6 Example 2: Frontloading

Frontloading is the calculation of the *earlyStart* and earlyFinish for each activity given the start time of the start activities. We see from figure 15 that actA and actB can both start when the project starts, e.g., in week 1. actA then finish in week 2 and actB in week 7. We can now compute for earlyStart and earlyFinish for actC. actD can now be computed since we know the earlyFinish for both actC and actB. The result of the frontloading is shown in the gantt diagram of figure 2 on page 4.

The frontloading operation is traditionally distributed among the activity objects. The default method could look like the following:

```
Activity >> public void frontloadSimple (Integer startWeek) {
    earlyStart = startWeek;;
    for(Activity pred : predecessors()) {
        earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1);
    }
}
```

There are problems with this simple solution. This method cannot be called in an activity object before the earlyFinish of all predecessors have been done. This means that the frontload operation belongs in the inter-activity space and must be treated at a higher system level.

The common frontload logic could be in a method in a Model class, but I feel that this is an overloading of a class that should be clean and simple. So the top level frontload() method is here coded in a separate class, the FrontloadAlgorithm class. Three problems need to be resolved. One is identifying activities that are ready to be frontloaded. The second is to determine the earlyStart of an activity once all its predecessors have been loaded. The third is to compute the earlyFinish for an activity once its earlyStart is known.

Identifying activities that are ready to be planned is essentially a query on the base. This work properly belongs in a collaboration class, here the FrontloadCollab class.

5. The earlyStart of an activity depends on all its predecessors and all modifiers such as overlapping activities etc. This logic belongs in the inter-activity space and is here coded in the FrontloadAlgorithm class.
6. The earlyFinish of an activity once its earlyStart depends on the activity alone. The code, therefore, belongs in the Activity class.

We'll discuss each of these actions in turn.

4.6.1 FrontloadCollab*, the frontloading collaboration*

There is a simple solution for finding the activities that are ready to be loaded. Activities with rank=0 have no predecessors, so they can be loaded first. Once they are done, activities with rank=1 can be loaded, and so on. This solution has the added benefit that we could reuse the RankedCollab collaboration.

We choose a different solution here because we are not 100% certain that the ranking solution holds for all possible structures. More important, we choose a query-based solution to illustrate how a query gives different results through the frontloading process. Here is a query in an unspecified language that finds a candidate activity for frontloading:

```
define frontloader() as
    (select act
     from Activities act
     where act.earlyStart == null
        and (for all pred in predecessors(act):
                pred.earlyStart != null)
    ) someInstance
```

Here is an excerpt from the FrontloadCollab class with the corresponding Java code:

```
public Activity frontloader() {                          (Java 61)
        for (Activity act : netBase.activities()) {      (Java 62)
            if (act.earlyStart() == null) {              (Java 63)
                Set<Activity> predSet = predecessorsOf(act);  (Java 64)
                if (areAllDone(predSet)) {               (Java 65)
                    frontloader = act;                   (Java 66)
                    return(frontloader);                 (Java 67)
                }                                        (Java 68)
            }                                            (Java 69)
        }                                                (Java 70)
        return null;                                     (Java 71)
    }                                                    (Java 72)
```

The areAllDone() method in code line Java 65

```
private boolean areAllDone(Set<Activity> actSet) {       (Java 73)
        boolean allPredsDone = true;                     (Java 74)
        for ( Activity pred : actSet) {                  (Java 75)
            if (pred.earlyStart() == null) {             (Java 76)
                allPredsDone = false;                    (Java 77)
                break;                                   (Java 78)
            }                                            (Java 79)
        }                                                (Java 80)
        return allPredsDone;                             (Java 81)
    }                                                    (Java 82)
```

This code is not trivial. But it is nicely isolated. I can give it to a colleague and ask her to audit it and sign it. (So that any unlikely bug will be her fault, not mine.)

The complete code for class FrontloadCollab is in appendix 5.

4.6.2 FrontloadAlgorithm*, implementing the frontloading interaction*

The frontloading interaction is implemented in the FrontloadAlgorithm class. It is here the simple, default case with no modifiers or other obfuscators:

```
public void frontload(Integer startWeek) {                                    (Java 83)
    // reset all                                                               (Java 84)
    for (Activity act : frontloadCollab.resetters()) {                         (Java 85)
      act.setEarlyStart(null);                                                 (Java 86)
    }                                                                          (Java 87)
    // frontload all                                                           (Java 88)
    Activity frontloader;                                                      (Java 89)
    while ((frontloader = frontloadCollab.frontloader()) != null) {            (Java 90)
      Integer earlyStart = startWeek;                                          (Java 91)
      for(Activity pred : frontloadCollab.frontPredecessors()) {               (Java 92)
        earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1);             (Java 93)
      }                                                                        (Java 94)
      frontloader.setEarlyStart(earlyStart);                                   (Java 95)
    }                                                                          (Java 96)
  }                                                                            (Java 97)
```

The algorithm code is pure with no confusing side issues. It is thus a good starting point for dealing with more complex situations.

5 Summary and Discussion

The Model-View-Controller paradigm bridges the gap between the human brain and the domain data stored in the computer.

- The domain data are represented in an object called the MVC *Model*.
- The human user observes and manipulates these data through an MVC *View*. The view shall ideally match the human mental model, giving the user the illusion that the model is faithfully represented in the computer.
- The MVC *Controller* creates several Views and coordinates their actions.

In the demo, the Model is implemented as a *DCA Component*. This is a “monster object” that looks like a regular object in its environment, but it has an internal structure clearly prescribed by the *DCA paradigm*.

- The *DCA Data* (state) are the *Domain Objects* that represent the substance of the component. They are kept in a “micro database” with an explicit conceptual schema. Ideally, its schema declaration is declarative.
- The essence of object orientation is that objects interact to accomplish some desired functionality. A DCA *Collaboration* is an external view on the DCA Data, declaring the Domain Objects and links that are active in realizing a certain interaction.
- The DCA *Algorithm* transcends individual domain objects and explicitly specifies how and why they interact. Its code is imperative.

5.1 The MVC paradigm

MVC is an old paradigm that has survived for more than 30 years. Its fundamental part is to separate the Model from the View. The model should be pure representation of information, while the view should be pure presentation.

In practice, may not be obvious what should go where. Should input checking be in the view or the model? I put it in the view if it is based on a rule that is independent of the actual model data; in the model

otherwise. The general rule we have used in several systems is to let the model handle all persistent data and let the view be automatically generated from the model data.

Another problem arises if the view cannot be generated automatically from the model data. An example could be the layout algorithm of dependency diagram in the current demo. I actually had to hack its code to make a satisfactory diagram. The diagram becomes unreadable if the position of *actA* and *actB* are interchanged. Automatic layout in two dimensions far from simple, and the best solution could be to let the user do the layout. Where should we then keep the layout data? One solution is to add the layout data to the Model database. Another is to create a separate database that is dependent upon the Model. If we have several views on the same data, we have to deal with obsolete views. One solution I have used is to let the view clearly show that it is obsolete and let it be up to the user when to update it.

In any case, the separation of Model and View is applicable when

- the View/Controller is deployed on a machine different from the Model,
- when the application code is complex,
- when the user needs to see the Model in different perspectives.

A Controller is applicable when

- the user needs to see several Views simultaneously.

5.2 The DCA paradigm

The DCA paradigm is new and untested in practice. Yet its seem to be applicable in many situations.

It seems a good idea to pull the data out of the domain objects if the data structure is not obvious from reading the domain object code. This should make it easier to check the code for correctness and for a new person to get an understanding of the data structure by reading the code.

Algorithms occur in two places in the DAC paradigm. Some are local to the domain objects and is coded as a method in its class. Other describe domain object interaction. They are coded in separate classes that are distinct from the domain object classes. This ensures that object interaction is specified explicitly, again making it easier to check the code for correctness and for a new person to get an understanding of what goes on in the system.

The Collaborations are applicable when the base data structure is in some way unsuitable for use by the algorithms. They lead to a simplification of the DCA Algorithms and also code that access the Data from outside the DCA component .

5.3 Where are we?

The current experiment is one in a series in the pursuit of the utmost simplicity. The experiment illustrates a number of simplifications. The MVC paradigm is well known simplification, we will not discuss it here. The DCA is new and deserves a few comments.

The first DCA simplification is the notion of a Component that looks like a regular object in its environment, but that encapsulates a strictly defined object structure within its boundary. The notion of a component is recursive; the encapsulated objects can turn out to be components in their own right without this being apparent from their external properties. The partitioning of the total system into components is an important contribution to simplicity.

As any object, a component encapsulates state and behavior. The state is a collection of *domain objects* that constitute the component's *base data*. Their organization is declared as a number of relations in first normal form, guaranteeing data integrity. The data declaration can be read and understood independently of the system around it; an important step towards system simplicity. The recursive property of components ensures that the data declaration is also recursive.

A next step in simplification is to make the component behavior explicit. The behavior of a component consists of its external operations. Each operation is accomplished by an interaction between some or all of the domain objects. For each operation, we thus need to answer the following three questions:

1. What are the objects?
2. How are they interlinked?
3. How do they interact?

The objects involved in an interaction are a subset of the domain objects. The answer to the first two questions is well known from database technology. We declare an external schema that defines this subset. We call it a *Collaboration* because it describes how the interacting objects play their individual roles in the realization of the external operation. In an instance of the Collaboration, the roles are bound to the component's actual domain objects by suitable queries. The Collaboration is another simplification. The substructure can be understood from its declaration and used independently of any complexity in the base data.

The interaction itself is defined by explicit Algorithm methods defined in classes that belong to the component as a whole. Again, we have made important, high level features explicit and visible. A reader of this code will know how domain objects interact to realize a particular component operation.

5.4 What's next?

I claim that the DCA paradigm in many cases will lead to simpler and more readable code. It has as yet only been applied to the current simple demo program. But I believe it has a value as it stands and that it will be well worth while to look for real applications where it can be applied.

The greatest weakness of the current demo is the size of the code. Johannes Brodwall wrote a version of the demo program in his usual style; his source code is about 20kB. My current implementation is has slightly more features and is about 35kB. We want to study the readability of the code itself so both implementations have very few comments. The difference is to be expected, I would expect more readable code to be more verbose. And of course, the difference would be almost invisible if we added the recommended comments to our code.

The next step in the pursuit of simplicity is to look at the code itself. Given the DCA structure, can we make the code more expressive, saving volume and increasing readability. I see three points of attack:

- *More powerful superclasses.* A most obvious start. There might be something in the ODMG package that could be used.
- Better IDE. It could be interesting to use Eclipse to explore more powerful programming tools.
- New declarative languages and new IDEs that fully support the MVC and DCA paradigms.

I plan to switch to Smalltalk for a while in order to explore the last alternative.

6 Acknowledgements

Many thanks to Ragnar Norman for sharing his deep understanding of database technology. (I apologize for any misrepresentations of his advice). My sincere thanks also go to Johannes Brodwall for his intelligent support and advice on Java technology.

7 References.

[Hoare]	Charles Antony Richard Hoare: *The Emperor's Old Clothes.* 1980 Turing Award lecture. Comm.ACM 24, 2 (Feb. 1981)
[Wein]	Weinberg, Gerard: *The Psychology of Computer Programming*. ....
[IFIP]	*IFIP-ICC Vocabulary of Information Processing*. North-Holland 1966.
[GOF]	Gamma et.al.: *Design Patterns.* Addison-Wesley, Reading, Mass. 1995. ISBN 0-201-63361-2
[ODMG]	Cattell, Barry: The Object Data Standard: ODMG 3.0. Academic Press, London, 2000. ISBN 1-55860-647-4 (http://www.odmg.org/)
[2]	Coplien, James: *Multi Paradigm Design for C++*, Addison-Wesley Professional, 1998, ISBN: 0-201-82467-1
[3]	Goldberg, Robson: *Smalltalk-80, the language and its implementation.* ("The Blue Book"). Addison-Wesley, Reading 1983. ISBN0-201-11371-6
[4]	Hay, David: *What Exactly IS a Data Model?* DM Review Magazine, February 2003
[5]	Hysing, Reenskaug: *A System for Computer Plate Preparation*. Numerical Methods Applied to Shipbuilding. A NATO Advanced Study Institute. Oslo-Bergen, 1963.
[6]	Reenskaug: *Administrative Control in the Shipyard.* ICCAS conference, Tokyo, 1973. (http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf)
[7]	Reenskaug: *Prokon/Plan. A Modelling Tool for Project Planning and Control.* IFIP Congress, Toronto, Canada, 1977. http://heim.ifi.uio.no/~trygver/1977/Prokon/IFIP-Prokon.pdf
[8]	Reenskaug et.al.: *Working with objects. The OOram Software Engineering Method.* Prentice-Hall 1996. Early version scanned at (http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf)

Trygve Reenskaug is professor emeritus of informatics at the University of Oslo. He has 40 years experience in software engineering research and the development of industrial strength software products. He has extensive teaching and speaking experience including keynotes, talks and tutorials. His firsts include the Autokon system for computer aided design of ships with end user programming language, structured programming, and a data base oriented architecture from 1960; object oriented applications and role (collaboration) modeling from 1973; Model-View-Controller, the world's first reusable object oriented framework, from 1979; OOram role modeling method and tool from 1983. Trygve was a member of the UML Core Team and was a contributor to UML 1.4. The goal of his current research is to create a new, high level discipline of programming that lets us reclaim the mastery of software.

Appendix 1: The NetBase code

```
package babydemo3.model.data;                                                    (Java 98)
                                                                                 (Java 99)
import babydemo3.model.data.Activity;                                            (Java 100)
import java.util.*;                                                              (Java 101)
import java.awt.Color;                                                           (Java 102)
                                                                                 (Java 103)
public class NetBase extends Observable implements Observer{                     (Java 104)
                                                                                 (Java 105)
   private Set<Activity> activities = new HashSet<Activity>();                   (Java 106)
   private Set<MemberDependency> dependencies = new HashSet<MemberDependency>(); (Java 107)
                                                                                 (Java 108)
   public NetBase() {                                                            (Java 109)
   }                                                                             (Java 110)
   public void reset() {                                                         (Java 111)
      activities.clear();                                                        (Java 112)
      dependencies.clear();                                                      (Java 113)
   }                                                                             (Java 114)
   public void update(Observable o, Object arg) {                                (Java 115)
      setChanged();                                                              (Java 116)
      notifyObservers((String)arg);                                              (Java 117)
   }                                                                             (Java 118)
   public void addActivity(String name, Integer duration, Color color) {         (Java 119)
      Activity act = new Activity(name, duration, color);                        (Java 120)
      activities.add(act);                                                       (Java 121)
      setChanged();                                                              (Java 122)
   }                                                                             (Java 123)
   public void addDependency(String predName, String succName) {                 (Java 124)
      Activity pred = activityNamed(predName);                                   (Java 125)
      Activity succ = activityNamed(succName);                                   (Java 126)
      if (!hasDependency(pred , succ)) {                                         (Java 127)
         MemberDependency dep = new MemberDependency(pred , succ);               (Java 128)
         dependencies.add(dep);                                                  (Java 129)
         setChanged();                                                           (Java 130)
      }                                                                          (Java 131)
   }                                                                             (Java 132)
   public Collection<Activity> activities() {                                    (Java 133)
      return activities;                                                         (Java 134)
   }                                                                             (Java 135)
   public Set<MemberDependency> dependencies() {                                 (Java 136)
      return dependencies;                                                       (Java 137)
   }                                                                             (Java 138)
                                                                                 (Java 139)
   private Activity activityNamed(String name) {                                 (Java 140)
      for (Activity act: activities) {                                           (Java 141)
         if(act.name() == name) return act;                                      (Java 142)
      }                                                                          (Java 143)
   return null;                                                                  (Java 144)
   }                                                                             (Java 145)
   private boolean hasDependency(Activity pred , Activity succ) {                (Java 146)
```

```
        for (MemberDependency assoc : dependencies ) {                    (Java 147)
            if ((assoc.predecessor() == pred) && (assoc.successor() == succ)) {   (Java 148)
                return true;                                              (Java 149)
            }                                                             (Java 150)
        }                                                                 (Java 151)
        return false;                                                     (Java 152)
    }                                                                     (Java 153)
}                                                                         (Java 154)
```

Appendix 2: The DependencyView class

```
package babydemo3.controller.views;                                          (Java 155)
                                                                              (Java 156)
import babydemo3.controller.*;                                                (Java 157)
import babydemo3.model.Model;                                                 (Java 158)
import babydemo3.model.data.Activity;                                         (Java 159)
import babydemo3.model.collaboration.RankedCollab;                            (Java 160)
import java.awt.*;                                                            (Java 161)
import java.util.List;                                                        (Java 162)
import java.util.*;                                                           (Java 163)
                                                                              (Java 164)
public class DependencyPanel extends PanelView implements Observer {          (Java 165)
  private Map<Activity, ActivityView> activityMapActivityViews = new HashMap<Activity,ActivityView>(0);  (Java 166)
  private RankedCollab rankedCollab;                                          (Java 167)
                                                                              (Java 168)
  public DependencyPanel(Controller cntr) {                                   (Java 169)
    super(cntr);                                                              (Java 170)
    setBackground(new Color(255, 200, 200));                                  (Java 171)
    rankedCollab= model.rankedCollab();                                       (Java 172)
    rankedCollab.addObserver(this);                                           (Java 173)
  }                                                                           (Java 174)
  public void update(Observable o, Object arg) {                              (Java 175)
    if ((arg == "MODEL") || (arg == "ACTIVITY")) {                            (Java 176)
      refresh();                                                              (Java 177)
    }                                                                         (Java 178)
  }                                                                           (Java 179)
                                                                              (Java 180)
  void refresh() {                                                            (Java 181)
    super.refresh();                                                          (Java 182)
    activityMapActivityViews = new HashMap<Activity, ActivityView>(0);        (Java 183)
    if (rankedCollab.maxRank() > 0) {                                         (Java 184)
      addActivityViews();                                                     (Java 185)
      addLines();                                                             (Java 186)
    }                                                                         (Java 187)
    validate(); repaint();                                                    (Java 188)
    controller.validate(); controller.repaint();                              (Java 189)
  }                                                                           (Java 190)
  private void addActivityViews() {                                           (Java 191)
    Integer gridX = getSize().width / (rankedCollab.maxRank() + 1);           (Java 192)
    Integer gridY = getSize().height / rankedCollab.maxSizeActivitySets();    (Java 193)
    Integer x0 = 10;                                                          (Java 194)
    Integer y0 = 10;                                                          (Java 195)
    Dimension buttonExtent = new Dimension(gridX-50 , gridY-20);              (Java 196)
    for (int rank=0; rank <= rankedCollab.maxRank(); rank++) {                (Java 197)
      Integer xPos = x0 + (gridX * rank);                                     (Java 198)
      Integer yPos = y0;                                                      (Java 199)
      for (Activity act : rankedCollab.activityListAtRank(rank)) {            (Java 200)
        ActivityView actView = new ActivityView(controller, act, 24) ;        (Java 201)
        activityMapActivityViews.put(act, actView);                           (Java 202)
        actView.setBounds(xPos , yPos , gridX-50 , gridY-20);                 (Java 203)
```

```
            controller.addActivityView(actView);                                    (Java 204)
            add(actView);                                                           (Java 205)
            yPos = yPos + gridY;                                                    (Java 206)
         }                                                                          (Java 207)
      }                                                                             (Java 208)
   }                                                                                (Java 209)
   void addLines() {                                                                (Java 210)
      super.addLines();                                                             (Java 211)
      for (Activity fromAct : activityMapActivityViews.keySet()) {                  (Java 212)
         ActivityView fromView = activityMapActivityViews.get(fromAct);             (Java 213)
         for (Activity toAct : rankedCollab.successorsOf(fromAct)) {                (Java 214)
            ActivityView toView = activityMapActivityViews.get(toAct);              (Java 215)
            DependencyLine line = new DependencyLine                                (Java 216)
                  (                                                                 (Java 217)
                  (fromView.getBounds().x + (fromView.getBounds().width))           (Java 218)
                  , (fromView.getBounds().y + (fromView.getBounds().height / 2))    (Java 219)
                  , (toView.getBounds().x)                                          (Java 220)
                  , (toView.getBounds().y + (toView.getBounds().height / 2))        (Java 221)
                  );                                                                (Java 222)
            lines.add(line);                                                        (Java 223)
         }                                                                          (Java 224)
      }                                                                             (Java 225)
   }                                                                                (Java 226)
}                                                                                   (Java 227)
```

Appendix 3: The RankedCollab class

```
package babydemo3.model.collaboration;                                    (Java 228)
                                                                          (Java 229)
import babydemo3.model.data.*;                                            (Java 230)
import java.util.*;                                                       (Java 231)
                                                                          (Java 232)
public class RankedCollab extends Observable implements Observer {        (Java 233)
    private NetBase netBase;                                              (Java 234)
                                                                          (Java 235)
    public RankedCollab(NetBase base) {                                   (Java 236)
        netBase = base;                                                   (Java 237)
        netBase.addObserver(this);                                        (Java 238)
    }                                                                     (Java 239)
    public void update(Observable o, Object arg) {                        (Java 240)
        if (arg == "MODEL") {                                             (Java 241)
            setChanged();                                                 (Java 242)
            notifyObservers(arg);                                         (Java 243)
        }                                                                 (Java 244)
    }                                                                     (Java 245)
                                                                          (Java 246)
    /** define maxRank as                                                 (Java 247)
     *   max (select rankOf(act) from Activities act)                     (Java 248)
     */                                                                   (Java 249)
    public Integer maxRank() {                                            (Java 250)
        Integer maxRank = 0;                                              (Java 251)
        for (Activity act : netBase.activities()) {                       (Java 252)
            maxRank = Math.max(maxRank, rankOf(act));                     (Java 253)
        }                                                                 (Java 254)
        return maxRank;                                                   (Java 255)
    }                                                                     (Java 256)
                                                                          (Java 257)
    /** define activityListAtRank(Integer rank) as                        (Java 258)
     *   select act                                                       (Java 259)
     *   from activities act                                              (Java 260)
     *   where rank(act) = rank                                           (Java 261)
     */                                                                   (Java 262)
    public List<Activity> activityListAtRank(Integer rank) {              (Java 263)
        List<Activity> activityListAtRank = new ArrayList<Activity>();    (Java 264)
        for (Activity act : netBase. activities()) {                      (Java 265)
            if (rankOf(act) == rank) {                                    (Java 266)
                activityListAtRank.add(act);                              (Java 267)
            }                                                             (Java 268)
        }                                                                 (Java 269)
        // Hack. Sort to ensure always same diagram.                      (Java 270)
        Collections.sort(activityListAtRank, NAME_ORDER);                 (Java 271)
        return activityListAtRank;                                        (Java 272)
    }                                                                     (Java 273)
                                                                          (Java 274)
    /** define maxSizeActivitySets as                                     (Java 275)
     *    max (                                                           (Java 276)
```

```
 *         select count(activityListAtRank(i))                                (Java 277)
 *         from (0..maxRank()) i                                              (Java 278)
 *     )                                                                      (Java 279)
 */                                                                           (Java 280)
public Integer maxSizeActivitySets() {                                        (Java 281)
    Integer maxListSize = 0;                                                  (Java 282)
    for (Integer rank=0; rank <= maxRank(); rank++ ) {                        (Java 283)
        maxListSize = Math.max(maxListSize, activityListAtRank(rank).size()); (Java 284)
    }                                                                         (Java 285)
    return maxListSize;                                                       (Java 286)
}                                                                             (Java 287)
                                                                              (Java 288)
/** define predecessorsOf(Activity act) as                                    (Java 289)
 *     select dep.predecessor                                                 (Java 290)
 *     from dependencies dep                                                  (Java 291)
 *     where dep.successor = act                                              (Java 292)
 **/                                                                          (Java 293)
public Set<Activity> predecessorsOf(Activity act) {                           (Java 294)
    Set<Activity> predecessors = new HashSet<Activity>();                     (Java 295)
    for (MemberDependency assoc : netBase.dependencies() ) {                  (Java 296)
        if (assoc.successor() == act) {                                       (Java 297)
            predecessors.add(assoc.predecessor());                            (Java 298)
        }                                                                     (Java 299)
    }                                                                         (Java 300)
    return predecessors;                                                      (Java 301)
}                                                                             (Java 302)
                                                                              (Java 303)
/** define successorsOf(Activity act) as                                      (Java 304)
 *     select dep.successor                                                   (Java 305)
 *     from dependencies dep                                                  (Java 306)
 *     where dep.predecessor = act                                            (Java 307)
 **/                                                                          (Java 308)
public Set<Activity> successorsOf(Activity act) {                             (Java 309)
    Set<Activity> successors = new HashSet<Activity>();                       (Java 310)
    for (MemberDependency assoc : netBase.dependencies() ) {                  (Java 311)
        if (assoc.predecessor() == act) {                                     (Java 312)
            successors.add(assoc.successor());                                (Java 313)
        }                                                                     (Java 314)
    }                                                                         (Java 315)
    return successors;                                                        (Java 316)
}                                                                             (Java 317)
                                                                              (Java 318)
private Integer rankOf(Activity act) {                                        (Java 319)
    // Extremely inefficient. Early candidate for cashing.                    (Java 320)
    // NOTE: A feature of the structure, not an individual activity           (Java 321)
    Integer rnk = 0;                                                          (Java 322)
    for (Activity pred : predecessorsOf(act)) {                               (Java 323)
        rnk = Math.max(rnk, (rankOf(pred))+1);                                (Java 324)
    }                                                                         (Java 325)
    return rnk;                                                               (Java 326)
}                                                                             (Java 327)
                                                                              (Java 328)
```

```
    static final Comparator<Activity> NAME_ORDER = new Comparator<Activity>() {      (Java 329)
        public int compare(Activity act1, Activity act2) {                           (Java 330)
            return act1.name().compareTo(act2.name());                               (Java 331)
        }                                                                            (Java 332)
    };                                                                               (Java 333)
}                                                                                    (Java 334)
```

Appendix 4: The FrontloadAlgorithm class

```
package babydemo3.model.algorithm;                                      (Java 335)
                                                                        (Java 336)
import babydemo3.model.Model;                                           (Java 337)
import babydemo3.model.data.Activity;                                   (Java 338)
import babydemo3.model.collaboration.FrontloadCollab;                   (Java 339)
import java.util.*;                                                     (Java 340)
import java.awt.Color;                                                  (Java 341)
                                                                        (Java 342)
public class FrontloadAlgorithm {                                       (Java 343)
    private Model model;                                                (Java 344)
    private FrontloadCollab frontloadCollab;                            (Java 345)
                                                                        (Java 346)
    public FrontloadAlgorithm(Model mod) {                              (Java 347)
        model = mod;                                                    (Java 348)
        frontloadCollab = model.frontloadCollab();                      (Java 349)
    }                                                                   (Java 350)
                                                                        (Java 351)
    public void frontload(Integer startWeek) {                          (Java 352)
        // reset all                                                    (Java 353)
        for (Activity act : frontloadCollab.resetters()) {              (Java 354)
            act.setEarlyStart(null);                                    (Java 355)
        }                                                               (Java 356)
        // frontload all                                                (Java 357)
        Activity frontloader;                                           (Java 358)
        while ((frontloader = frontloadCollab.frontloader()) != null) { (Java 359)
            Integer earlyStart = startWeek;                             (Java 360)
            for(Activity pred : frontloadCollab.frontPredecessors()) {  (Java 361)
                earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1); (Java 362)
            }                                                           (Java 363)
            frontloader.setEarlyStart(earlyStart);                      (Java 364)
        }                                                               (Java 365)
    }                                                                   (Java 366)
}                                                                       (Java 367)
```

Appendix 5: The FrontloadCollab class

```
package babydemo3.model.collaboration;                                    (Java 368)
                                                                          (Java 369)
import babydemo3.model.data.*;                                            (Java 370)
import java.util.*;                                                       (Java 371)
                                                                          (Java 372)
public class FrontloadCollab extends Observable implements Observer {    (Java 373)
   private NetBase netBase;                                               (Java 374)
   private Activity frontloader;                                          (Java 375)
                                                                          (Java 376)
   public FrontloadCollab(NetBase base) {                                 (Java 377)
      netBase = base;                                                     (Java 378)
      netBase.addObserver(this);                                          (Java 379)
   }                                                                      (Java 380)
   public void update(Observable o, Object arg) {                         (Java 381)
      if ((arg == "MODEL") || (arg == "ACTIVITY")) {                      (Java 382)
         setChanged();                                                    (Java 383)
         notifyObservers(arg);                                            (Java 384)
      }                                                                   (Java 385)
   }                                                                      (Java 386)
                                                                          (Java 387)
   /** define resetters as                                                (Java 388)
    *    Activities                                                       (Java 389)
    */                                                                    (Java 390)
   public Collection<Activity> resetters() {                              (Java 391)
      return netBase.activities();                                        (Java 392)
   }                                                                      (Java 393)
                                                                          (Java 394)
   /**                                                                    (Java 395)
    * define frontloaders() as                                            (Java 396)
    *  select act                                                         (Java 397)
    *    from Activities act                                              (Java 398)
    *    where act.earlyStart == nil                                      (Java 399)
    *      and (for all pred in predecessors(act): pred.earlyStart != nil) (Java 400)
    */                                                                    (Java 401)
   public Activity frontloader() {                                        (Java 402)
      for (Activity act : netBase.activities()) {                         (Java 403)
         if (act.earlyStart() == null) {                                  (Java 404)
            Set<Activity> predSet = predecessorsOf(act);                  (Java 405)
            if (areAllDone(predSet)) {                                    (Java 406)
               frontloader = act;                                         (Java 407)
               return(frontloader);                                       (Java 408)
            }                                                             (Java 409)
         }                                                                (Java 410)
      }                                                                   (Java 411)
      return null;                                                        (Java 412)
   }                                                                      (Java 413)
                                                                          (Java 414)
   /** define frontPredecessors(Activity act) as                          (Java 415)
    *     select dep.predecessor                                          (Java 416)
```

```
 *    from dependencies dep                                             (Java 417)
 *    where dep.successor = frontloader                                 (Java 418)
 **/                                                                    (Java 419)
public Set<Activity> frontPredecessors() {                              (Java 420)
    return predecessorsOf(frontloader);                                 (Java 421)
}                                                                       (Java 422)
                                                                        (Java 423)
/** define predecessorsOf(Activity act) as                              (Java 424)
 *    select dep.predecessor                                            (Java 425)
 *    from dependencies dep                                             (Java 426)
 *    where dep.successor = act                                         (Java 427)
 **/                                                                    (Java 428)
private Set<Activity> predecessorsOf(Activity act) {                    (Java 429)
    Set<Activity> predecessors = new HashSet<Activity>();               (Java 430)
    for (MemberDependency assoc : netBase.dependencies() ) {            (Java 431)
        if (assoc.successor() == act) {                                 (Java 432)
            predecessors.add(assoc.predecessor());                      (Java 433)
        }                                                               (Java 434)
    }                                                                   (Java 435)
    return predecessors;                                                (Java 436)
}                                                                       (Java 437)
private boolean areAllDone(Set<Activity> actSet) {                      (Java 438)
    boolean allPredsDone = true;                                        (Java 439)
    for ( Activity pred : actSet) {                                     (Java 440)
        if (pred.earlyStart() == null) {                                (Java 441)
            allPredsDone = false;                                       (Java 442)
            break;                                                      (Java 443)
        }                                                               (Java 444)
    }                                                                   (Java 445)
    return allPredsDone;                                                (Java 446)
}                                                                       (Java 447)
```