Trygve Reenskaug[1]

MVC and DCA
Example program comments

FINAL VERSION

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

---

1.   Department of Informatics, University of Oslo, Norway

TABLE OF CONTENTS

1 Introduction

1.1 The BabyUML Project

Once in a while, I have written a piece of 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 its code is distributed between its class and superclasses. Objects exist in a featureless landscape since there is no support for clustering. I can study every detail, but I have to imagine the whole. This is OK with small and insignificant programs, but my programming style does not scale and the complexity of significant 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:[Hoare-81]

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

The project shall give me a high level programming discipline that facilitates compact and simple descriptions of significant systems of collaborating objects. Lines of code and even efficiency come second. I argue that while low level code can be very efficient in the small; high level code is superior in the large because it is more readable and thus 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.

I believe the key to significant progress is to make the objects more visible in the code. There are two important abstractions on objects. The well known *class abstraction* clusters objects with common features. I claim that the *role abstraction* is equally useful.[Roles] because it clusters objects with a common purpose.

A source of ideas is the UML 2.x modeling language[UML]. UML is a very large edifice of reasonably unified and consistent programming concepts. My assumption is that some of its concepts can be adapted to the needs of a BabyUML discipline of programming.

The world's first stored program digital computer was called the Baby. It performed its first operations on the 21st of June 1948 at University of Manchester in England[HiNC] and was characterized by its programs being represented and operated upon as regular data. I expect BabyUML will similarly need to represent and operate upon its own programs as regular objects. This means that BabyUML will operate within a stored program object computer. Thus, somewhat whimsically, the *Baby* in the name.

The BabyUML project is essentially experimental. This report is about an experiment where I experiment with candidate high level object structures that can later be used as examples of BabyUML programming. The experiment is implemented in Java.

Section 1.2 introduces an example and *section 1.3* discusses why my usual programming styles doesn't scale to significant problems. The example implementation combines two paradigms, MVC and CDA.

MVC is the Model-View-Controller paradigm discussed in *section2*. *DCA* is the Data-Collaboration-Algorithm paradigm discussed in *section3*. I finally sum up what I have learned from the experiment in *section4*.

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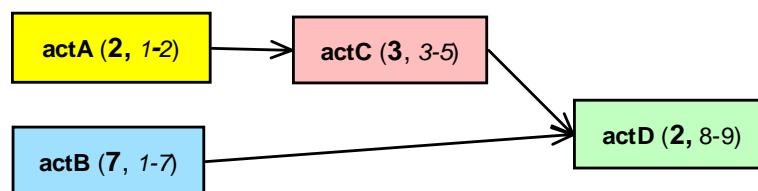
The Java program GUI is shown in figure 2. It's partitioned into four strips. The top strip has three command buttons: Create *First network* (the one shown in figure 1). *Frontload* the network and allocate resources. Create *Second network*. The second strip shows the dependency network. The third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

Fig. 2: The Java program user interface.
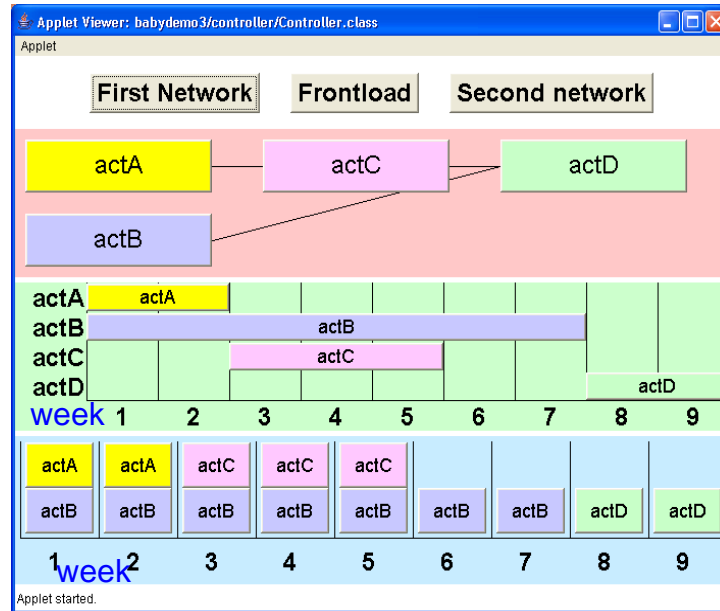
This example could be programmed in many different ways. I use it to illustrate the MVC and DCA paradigms, pretending that I'm working on a non-trivial, comprehensive planning system.

1.3 The Problem

1973 marked our transition into a new and more complex world where *communication* added a new dimension to my challenges. O*bject orientation*, a technology created by Nygaard and Dahl, appeared to be a viable technology for meeting the new challenges. The beginning was very promising, but OO hasn't as yet delivered what I need. I claim the reason to be that the scope of current OO programming language is too narrow with their focus on isolated objects.

Figure 3 illustrates how I would normally implement the Java example. The rounded rectangles denote objects, the solid lines denote links between them, the white rectangles denote classes, and the dotted arrow denotes «instanceOf».

The activity objects are shown bottom right with heavy outlines. The idea is that planning is realized by negotiation; internally between the activity objects themselves and externally between activity objects and their required resources (manpower, machinery, etc.) The technicalities of the user interface have been separated from the domain objects; the GUI objects are shown on the left.

My normal implementation style tends to give fairly small objects in a distributed structure and with distributed control. It leads to a large number of crisscrossing links and complex interaction patterns. An activity uses a certain resource; let the activity object negotiate directly with the resource object to establish a mutually acceptable schedule. A symbol on the computer screen represents a certain activity; let the symbol object interrogate the activity object to determine how it is to be presented, and let the activity object warn the symbol object of significant changes. A significant system can resemble a bowl of spaghetti.

Fig. 3: A typical application.

Resources

GUI

actA **actC**

actB **actD**

Activity Network

class Activity
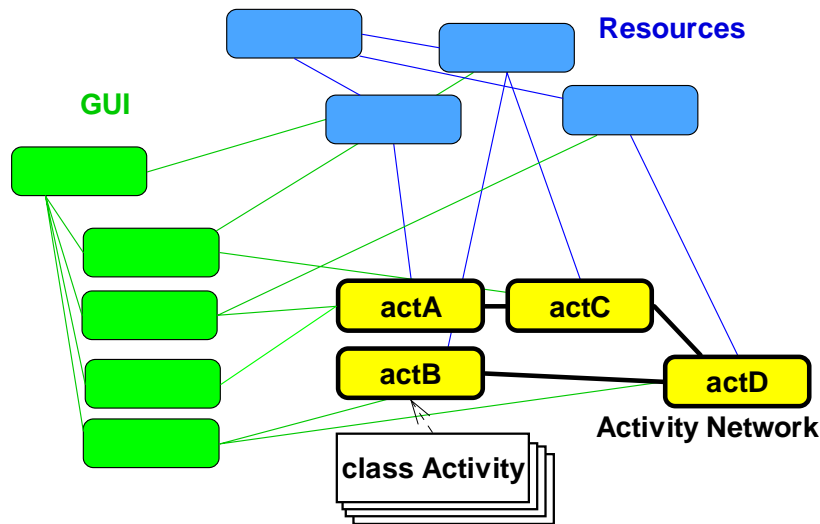
Every object is an instance of some class written in a language such as Simula, Java, or Smalltalk. The structure and domain logic is distributed among the methods of the classes with their superclasses; effectively fragmenting the bowl of spaghetti into a dish of noodles. The system as a whole is nowhere to be seen.

The most interesting part of a system is the communication that takes place in the space between the objects. This is where my current style and language fail miserably. I need to replace the dish of noodles with a clear and manageable declaration of structure. I need to pull essential information out of the noodles to define object interaction in explicit and readable code. I simply need a bird's-eye view that separates the code that defines the whole from the code that defines the details.

1.4 The Java experiment

In the Java experiment, I explore two structuring paradigms that explicitly answer the critical questions: What are the objects? How are they interlinked? How do they interact?

The first is the old MVC paradigm that differentiates objects according to their purposes in an application:

- A *Model* is responsible for representing the domain information.
- A *View* is responsible for presenting model data to the user and to receive input from the user, thus bridging the gap between the model data and the user's mental model.
- A *Controller* is responsible for setting up and coordinating a number of related views. The Controller is sometimes called a Tool.

The second generic structure is the new DCA paradigm that explicitly describes high-level system features. The object space is partitioned into *Components*, where a component is an encapsulation of a number of *domain objects*. A component is characterized by its provided and required operations; its internal structure is invisible from its outside. The component inside is organized as follows:

- The *Data* part. The domain objects and their structure are organized in a "micro database", ensuring the data integrity and providing explicit code for the data conceptual schema.

- The *Collaboration* parts. A Collaboration is similar to an external view in database terminology. It links to a subset of the member objects that interact to realize an external operation, explicitly naming the role they play in the interaction.
- The *Algorithm* parts include explicit code for the object interaction.

I claim that the DCA pattern does scale, even if I do not have a good example as yet. Further, the DCA separation of concerns simplifies the code, makes it more readable, and results in bounded chunks of code that can be audited by separate readers.

The programs in this experiment are written in Java. The resulting code fills more lines of code than a conventional implementation where high-level code for object structure and interaction is embedded in code for low-level, local issues. But this is exactly the code that tends to be unreadable and doesn't scale, see *section 1.3*.

It may be possible to compress the code by applying programming languages and IDEs that are better suited to the purpose. The languages may be multi-media; database design has been done graphically for years. I see no reason why programming should insist that all code shall be textual when all other application domains use graphics whenever it is better for man-machine communication.

2 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 in *section 1.2 on page 4*. A manager was responsible for a part of a large project. His department had its own bottlenecks and its own considerations for planning. Other departments were different; a pipe shop was very different from a panel assembly line which was again very different from a design office. How could each manager have his own specialized part of the planning system while preserving the integrity of the plan as a whole?

The answer was to replace the "dead" activity records in traditional, procedure oriented planning systems with interacting objects. The objects would represent their owners within the universe of interacting objects. The objects could be specialized according to the needs of their owners, yet they could all interact according to a common scheme.

MVC was conceived to bridge the gap between the users' complex mental models and the information stored in the computer. The idea is illustrated in figure 4.

Fig. 4: Bridge the gap between the user's mind and the stored data.
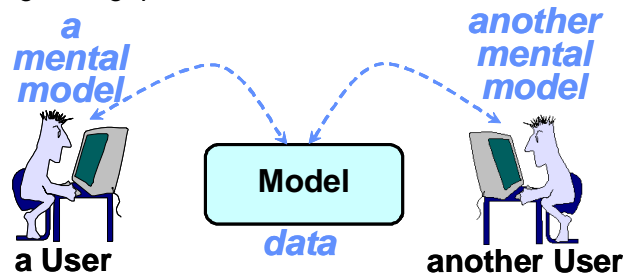
I implemented the first MVC while being a visiting scientist with the Smalltalk group at Xerox PARC. The conventional wisdom in the group was that objects should be visible and tangible, thus bridging the gap between the human brain and the abstract data within the computer. This simple and powerful idea failed for the planning systems for two reasons. The first was that a plan was a structure of many activity and resource objects so that the focus on a single object at the time was too limiting. The other was that users were familiar with the planning model and were used to seeing it from different perspectives. The visible and tangible object would get very complex if it should be able to show itself and be manipulated in many different ways.

2.1 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 example, the Model is the data representing the activity network and the resource.

The Model data may be considered *latent* because it needs to be transformed to be observable to the user and related to the user's mental project model. We will discuss the Java code linking the Model and the View-Controller pair in *section 3 on page 14*.

2.2 The View

The View transforms the latent Model data into a form that the human can convert into *information* as illustrated in figure 5.

The MVC triad could be implemented in a single class. If this class turns out to be complex, it can be simplified by a separation into 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. 5: The View couples model data to the information in the user's brain
so that they appear fused into one.
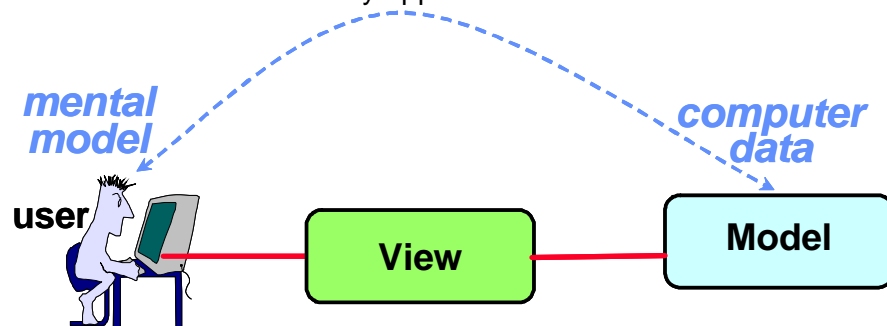
I will discuss the Java implementation in *section 3.5 on page 17*.

2.3 The Controller

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

Fig. 6: The Controller creates and coordinates multiple Views
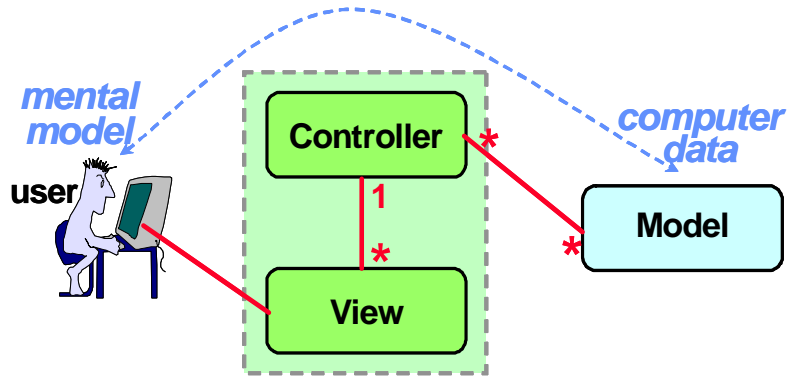
Note that the Smalltalk 80 Controller is responsible for input and thus different from the one discussed here.

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.

2.4 The anatomy of the Java user interface

The Java tool is shown in figure 7. 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 period.

Fig. 7: The anatomy of the MVC Java tool.
(: ButtonStrip means an instance of class ButtonStrip).
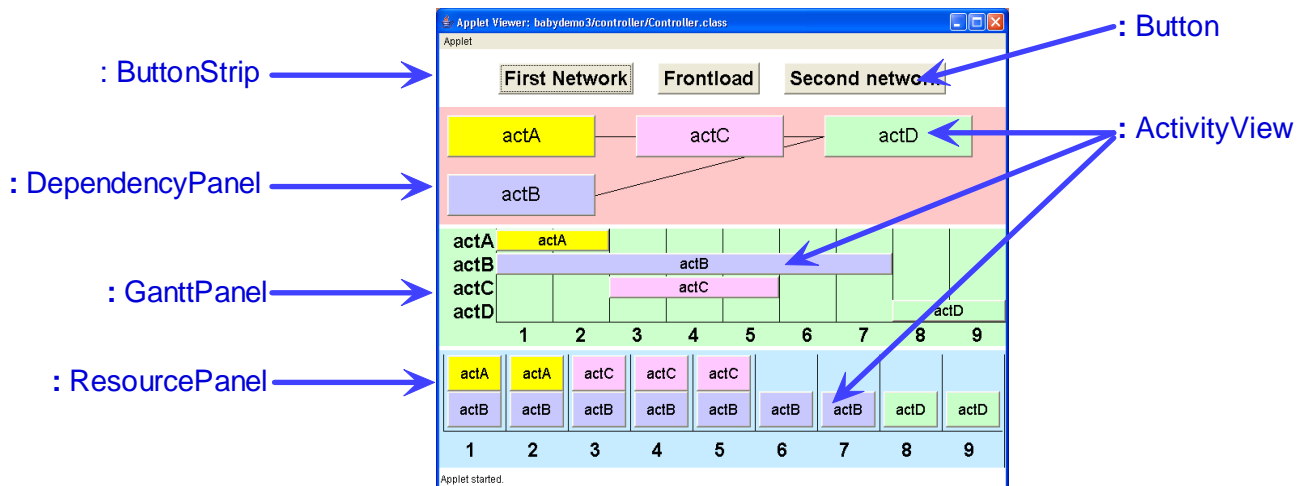
An overview of the implementation is shown in the class diagram of figure 8. We see the classes described above and their main associations.

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 8.

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. 8: Java class diagram
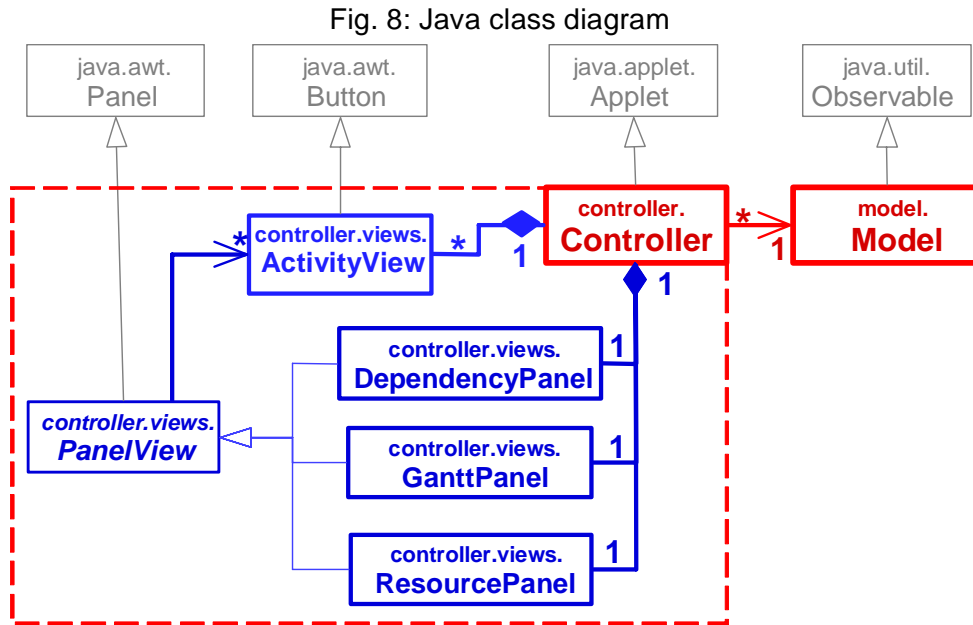
The component structure is reflected in the package structure as illustrated in figure 9.

Fig. 9: The Java package diagram.
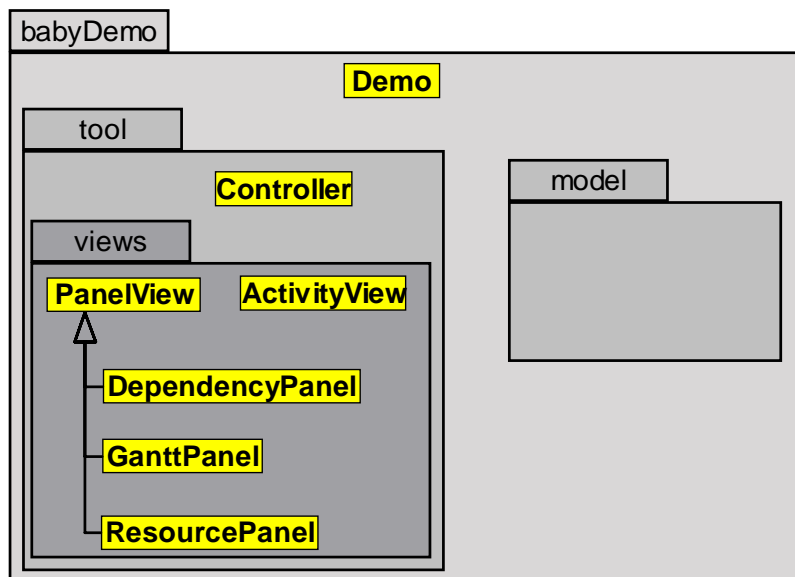
We will go into the details when we discuss the model internals and system behavior in *section 3 on page 14*.

2.5 Controller coordinates selection

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

Fig. 10: actC is selected in all views where it appears.
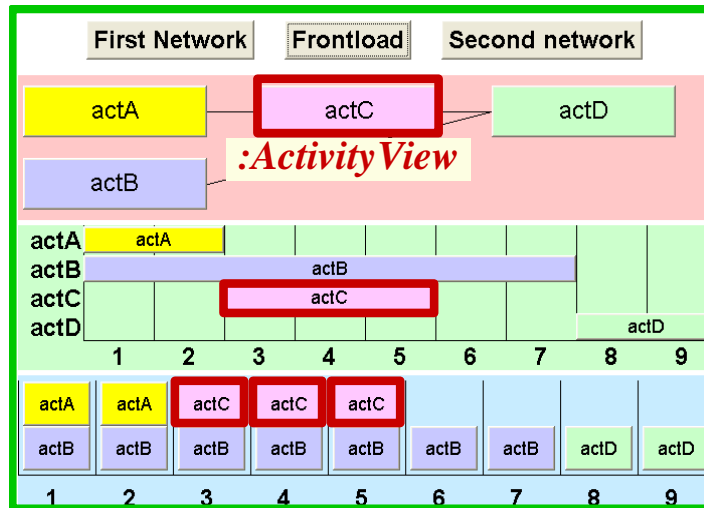
Figure 10 shows the tool after the user has clicked on any of the actC activity symbols. The key to simplicity and generality is that the view being clicked only reports this event to the Controller. The Controller decides that this is indeed a selection command, and that it shall be reflected in the appearance of all activity views. This behavior is illustrated in the sequence diagram of figure 11.

Fig. 11: The selection interaction.
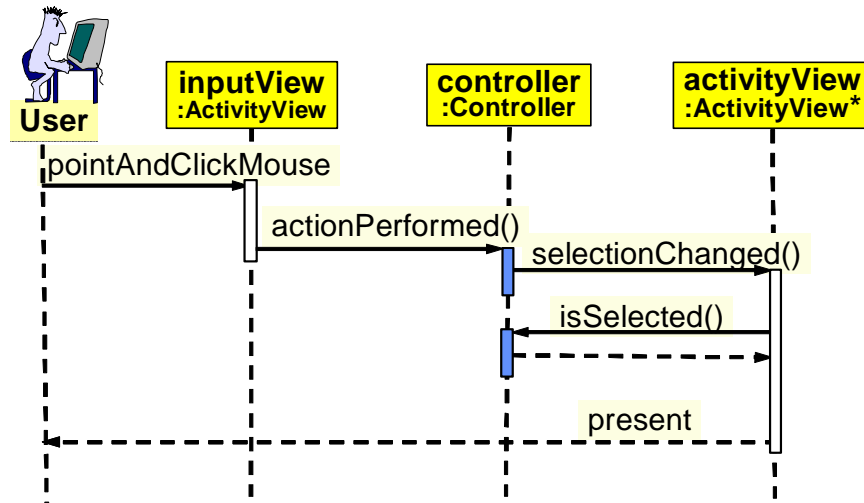
The inputView role is shown to be some instance of class ActivityView. We see from figure 8 that ActivityView is an awt.Button, so it sends an actionPerfomed event to its actionListener. All activityViews are created to let their actionListener be the Controller:

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

```
(Java 8)    }
```

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

```
(Java 9)    ActivityView>>public void selectionChanged() {
(Java 10)       if (controller.isSelected(activity)) {
(Java 11)           setBackground(activity.color().darker());
(Java 12)       } else {
(Java 13)           setBackground(activity.color());
(Java 14)       }
(Java 15)  }
```

and

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

2.5.1 Discussion: Observer pattern or direct access between controller and views?

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 could 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.

2.5.2 Discussion: The panels could be subordinate controllers.

We see from figure 8 on page 11 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 programmer of the top level controller would then not need to know the inner workings of the panels. We did not choose this solution because the responsibility for activity selection is anchored in the top level controller. The structure is known at the top, so there is no reason to complicate the code to attain a fictitious flexibility.

3 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 represent all activities and resources 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, characterized by its provided operations. Inside, there is a well ordered and powerful object structure partitioned into three parts, *Data*, *Collaborations*, and *Algorithms*. The *Data* part. is a set of objects that instantiate the domain conceptual schema. The *Algorithm* part is a set of methods that specify domain object interactions for all provided operations. The *Collaboration* part is a set of classes specifying external data schemas that provide access methods tailored to simplify data access in the Algorithms.

3.1 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 12(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 its methods. The Java object is different; the fields *are* visible from the outside. I write x = foo.fieldX; to access a field directly, and I write x = foo.getFieldX(); to access it through a method.

Figure 12(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). 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() could be a derived attribute. Other methods implement the object's provided operations.

Fig. 12: (The regular object is an instance of a class.
a) The object as depicted in the Java tutorial. (b) A more accurate object model.

<div align="center">(a) (b)</div>

1.http://java.sun.com/docs/books/tutorial/java/concepts/object.html
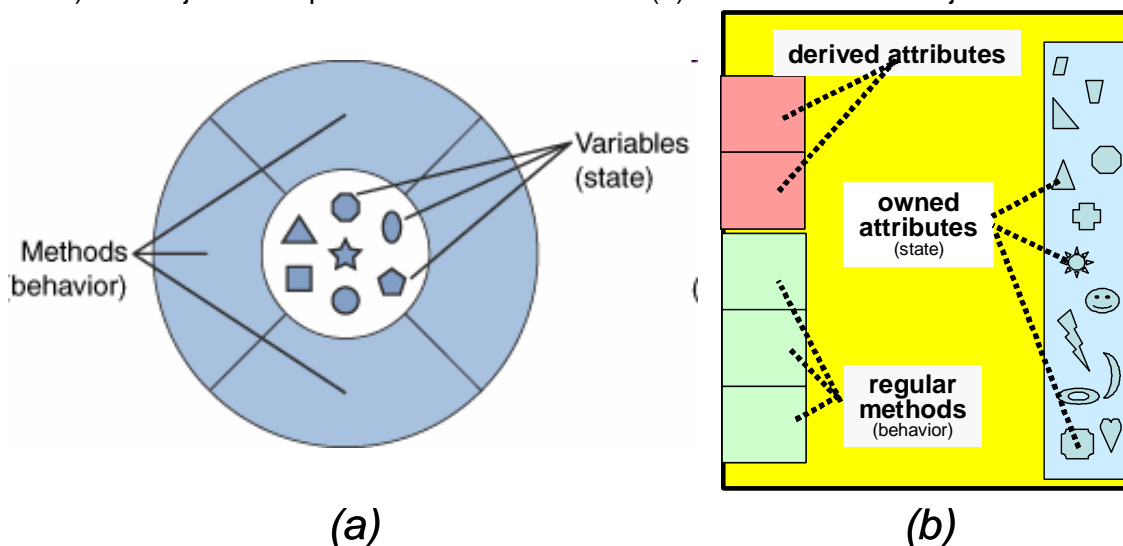
3.2 The DCA Component; a well-structured monster object

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.

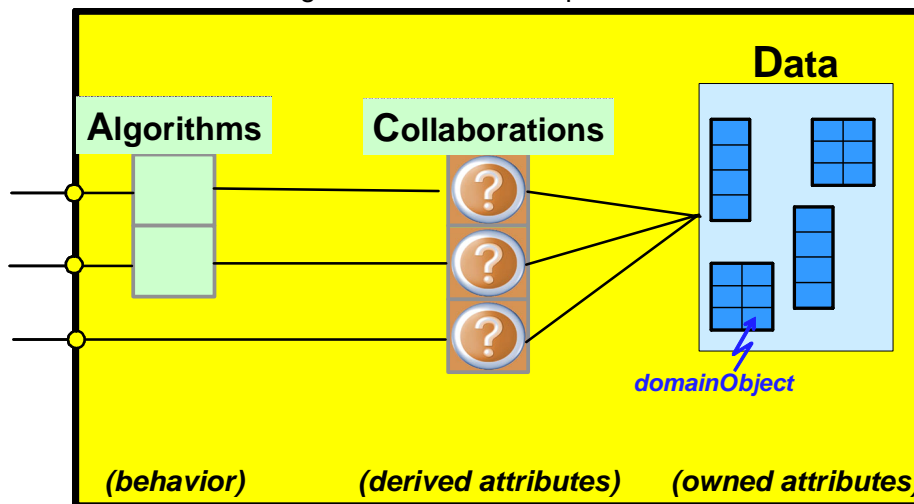Fig. 13: The DCA component.

Figure 13 illustrates the DCA component as an object that encapsulates other objects. The DCA component looks like the object of figure 12 when seen from its environment. Inside, we find a number of specialized parts:

¤ **Data**. This corresponds to the owned attributes of the regular object. The fields are replaced by a "baby database" that holds the component's domain objects and their structure. The term "database" is used in a restricted sense, it is a set of domain objects organized according to a conceptual schema. We do not assume persistence, concurrency, access control, security, or any other goodie usually associated with databases. We just call it the *Data*.

 • The Data is organized as a number of relations in first normal form, ensuring referential integrity.
 • The values of these relations are the domain objects (including structure objects).
 • The structure is represented in explicit relations. Contrast with my traditional representation where structure information is distributed among the domain objects. The DCA domain objects are correspondingly simplified.
 • The code for the Data part 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 Data conceptual schema may not be ideal for the access requirements of the different uses of the data. The collaborations implement external schemas, each optimized for a particular usage of the Data.

 • In UML, " *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 CollaborationUse 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 experiment, a *DCA Collaboration* is coded as a class that has the collaboration roles as attributes and database queries as its methods. In its abstract form, it is one or more temporary relations that are derived from the base relations by queries.

- The *DCA Collaboration* 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. The DCA Collaboration binds role names to domain objects. This binding is valid in a certain context and at a certain time. It is a kind of dynamic, indirect addressing, The DCA Collaboration corresponds to the UML *CollaborationUse*. The choice of name reflects our focus on objects rather than code.
- Objects using a Collaboration see the base data in a perspective optimized for their needs. 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.

3.2.1 Discussion

The code becomes more readable because the interaction is explicit and the domain object classes are simplified.

3.3 The Java model as a DCA component

The Java object model is shown in figure 14. For illustrative purposes, the Data is separated into two parts. 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*).

The GUI is split into a Controller object and three panelView objects, each with a layout algorithm that creates its display. In addition, the *frontload* command button activates the frontload and resourceAllocation algorithms. All five algorithms are users of the DCA Data and access it through a suitable Collaboration.

In the following, we will discuss the code for the dependencyPanel and the frontload algorithm together with their tailored data access collaborations as illustrated in figure 14.

Fig. 14: The Java model is a DCA model.
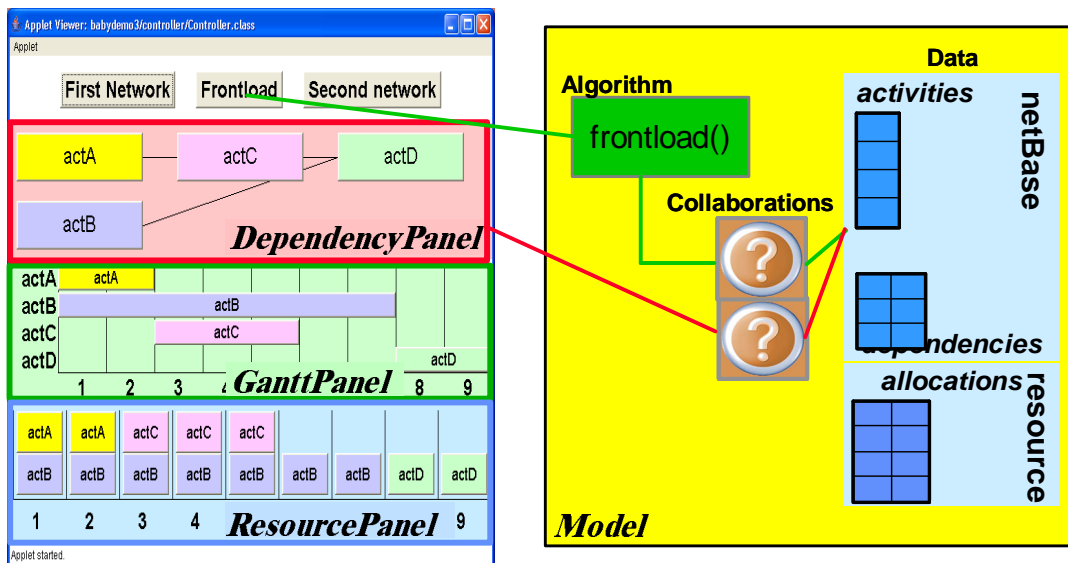
3.4 The Data structure defined by a schema

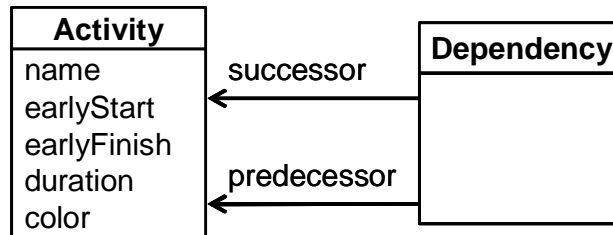The databases are defined by their schemas. Figure 15 shows the netBase schema expresses as a UML class diagram. (An ideal schema language would also name the relations, here *activities* and *dependencies*).

Fig. 15: The *netBase* schema as a UML class diagram.

The corresponding Java class declarations are as follows:

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

and:

```
(Java 25)  public class Dependency {
(Java 26)       private Activity predecessor, successor;
(Java 27)       ...
(Java 28)  }
```

The complete Java code can be found at http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/.

> *Comment*: I personally prefer a 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 15 *be* my schema declaration code.

3.5 Example 1: Panel layout

Figure 16 illustrates that the unit on the horizontal axis in the dependency panel 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. 16: The ranked activities
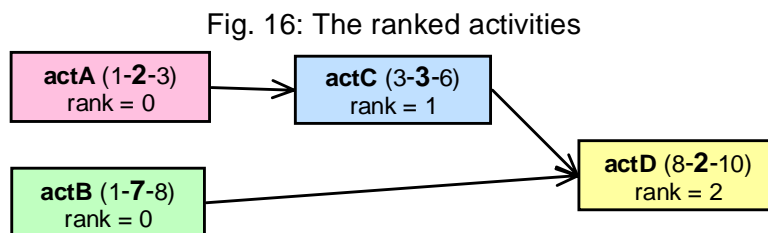
The DependencyPanel layout algorithm is as simple as can be. (Actually too simple, it will often lead to overlapping dependency lines.)

```
(Java 29) private void addActivityViews() {
(Java 30)      Integer gridX = getSize().width / (rankedCollab.maxRank() + 1);
```

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

This layout algorithm 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. The table is accessed through the call to activityListAtRank() in (Java 38) above.

The corresponding code in the RankedCollab class is as follows:

```
(Java 48)  public List<Activity> activityListAtRank(Integer rank) {
(Java 49)      List<Activity> activityListAtRank = new ArrayList<Activity>();
(Java 50)      for (Activity act : netBase. activities()) {
(Java 51)          if (rankOf(act) == rank) {
(Java 52)              activityListAtRank.add(act);
(Java 53)      }
(Java 54)  }
(Java 55)
(Java 56)  private Integer rankOf(Activity act) {
(Java 57)      // Extremely inefficient. Early candidate for cashing.
(Java 58)      // NOTE: A feature of the structure, not an individual activity
(Java 59)      Integer rnk = 0;
(Java 60)      for (Activity pred : predecessorsOf(act)) {
(Java 61)          rnk = Math.max(rnk, (rankOf(pred))+1);
(Java 62)      }
(Java 63)      return rnk;
(Java 64)  }
```

The complete RankedCollab code can be found at
    http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/BabyProject-3.zip.

3.6 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 16 that actA and actB can both start when the project starts, e.g., in week 1. actA then finishes 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 5.

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. The method cannot be called in an activity object before the earlyFinish of all predecessors are known. 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 the 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.

1. 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.
2. The earlyStart of an activity depends on all its predecessors and all modifiers such as activity overlap etc. This logic belongs in the inter-activity space and is here coded in the FrontloadAlgorithm class.
3. 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.

3.6.1 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 have chosen 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 changing 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:

```
(Java 65)  public Activity frontloader() {
(Java 66)      for (Activity act : netBase.activities()) {
(Java 67)          if (act.earlyStart() == null) {
(Java 68)              Set<Activity> predSet = predecessorsOf(act);
(Java 69)              if (areAllDone(predSet)) {
(Java 70)                  frontloader = act;
```

```
(Java 71)                   return(frontloader);
(Java 72)              }
(Java 73)          }
(Java 74)     }
(Java 75)     return null;
(Java 76) }
```

The areAllDone() method called from line (Java69) is shown below:

```
(Java 77) private boolean areAllDone(Set<Activity> actSet) {
(Java 78)     boolean allPredsDone = true;
(Java 79)     for ( Activity pred : actSet) {
(Java 80)         if (pred.earlyStart() == null) {
(Java 81)             allPredsDone = false;
(Java 82)             break;
(Java 83)         }
(Java 84)     }
(Java 85)     return allPredsDone;
(Java 86) }
```

The FrontloadCollab code is not trivial. But it is nicely isolated. I can give it to a colleague and ask her to audit and sign it. (So that an unlikely bug will be her fault, not mine.) The complete code for class FrontloadCollab can be found at

http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/BabyProject-3.zip.

3.6.2 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 obfuscations:

```
(Java 87)  public void frontload(Integer startWeek) {
(Java 88)      // reset all
(Java 89)      for (Activity act : frontloadCollab.resetters()) {
(Java 90)          act.setEarlyStart(null);
(Java 91)      }
(Java 92)      // frontload all
(Java 93)      Activity frontloader;
(Java 94)      while ((frontloader = frontloadCollab.frontloader()) != null) {
(Java 95)          Integer earlyStart = startWeek;
(Java 96)          for(Activity pred : frontloadCollab.frontPredecessors()) {
(Java 97)              earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1);
(Java 98)          }
(Java 99)          frontloader.setEarlyStart(earlyStart);
(Java 100)     }
(Java 101)}
```

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

> Comment: It is clearly an overkill to use the full power of DCA for this extremely simple example. We defend its application here because it illustrates the principles without confusing them with the real complexity of significant programs.

4 Summary and Discussion

The Java experiment reported here is one in a series in my *pursuit of the utmost simplicity.* The current experiment illustrates a number of simplifications. The *MVC paradigm* is a well known simplification with its separation of tool from substance. The *BabyUML Component* is a "monster" object that encapsulates other objects, Many remote links are removed, other links are hidden and the overall structure is greatly simplified. The *DCA paradigm* is new, but looks promising for selected applications where its strict separation of concern can lead to simpler and more readable code.

4.1 The MVC paradigm

The Model-View-Controller paradigm (MVC) 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 the data through an MVC *View*. The view shall ideally match the human mental model, giving the user the illusion that what's in his mind is faithfully represented in the computer.
- The MVC *Controller* creates several related Views and coordinates their actions.

MVC is an old paradigm that has survived for more than 30 years. Its fundamental quality is that it separates model from view, i.e., tool from substance. The ideal Model is pure representation of information, while the ideal View is pure presentation. The separation of Model and View is applicable

- when the Views are 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 separate Controller is applicable when the user needs to see several Views simultaneously.

In practice, it may not be obvious what should go where; in the M, in the V, in the C? Should input checking be in the view or in the model? I put it in the view if it is based on a rule that is independent of the model data; in the model otherwise. The general rule I have used in several systems is to let the model handle all persistent data and let the views be generated automatically.

A problem arises where the views cannot be generated automatically. An example could be the layout algorithm of our example dependency panel (see figure 7 on page 10). 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 is 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 persistent database in the tool component. The views become obsolete if the data is changed without a corresponding change i the manual layout. I have dealt with this problem by letting the view clearly show that it is obsolete, e.g., by using a distinctive background color. The user can then update the layout whenever it is convenient.

4.2 The BabyUML Component

In the current Java experiment, the MVC and DCA implementations are both subordinated the overriding idea of partitioning the object space into *components*. The BabyUML component is a "monster object" that looks like a regular object in its environment where it is completely characterized by its *provided interface*. Inside, the component implementation can obey different paradigms; here exemplified by MVC and DCA. We have also discussed programming he component in the traditional style with extreme distribution.

The notion of a BabyUML 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.

Note that the *BabyUML component* is a run time structure of objects. Contrast with the UML Component that is a kind of class and thus a build time notion. The BabyUML component is somewhat like the *UML CollaborationUse* (an instance of UML Collaboration). But of course, the UML ComponentUse is a model element, while the BabyUML component is a run time occurrence.

4.3 The DCA paradigm

The DCA paradigm is new and untested in practice. It is an overkill in this simple example, yet it seems to be applicable in many situations. I have here illustrated the idea by implementing the MVC Model as a DCA Component

- *The D stands for Data.* Like any object, a DCA 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.
- *The C stands for Collaboration.* A DCA *collaboration* is useful when the data schema is unsuitable for users of the data. Users access the data objects through the roles these objects play in the particular usage. Collaborations bridge the gap between usage and data by dynamically binding roles to domain objects. A role can be seen as indirectly addressing a domain object, making it possible to address different objects at different times without changing the usage code. The notion of Collaborations is derived from the OOram role model and corresponds to the external views used in database technology.
- *The A stands for Algorithm.* Algorithms occur in two places in the DCA paradigm. Some are local to the domain objects and are coded as methods in the domain class. Other algorithms describe domain object interaction and is a property of the inter-object space. The interaction algorithms are coded in separate classes, distinct from the domain classes. This ensures that object interaction is specified explicitly and makes it easier to check the code for correctness and to study its operation.

The behavior of a component is defined by its external operations. Each operation is accomplished by an interaction between some or all of the domain objects. For each operation, we need to answer the following three questions:

1. What are the involved 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 that 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. A Collaboration binds each role to an actual domain object by a suitable query. The Collaboration is another simplification of the system description. The collaboration can be understood from its declaration and used independently of any complexity in the base data.

The member object interaction 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.

4.4 What's next?

I claim that the MVC and DCA paradigms in many cases lead to simpler and more readable code. DCA has as yet only been applied to the current simple Java program. 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 used.

The greatest weakness of the current example is the size of the code. Johannes Brodwall wrote a version of the 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 in size 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.

I have still not touched the BabyUML goal of increased leverage. The next step is to look at the code itself. Given the MVC and DCA paradigms, can we make the code more expressive, saving volume and increasing readability? I see three points of attack:

¤ *More powerful superclasses.* An obvious start. There might be something in the ODMG or JDO packages that could be used. But they could represent an overkill since we do not always require persistency and multi-threading.

¤ Better IDE. It could be interesting to use Eclipse to explore more powerful programming tools.

¤ New declarative languages, new metaclasses, and new IDEs that fully support the MVC, DCA and other paradigms.

I plan to continue my experiments in Smalltalk in order to explore the last alternative.

5 References.

[Arisholm-04]	Erik Arisholm Dag I.K. Sjøberg: *Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software.* IEEE Transactions on Software Engineering, **30**, 8 (August 2004)
[Bluebook]	Goldberg, Robson: *Smalltalk-80, the language and its implementation.* ("The Blue Book"). Addison-Wesley, Reading 1983. ISBN0-201-11371-6
[Coplien-98]	Coplien, James: *Multi Paradigm Design for C++*, Addison-Wesley Professional, 1998, ISBN: 0-201-82467-1
[GOF]	Gamma et.al.: *Design Patterns.* Addison-Wesley, Reading, Mass. 1995. ISBN 0-201-63361-2
[Hay-03]	Hay, David: *What Exactly IS a Data Model?* DM Review Magazine, February 2003
[HiNC]	Trygve Reenskaug: Applications and Technologies for Maritime and Offshore Industries .Springer Boston 2005. ISSN 1571-5736 (Print) 1861-2288 (Online). Pages369-390. Last draft at http://heim.ifi.uio.no/~trygver/2003/HiNC/hinc-18final.pdf
[Hoare-81]	Charles Antony Richard Hoare: *The Emperor's Old Clothes.* 1980 Turing Award lecture. Comm.ACM 24, 2 (Feb. 1981)
[IFIP]	*IFIP-ICC Vocabulary of Information Processing.* North-Holland 1966.
[ODMG]	Cattell, Barry: The Object Data Standard: ODMG 3.0. Academic Press, London, 2000. ISBN 1-55860-647-4 (http://www.odmg.org/)
[Ree-63]	Hysing, Reenskaug: *A System for Computer Plate Preparation.* Numerical Methods Applied to Shipbuilding. A NATO Advanced Study Institute. Oslo-Bergen, 1963.
[Ree-73]	Reenskaug: *Administrative Control in the Shipyard.* ICCAS conference, Tokyo, 1973. (http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf)
[Ree-77]	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
[Roles]	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)
[UML]	*Unified Modeling Language: Superstructure.* The Object Management Group 2006. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

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.