Model-View-Controller and Object Teams: A Perfect Match of Paradigms

Matthias Veit
Fraunhofer FIRST, Berlin
Matthias.Veit@first.fraunhofer.de

Stephan Herrmann
Technical University Berlin
stephan@cs.tu-berlin.de

ABSTRACT

From the early days of object-oriented programming, the model-view-controller paradigm has been pursued for a clear design which separates different responsibilities within an interactive application. In contrast to its untyped implementation in Smalltalk, any implementation in a statically typed language involves certain trade-offs which either blur the clear structure, destroy the intended independence, or introduce undue administrative overhead. Each alternative creates a different caricature of the originally crisp architecture. The programming model Object Teams provides a new modeling unit called Team plus a number of binding mechanisms by which a Team and its contained roles can be bound to existing parts of an application. It turns out that these mechanisms fit nicely for implementing a clear model-view-controller design not only for single elements but also for complex structures of GUI elements plus their binding to complex model structures. We furthermore propose to use the model-view-controller paradigm as a benchmark for AOSD approaches, since it combines a set of typical problems concerning the separation and integration of concerns.

Keywords

GUI design, a-posteriori integration, composition, collaborations, programming language, evaluation

1. INTRODUCTION

Long before the dawn of aspect-orientation, separation of concerns for user interfaces has been addressed by research at very different levels. Most prominently, the model-view-controller paradigm [8] (MVC for short) defines an architectural pattern, which ensures a good degree of decoupling if applications strictly adhere to its constraints. Other work defines architectures at an even larger scale, like the Chiron [12] system, which also supports distribution of its components. Libraries — frequently addressed as toolkits — apply common principles of object oriented design, notably a wide range of design patterns, for providing reusable elements of user interfaces. Examples are Interviews, Motif and Swing. Usage of such libraries employs a mixture of inheritance and aggregation. GUI-builders are tools, which support the interactive assembly of user interfaces. These tools usually generate source code. Also trigger mechanisms as present in some databases are used for implementing part of the link between an application and its user interface. Some of these approaches also enhance a given programming language for convenient development of user interfaces.

All in all, separation of concerns for user interfaces can be said to be a well explored issue. We see two reasons for re-considering this topic:

1. Practical development of user interfaces is still subject to compromise between several goals that haven't been reconciled yet. Each of the solutions mentioned above has specific draw backs when they are put into practice in the real world.
2. The conceptual understanding of the issues in separating concerns for user interfaces makes this a well suited *benchmark* for approaches to aspect-oriented software development, which claim to improve modularity and separation of concerns.

Taylor and Johnson [12] give a very good description of why many approaches that are sound concepts on their own fail under real world conditions: *"Current user interface technologies often impose a fairly rigid set of constraints on how the application must be structured. In a large application, however, there are typically many architectural desiderata, so it is undesirable for the user interface to enforce a particular architecture."* If specialized technology fails because it conflicts with other requirements, it should be worth consideration how close to an optimal solution for user interfaces we can get using general purpose techniques only. On the other hand the MVC architectural pattern can also be seen as an archetype in module decoupling and integration.

Thus, we propose to regard the realization of an MVC architecture as a benchmark for approaches for AOSD. This paper reports on our results in using a recent AOSD approach called *Object Teams* [6] for the development of user interfaces and investigates the gains in decoupling and reusability that are achieved by this approach. We claim that Object Teams not only technically provide a basis which very well fits the task at hand, but also the metaphors behind Object Teams correspond very smoothly to the MVC paradigm, such that both "paradigms" support each other very well.

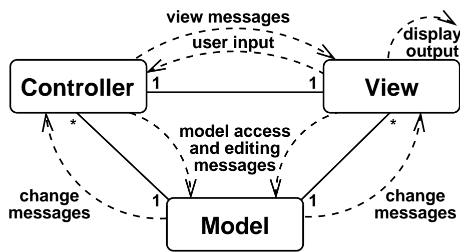

Figure 1: MVC – multiplicities and message flows

Sect. 2 will recap the MVC architectural pattern and point out some weaknesses. Sect. 3 introduces Object Teams in a nutshell. In Sect. 4 we present how MVC can be realized using Object Teams. For this presentation we use two examples: a simple stop watch and a class browser. Sect. 5 presents some details of Object Teams that make a difference in realizing MVC. Sect. 6 evaluates the presented solutions. We conclude with a discussion of related work (Sect. 7) and an outlook (Sect. 8).

2. MODEL VIEW CONTROLLER

Almost all modern graphical user interface (*GUI*) widget libraries rely on the distinction between model, view and controller [8] (see Fig. 1). A model in this paradigm is a class which originates in a specific domain. It is an abstraction of a domain specific entity and has no knowledge about the GUI. The representation of the model as GUI element is called view. A view can be seen as a wrapper around the model, which is capable of displaying a subset of the data that is encapsulated in the model. Each view has an associated controller. A controller is responsible for all possible actions that are defined in the view concerning the associated model. A model can have multiple views, where the views among themselves can differ. This is not surprising because there are many possibilities to represent the same set of data. The encapsulation of GUI-specific details inside a view and the management of the view inside a controller lead to a clean code base of the domain specific model.

The model is built without any knowledge about views and controllers. There is an implicit association via an observer mechanism [4, observer pattern]. The model must implement the functionality defined in the role 'Subject' and introduce notifications in all places where the state of the model is changed. The update message is sent to every attached view or controller. View and controller use an explicit association to the model to query its state. A view can define actions and can send events, if a specific action has happened. The controller is able to listen to specific events of the associated view and has to handle all those events, which have meaning to the associated model. A controller 'translates' from specific GUI events to application logic.

2.1 A simple example

To demonstrate some design principles, a simple example shall be introduced: a stop watch. The concept of a stop watch comprises the capability to start, stop and reset a counter. When running, the stop watch continuously advances this counter. The value of the counter can be queried at any time. A class called `StopWatch` is introduced, which implements the described behavior. The method `start` starts a thread. The thread invokes `advance` at every second and sleeps for the rest of the time until `stop` is called, which will stop the running thread. The method `getValue` returns the elapsed time as an object of type `Duration`. A view class `WatchDisplay` consists of a label which displays the elapsed time and three buttons start, stop and clear. The controller implements methods to handle the events, that are sent by these buttons. To synchronize the model and the view five styles of communication are needed:

1. The display has to register itself as an observer of the stop watch (`View`→`Model`).
2. The stop watch must call notify every time the elapsed time has changed (`Model`→`View`). The only mutator of `StopWatch` is `advance`.
3. The display has to ask the stop watch about the actual elapsed time (`View`→`Model`).
4. If the user presses start, stop or clear the view must send proper events to the controller (`View`→`Controller`).
5. Events processed by the controller have domain semantics. The controller 'translates' those events to domain specific actions, which are delegated to the model (`Controller`→`Model`).

Without loss of generality, interactions Model→Controller and Controller→View are not considered in this paper.

2.2 Weak points of a MVC based design

The strict separation of responsibilities concerning the same entity leads to a very tight structure of the distinct parts that belong together. Evidence for this can be seen in Fig. 1 — each pair of components is linked by a bidirectional communication path.

The model should not be aware of the GUI. To gain abstract synchronization of model and view an observer infrastructure is needed, which in fact is not a functionality of the domain class. The design of this infrastructure requires preplanning in the model. At a first look this seems well tolerable. This includes, however, the definition at which level of granularity changes should be propagated, i.e., which events are available for observation. This may lead to designs of overly eager model classes that broadcast a large number of events, of which only few are used by views. On the other hand, if some changes are initially not considered for notifications the addition of a new view might require the model class to be modified to this new requirement, which again demonstrates an undesirable dependency of the model.

View and controller are developed with an explicit binding to a specific model. Thus, view and controller classes are not reusable for different models. A common solution to circumvent that problem can be found in many widget libraries. To gain reusability of the view and controller, the definition of the model is declared abstract: as an interface or pure abstract class. To apply such views/controllers to a domain class, this class has to be adapted to fulfill the specified contract and type. This leaves two alternatives: either the model must be developed conforming to the abstract model interface as defined by the view library, or object adapters have to be implemented which provide the desired

interface and delegate all requests to the real model. Neither alternative is optimal for obvious reasons.

The problems that arise with a MVC architecture are addressed by special techniques of object-orientation. It is possible to circumvent many problems, but there is a need to tackle the real cause, not just the effect.

The actual nature of the problems at hand might become clearer, if we forget about MVC for a moment and analyze the naïve solution where each model element is also responsible for its presentation and user interaction. Here the implementation of each view is scattered over the set of involved model classes. Views are tangled with application code and if different views use the same model class these views are also tangled with each other. From this it should be clear that views and controllers are aspects, too. Applying aspect technology should also help to improve the modularity of GUIs. Hence, we are seeking aspect-oriented solutions that address the issues of MVC in a more radical way than traditional object-oriented techniques do.

3. INTRODUCTION TO OBJECT TEAMS

Object Teams is a recent programming model, which combines two concepts for improving modularity: aspects and collaborations. The direct predecessors of Object Teams are Aspectual Components [9] and LAC [5]. From this background, Object Teams is a general AOSD approach aiming at a broad range of modularity issues. Before demonstrating the applicability of Object Teams for the MVC–architecture, let us briefly show, how the concepts of collaborations and aspects are refined to teams, roles and different styles of binding.

Aspects are functionalities, that cannot cleanly be encapsulated by means of object oriented techniques. Different aspects with collaborating functionality can be scattered over different classes of a domain. The paradigm of Object Teams introduces a module concept called *Team*, where collaborating aspects can be grouped to one larger entity. An aspect inside a Team encapsulates a behavior, which is specific for its surrounding Team. Such a stereotypical behavior is called 'role behavior'. Aspects inside a Team are therefore called *Roles*. Roles are implemented as classes. A Team can be seen as an outer class, in which all collaborating roles are realized as inner classes.

A role class is marked abstract if it has abstract methods. A Team is marked abstract, if it contains one or more abstract roles. Because a Team itself is a class, inheritance can be applied. Team specialization uses a particular inheritance technique, called *implicit inheritance*: all roles defined in the super-Team are automatically available in the inherited Team. If a sub-Team defines a role with the same name as a role within a super-Team, this is automatically a sub-class of the previously defined role with the option of overriding inherited methods.

An approach that declares separation and encapsulation of scattered functionality in a structured way must also declare how to bring together these separate pieces. Because roles of a Team are incomplete entities which are intended to contribute one aspect to a larger domain entity, Object Teams

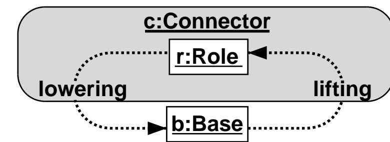

Figure 2: Role to base relation

introduces structured gluing techniques for applying roles to domain classes. For this purpose, a Team can be used as a *connector*. A connector is a refinement of a Team and contains definitions like: the behavior of role class R is applied to the domain class D. Bindings are explicitly denoted in a succinct, declarative style. At method level binding uses the mechanism of forwarding.

Forwarding from an abstract method of a role class to a dedicated method of a domain class, called base, is named *callout*. The base class implements an abstract method of a role class and can be seen as sub-class of the role class. The opposite direction, forwarding from a base to its role, is named *callin*. A callin definition declares an invocation of a role method:

- `before` the base method gets called,
- `after` the base method gets called, or
- instead of calling the base method (`replace`).

The advice of such a forwarding is woven into the base class at runtime. It is possible to connect many role classes to the same domain class.

Another way to call the base from the role exists during a `replace` callin, where the replaced base method is accessible. Here the role acts as a sub-class of the base which overrides an existing method. A new keyword `base` is introduced, which can be used inside such methods. This mechanism is known from overriding, where `super.m()` calls the overridden method in the superclass.

An effective connector (Team) can be instantiated. The activation and deactivation of bindings is based on the activation and deactivation of the enclosing connector instance. If a connector is activated, all callin bindings are woven into the base classes, deactivation removes those advices.

Object Teams introduce a translation called *lifting*, which maps from a base object to a corresponding role object. The translation back to the base is called *lowering* (see Fig. 2). If a callin binding is hit, the base is lifted to its role. The call is forwarded to the specified role method. A role can call its base via callout or in a replaced method using `base`. In this case the role is lowered to its base and the call is forwarded. The process of lifting and lowering during method forwarding includes all parameters and results. It is completely automated: all necessary translations are woven automatically into the system. Such a mechanism ensures a safe disjunction of two worlds: the objects of a Team and the objects of the domain.

In Sect. 4.1 we will show these mechanisms at work using the stopwatch example of Sect. 2.1.

4. MVC À LA OBJECT TEAMS

The paradigm of Object Teams enables a type of binding between classes, that is as powerful as the binding by inheritance but gains much more modularity and flexibility. This binding is as versatile as the binding by (hand-coded) forwarding, but it is supported by the runtime system, which takes care of safety. The paradigm of MVC relies on the binding by forwarding, which is extremely flexible but taken in isolation it is too weak to solve all requirements.

Using the techniques of Object Teams we replace this weak binding by role binding. The idea of perceiving the view as a role of the model enables an equally flexible but more powerful structure. Those bindings lead to better independence between model and view: View classes do not operate on a specific type of model, they declare open spots that are bound by callout. Views remain abstract, customization is done in the connector. Synchronization between model and view is defined by callin binding, any explicit observer structure becomes obsolete. Different views with different synchronization policies are possible — a result of different callin bindings, requiring no change to the model.

Another problem always comes along with forwarding: identity. From an abstract point of view the model and its representation are the same entity, from a structural point of view they are not. Object Teams solve this duality: the model and the view seem to have the same identity, the view in the context of its Team, the model in the domain. The relation is ensured by a connector and the techniques of lifting and lowering. As a result, any attempt to compare a role and a base object will cause either object to be translated to the context of the other object, thus re-establishing identity.

In the next sections we present examples for better support of separation *and* gluing of functionality in the sense proposed by the paradigm of MVC.

4.1 Stop watch with Object Teams

The design of the stop watch example is quite different from the classical solution when the technology of Object Teams is used. A new Team `Watch` is created, where one role `WatchDisplay` is defined. This role drives a simple label and three buttons. The contract for the corresponding model is defined by abstract methods: `stringValue` is used to obtain the string to display, `start`, `stop` and `clear` are called, if the corresponding buttons are pressed. Those methods are necessary for the role to act properly, but the role can not implement them, because it should remain independent and reusable. Conversely, `WatchDisplay` implements the method `update`, which displays the string returned from `stringValue`. The Team `Watch`, however, contains no call to this method, because the events that should trigger an update occur in the model not in the view. Thus, the method has to be bound to all methods in the model, where the displayed value will be changed. Using this style of a-posteriori method binding, the model does not have to support an explicit observer infrastructure. The role class is declaratively complete in the sense that all open spots are explicitly declared using abstract methods. All declared open spots must be implemented, either in a subclass or in the corresponding base.

```
1  Team class WatchConnector extends Watch {
2    class WatchDisplay playedBy StopWatch {
3      // model operations:
4      start → start;
5      stop → stop;
6      clear → reset;
7      // model query:
8      String stringValue() → Duration getValue()
9        with { result ← result.toString() };
10     // observer trigger:
11     update ← after advance;
12   }
13 }
```

Figure 3: Binding view and model using a connector

The Team `Watch` is abstract, because of the abstract role `WatchDisplay`. This role is capable of displaying a string as a label, which can be controlled by three buttons. It does not dependent on any other class. To use this team, four abstract methods have to be defined.

Binding a view to a model is done by defining a connector that refines this Team and binds the relevant methods to a base class. Fig. 3 shows the Team `WatchConnector` that refines Team `Watch` (line 1) and binds the role class `WatchDisplay` to the domain class `StopWatch` (line 2).

To make this team effective, all abstract methods of the role class `WatchDisplay` have to be defined, either by in-place implementation or by forwarding to a base method. All abstract methods are bound to the class `StopWatch`. Callout bindings are defined by $\rightarrow$, callin bindings by $\leftarrow$ plus modifier. The abstract methods `start` and `stop` have equal names as their counterparts in `StopWatch`; `clear` is mapped to `reset` (line 4,5,6). If one of the three buttons in the view is pressed, the bound method of the model is called on the associated model instance. This instance is obtained by the lowering translation, which is automatically inserted by the compiler. Similarly, the abstract method `stringValue` is bound to `getValue` in the domain class (line 8). Since `getValue` returns an object of type `Duration` but `stringValue` needs an object of type `String`, the result has to be adapted. Object Teams allow the adjustment of parameters and results as part of a method binding. Each parameter adjustment maps an expression of the calling level to a formal parameter of the receiver method. `result` serves as a special identifier for method results. In the example, the object of type `Duration` is converted to a string via the standard `toString` method.

Due to these bindings the connector Team is effective, but synchronization of model and view is not yet handled. The view shall be updated every time the model has changed. A change to the observable state of `StopWatch` is only performed by the method `advance`. The callin binding (line 11) ensures an invocation of the method `update` every time `advance` is called. The `WatchConnector` is effective and instantiable. If it is activated, the observer hook is woven into `advance`. Each instance of `StopWatch` can be associated to one or more views.

Lifting comes into play when advance is called while a WatchConnector is active. The runtime system automatically lifts the StopWatch instance to a WatchDisplay in order to invoke its update method. Lifting is performed relative to a given Team instance. By this mechanism the advantages of static and dynamic integration techniques are combined: the role base binding has the flexibility of delegation based techniques but it does not exhibit the administrative overhead, which is usually the price of such flexibility. The key to automating the base-to-role translation is the concept of Teams as contexts for roles. Whenever a StopWatch instance is about to enter the context of a WatchConnector Team, that team is responsible for looking up the correct WatchDisplay role. While several roles of this type may exists for the same base object, this lookup is disambiguated by the current Team.

In Sect. 2.1 five types of communication are described. The communication itself is not replaced, but the style by which it is defined has changed:

1. The observer registration is realized by advice weaving during Team activation (line 11).
2. Update notification is ensured by callin binding (line 11).
3. Value queries on the model rely on callout bindings of abstract methods in the view (line 8).
4. Event invocation uses the callback mechanism of the used widget library (classical).
5. Operation invocation is done using callout bindings of open spots, declared in the view (lines 4, 5, 6).

The communication channels between model and view are replaced by callout and callin bindings. Two classes are defined separately, become connected externally and remain independent.

4.2 Complex Structures

Modern GUI widget libraries have achieved a high degree of reusability. A set of low level widgets can be synthesized to widgets of arbitrary complexity with the help of GUI-builders and code generators — no line of code has to be implemented manually. The binding of these widgets to domain classes usually have several drawbacks, either in maintenance or in type safety. Object Teams support the definition of the interaction between model and view/controller without introducing new structural types, which lead to implementation dependency of the model (see 2.2). The binding takes place neither in the view nor in the model, but in a third entity: the connector. Explicit provided interfaces of view classes and structured specialization via team inheritance ensure easy maintenance. Type safety is ensured by (static) type checking. The creation of a GUI with widgets that are aware of Object Teams should be more natural, elegant, safe and modular than known from classical widget libraries.

As a proof of concept we designed a Team that is able to display tree hierarchies. This Team is applied to a code parser infrastructure. The result is a simple class browser, which allows graphical browsing and navigation of source code, known from typical IDEs. This example is implemented in Ruby [13] using Ruby Object Teams [15] and uses the GTK v1.2 [16] widget library. Since the concept can be transfered to any other language following the Object Teams model, the code examples are written in ObjectTeams/Java [1, 7] to ensure a well-known syntax.

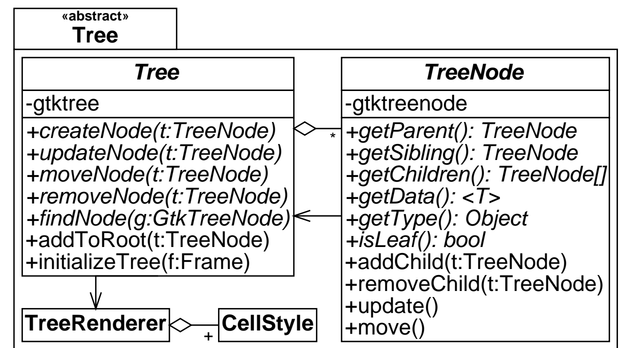

Figure 4: An abstract tree model

4.2.1 The View: Tree Widgets

A graphical tree structure defines two responsibilities, that are encapsulated as roles: a tree as root of all branches and leaves and a tree node, which model the visible elements of a tree. Each graphical representation of a tree node consists of the displayed data and a set of graphical attributes (color, icon, et cetera) that is defined by a specific tree style. The abstract definition of those requirements can be seen in Fig. 4. Classes Tree and TreeNode act as object adapters to their pendant defined by GTK. The API of those classes has several shortcomings that are leveraged here.

A Tree is a widget, which can display a whole tree structure consisting of TreeNode-objects. Because this class is defined in the abstract Team TreeView, only two methods are implemented, but five are declared abstract: each specializing concrete tree style has to implement these methods. By the abstract methods both roles define their *expected interface*, i.e., functionality that must be provided at integration time. This interface is defined in a way that is most appropriate for the TreeView Team. Considerations about those classes that eventually provide this behavior are hardly needed, because we can rely on the mapping capabilities of role binding with callout declarations.

The abstract method findNode has an exceptional position, it is needed only because of the object adapter: the tree has to make a relation between a GTK tree node and a Team tree node — findNode implements this mapping. Each tree uses a TreeRenderer, which controls the appearance of a tree node. To support custom representations, the tree renderer uses CellStyle-objects, which encapsulate a specific representation and are registered under a certain type. This type is matched by the type of a tree node and defines its graphical representation. The class CellStyle must be specialized in every specializing tree style, to support the custom appearance.

A TreeNode is a graphical representation (view) of an entity as item inside a tree. The tree is defined by the three methods parent, sibling and children. The position of a tree node inside a tree is specified by the relation to the

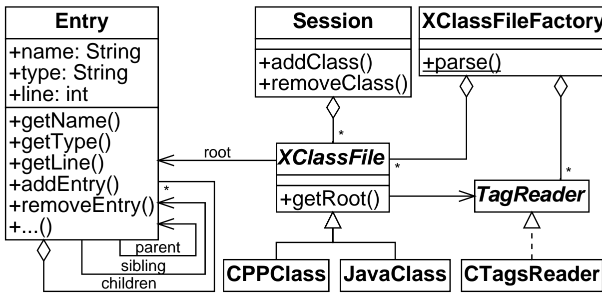

Figure 5: A source code parsing infrastructure

parent and the sibling. The displayed content of the item is fetched via `getData`[1]. The graphical representation, e.g. background color, pixmap, font et cetera, is specified by a specific object returned by `getType`. This type object is used by the tree renderer to specify a cell style. The method `isLeaf` indicates, whether this tree node is a leaf. All those methods are abstract and can not be defined in the view, because this relates to application logic. These methods are needed to correctly paint a tree node. There are four additional methods to synchronize the model and the view via callin: `addChild` and `removeChild` should be called whenever the set of children has changed, `update` should be called, whenever the displayed data has changed and `move` should be called, whenever the position of this tree node has changed.

All roles are grouped into the abstract Team `TreeView`. Two refinements of `TreeView` have been implemented: a normal tree without any graphical additions (`NTree`) and a column tree, called `CTree` — a table like widget, where each entry itself consists of multiple columns. In both Teams, role class `Tree` implement all abstract methods of its super class. The Team `CTree` refines the `CellStyle` to support pixmaps for any state of each tree node and colored text. Both Teams `NTree` and `CTree` remain abstract, because the class `TreeNode` is still abstract. All abstract methods shall be bound to domain classes to work properly, no further refinement is needed.

4.2.2 The Model: A Code Parsing Infrastructure

We have implemented a simple code parsing infrastructure, which is shown in Fig. 5. The details of most classes are omitted. The focus is set on the structure and binding to a graphical view. The process of parsing relies on source code and abstracts from specific programming languages. The abstract class `XClassFile` represents a source file of a specific programming language. Each specific `XClassFile` object has a number of entries found in the file: package, class, method, attribute definitions et cetera are modeled as `Entry` objects. An `Entry` object encapsulates the name and type of the entry plus the line number, where the entry was found. Such entries can build a tree, e.g.: package $\rightsquigarrow$ class(es) $\rightsquigarrow$ method(s). The knowledge of the structure is implemented

[1]The return type of `getData` is used inside class `Tree`. A specific sub Team has to bind the template parameter to its used type. As long as there is no support for genericity in the programming language, this technique can be replaced by an additional role declaration `Data`, which is refined to the needed type for a specific tree.

Figure 6: Architecture of the class browser

in sub-classes of `XClassFile`. To support parsing of different programming languages, different parsers are needed. Every `XClassFile` associates a parser which implements the `TagReader` interface. To match a dedicated `XClassFile` and parser against a specific file type, the `XClassFileFactory` is used. The static method `parse(file:File):XClassFile` is the main entry point. A `Session` is a container, to hold a set of `XClassFile` objects.

The model is a regular package without any Object Teams enhancements.

4.2.3 Binding View and Model

To achieve a graphical representation of a parsed structure, we have to apply a view to the domain package. We have chosen the column tree, where each item consists of the name, the type and the line number. The overall architecture is shown in Fig. 6. The central Team `ClassBrowser` inherits from `CTree` and adapts the `Parser` package by callout and callin bindings. A binding of the Team `CTree` requires a binding of two roles: the tree and the tree node. All classes that have been parsed are kept in a session. To display all classes, we apply the `Tree` role to `Session`. The role `TreeNode` is applied to the `Entry` class, because each entry of a source file should be displayed. This role requires binding of abstract methods: the methods that define the tree structure are implemented with the same names as declared in the abstract role. As definition of `getType` in `TreeNode` we use the equally named method of `Entry`. The encapsulated `type` is a simple string, so it is easy to create cell styles and register them under a certain type that can appear in a class: e.g. "method" or "class". This style is used by the tree renderer if an entry with that type is found. A node is supposed to display the name, type and line number of the corresponding entry. It would be possible to implement `getData` inside `Entry` to fit this requirement. This would result in view functionality tangled within the model. Moreover, the only possibility to declare different views with different content is the implementation of different methods bound to `getData` for each different view. To customize the view for this specific requirement `getData` is implemented inside the connector. The `TreeNode` remains reusable, the `Entry` is not aware of any view functionality — the connector fills the gap for customization:

```
public team class ClassBrowser extends CTree {
  class TreeNode playedBy Entry {
     // declare methods needed by getData
     abstract int getLine();
     abstract String getName();
     // customize content by implementing getData
     String[] getData() {
        return new String[]{ getName(),
                             (String)getType(),
                             Integer.toString(getLine())};
     }

     continued ...
```

```
... class TreeNode continued
        // callout bindings
        getParent → getParent;
        getSibling → getSibling;
        getChildren → getChildren;
        isLeaf → isLeaf;
        getName → getName;
        getLine → getLine;
        getType → getType;
        // synchronize model and view:
        addChild ← after addEntry;
        removeChild ← before removeEntry;
        update ← after { setName, setLine, setType };
        // (no bindings to move )
    }
    class Tree playedBy Session {
        void addToRoot(TreeNode tn)
                    ← after void addClass(XClassFile xf)
            with { tn ← xf.getRoot() };
        void removeNode(TreeNode tn)
                    ← before void removeClass(XClassFile xf)
            with { tn ← xf.getRoot() };
    }
    // Team features omitted ...
}
```

To synchronize model and view both roles have to declare some callin bindings: Every time a class is inserted into a session, all its entries are made visible inside the tree. Before a class is removed from the session, all visible items are deleted. The same kind of logic is applied inside `TreeNode`: every time an entry is inserted beneath a parent entry, this entry is displayed as sub-entry. Deletion works accordingly. For all these bindings, lifting is essential, because the operation of adding an entry to a parent entry is mapped to adding a tree node to a parent tree node. Lifting is applied uniformly to the call target and all relevant arguments of a callin invocation. The callin bindings in class `Tree` need more adjustment, because the base methods `addClass` and `removeClass` have arguments of type `XClassFile`. This class is not mapped within the Team. A simple parameter mapping after the keyword `with` inserts the expression `xf.getRoot()` which yields the root `Entry` of that class. Lifting finally translates the root entry to its corresponding tree node, which is passed as argument `tn` to the role method.

Even if an entry is represented within different views, lifting will always yields the correct role instance with regard to the enclosing Team.

There are three mutator methods inside `Entry`. These methods have to be adapted to also call `update` of the view, since the displayed content has changed. The hook method `move` is not bound to the model, because there is no corresponding functionality in the model. Another application might use this callin method for its desired behavior.

As the definition of Team `ClassBrowser` shows, this Team is designed as a *connector*, which means, that little functionality is added but the main purpose of this Team is to bind classes and methods from the packages that implement model and view. This is the only part of the program that knows about both worlds. Model and view remain ignorant of this integration. Both are independently re-usable.

4.2.4 The Controller

A controller is responsible for mediating between view and model. It must translate view events, which originate from user input, to model operations. It can also influence the behavior of the view in accordance to the state of the model. Such a functional description can have several realizations, which gives rise to a discussion of available design choices. There is one characteristic that all choices share: the controller needs to directly communicate with the role that implements the view functionality. To enable this access, the controller is designed as a part of the Team.

Inline with view. This design directly maps to the Document-View pattern, a special form of the MVC paradigm. This is the easiest form, because the controller is integrated into the view. This is sufficient for most simple tasks. GUI-events are directly mapped to callout methods, as seen in the stop watch example. If a direct mapping requires additional translations, customization can be done in the connector.

Team features. This approach is very similar to the first one and can be seen as pendant to the ‘classical’ controller. All controlling functionalities are centralized in one place and can be refined by standard object oriented techniques. A Team has only access to the view. Specific methods in the domain class have to be declared in the view and must get bound to the model. Because an instance of a Team is always required, no additional controller objects have to be maintained.

Contained objects. It is possible to encapsulate a specific state inside a separate controller, which refers to specific view role(s). These controllers are not bound roles, but are still contained in the Team. As with Team features, these controllers operate only on view objects. Invocations on the model are only possible via callout bindings of the view. Any translation of GUI-events is possible. Such a controller is not accessible outside the Team.

Bound role objects. The most versatile approach is the usage of controllers that share the base objects (model) with their corresponding view roles. Thus, view and controller are roles of the same model inside the same Team. Such a controller can declare its own callout definitions to invoke specific model operations that are needed.

Fig. 7 shows the controller functionality of our class browser. It is implemented as Team features. The constructor of the Team is initialized with a `Session`-object. This session object is lifted explicitly to its role. Object Teams allows an explicit lifting via a special type declaration: `Session` *as* `Tree`. This tree is placed in its own window. To gain full control over the view, additional methods are implemented: The method `show` activates the Team and shows the window, `hide` will hide the window and deactivates the Team. Activation and deactivation correlates with the visibility of the view. The initialization process looks like this:

```
Session session = new Session();
ClassBrowser browser = new ClassBrowser(session);
browser.show();
```

```
public team class ClassBrowser extends CTree {
   // explicit lifting of Session to Tree
   public ClassBrowser(Session as Tree tree) {
      // create window, register for events, setup ...
   }
   void show() {
      activate();
      // connector is activated. show window...
   }
   void hide() {
      // hide window and deactivate connector...
      deactivate();
   }
   void onTreeNodeSelected(SelectionEvent sev) {
      // jump to the specified entry
   }
   // role mappings omitted ...
}
```

Figure 7: Tree controller as Team features

If a `XClassFile` object is put into this session, all entries of that class become visible inside the tree. The user interaction with the class browser is not very complex. The user has the possibility to click on an entry. An associated editor jumps to the position, where this entry was found. The handler for this event is implemented in the method `onTreeNodeSelected`. The event passes the `GTK` tree node which was clicked. The associated role acting as an object adapter is accessible via the method `findNode` of the tree.

4.2.5 Instance considerations

The above code has shown, how a browser and its model (`session`) are associated at runtime. The dynamism of this association is a central feature of the MVC architecture. With Object Teams a base object may play a role in multiple Teams simultaneously. Roles within different Teams do not directly communicate with each other. Of course, modifications of a shared base object are propagated to all its roles in different views. As a result, an Object Teams implementation of MVC supports multiple views (of same or different types) sharing common base objects without further effort.

5. DETAILS OF OBJECT TEAMS

During the development of the class browser example, the fairly young model of Object Teams has been subject to a few enhancements. Please note that none of these extensions concerns the language proper, but all are realized by defining additional methods by with the client program can interact with the runtime environment of Object Teams.

5.1 Team freezing

An active Team usually creates a new role for each mappable base object when a callin binding is hit. This means, that a base class as a set of objects is mapped in full to a set of corresponding role objects. In GUI implementation it is, however, desirable to explicitly map a subset of a base class only. Initializing a window requires to access all base objects relevant for its display. After initialization, a Team can be frozen, which means that no callin binding is allowed to create new role objects for this Team, i.e., a callin binding has no effect for a base object that does not yet have a corresponding role in the Team under consideration.

If a more explicit style is desired for associating objects to a Team the following idiom can be used:

```
public team class ClassBrowser {
   public addEntry (Entry as TreeNode e) { /* empty body */ }
   /* ...*/
}
ClassBrowser cb = ...
cb.addEntry(e1);
cb.addEntry(e2);
cb.freeze();
```

Here the `addEntry` method has the only purpose of causing its `Entry` argument to be lifted to a `TreeNode`, which ensures that `e1` and `e2` have roles within `cb`.

5.2 Sibling role creation

If view and controller are defined as distinct roles over shared base objects, widget setup requires to create a controller role given a view role. For this purpose a controller role may have a constructor of this fashion:

```
public MyController (MyView v) {
   shareBase(v);
   // specific initialization
}
```

The reason for encapsulating this initialization in `shareBase` is, that (for the sake of safety) the program is not allowed to directly access the role-base link, which is completely managed and encapsulated by the runtime system.

5.3 Suspending Teams

The issue of update granularity has been touched already. It is a recurring topic of design trade-offs, since two requirements seem to conflict with each other: (1) each small change at a base object should be propagated to the view. (2) view updates should be bundled if possible to reduce computational overhead and screen flickering. In AOSD a similar problem has been reported as the "jumping aspects" problem [2]. Obviously, observer granularity and join-point granularity refer to the same causes and effects.

The Object Teams model provides a solution using temporary Team deactivation. Recall that callin bindings (advice weaving in AOP terminology) are coupled to Team activation and have no effect if no Team is active. API functions `suspend` and `resume` of class `Team` can now be used, to temporarily deactivate a Team. This can be used to observe a compound operation (say: `addedNodeList()`) and disable observation of its atoms (e.g.: `addedNode()`). The effect is, that an atomic operation only triggers the callin method if not invoked via the compound operation. While the effect is similar to the AspectJ `cflow` construct, we consider activating and suspending more general, since it gives full control to the programmer by a seamless representation of context within the imperative part of the language.

AspectJ programmers might be more familiar with complex conditional pointcuts. In contrast, we feel that the advice of `addedNodeList()` is indeed a suitable place for solving the problem, because this is what actually caused the undesirable aspect re-entrance.

6. EVALUATION

When using Object Teams for developing applications according to the MVC paradigm, the goal of decoupling has been achieved by the combination of advice weaving and role binding.

Callin binding is used to insert triggers into base classes which have not been designed for such notifications. We argue that this is preferable over including observer functionality in base classes not only because base classes should not know about any notification mechanism but also because the granularity of notifications is unknown when designing base classes and even conflicting requirements may exist within the same application. Lifting of call target and arguments automatically manages all necessary instance relations.

Role binding and callout method binding help to bridge the gap between a view that is implemented against specific interface assumptions regarding its underlying model and the concrete model to be connected. This allows fully independent development of both components and a-posteriori integration thereof. This is based on the concept of abstract Teams as *open modules* which are completed not by inheritance but by role-base relations and callout bindings. This binding technique can be regarded as on-demand remodularization of model classes in order to adapt these to their corresponding view classes.

Using both directions of method binding in conjunction with the possibility to hand-code certain translations by methods within a connector we have at hand a well suited set of tools for programming the integration between a model and a view. Such a-posteriori integration gains in importance if UI-builders come into focus. The code generated by such tools should better not be edited manually. Nor should the model be changed to integrate the generated UI. A distinct connector module is a perfect place to implement this integration.

Object Teams provide a good balance of guiding the developer and providing openness and freedom of design choices. The metaphor of roles as views of underlying model elements provides a good intuition on how to come to a good design. An essential concept in a MVC-architecture is the mapping between view elements and model elements. A pure object-oriented solution has to fight a proliferation of objects or needs further mechanisms for managing the associations between the different facets of a common abstraction. The solution using Object Teams can exploit the services provided by the general Team infrastructure. Lifting and lowering transparently manage the role–base association without the need for manual house-keeping.

Despite this guidance by metaphors, the given programming constructs are general enough to allow arbitrary combination of concepts. Our discussion of design decisions regarding controller implementations should have given an idea of how the general concept can be adjusted to different scales and requirements.

Finally, the Object Teams infrastructure has drawn benefit from our case study, since it helped identify some fine points like Team freezing. Such refinements of the Object Teams model do not bloat the language but can be provided simply by adding new methods to the most general class `Team`.

The architectural style of model-view-controller proved to be a multi-faceted challenge for separation and integration of concerns. Only a precise analysis of communications and reuse requirements leads to a solution that doesn't just propagate the intermediate solution based on various design patterns but transcends the capabilities of traditional object oriented programming.

7. RELATED WORK

This paper proposes to use the model-view-controller paradigm as a benchmark for AOSD approaches. So far, only few AOSD techniques have been evaluated for GUI development.

Fidgets [3] emphasize the importance of joint refinements of collaborating roles especially for the context of complex widget structures. Their mechanisms of sibling pattern and constructor propagation have semantics similar to our technique of implicit role inheritance. The main problem addressed in [3] is, however, flexible bindings of widgets to different hardware/OS platforms. The focus of this paper was on integrating the three components model, view and controller.

On-demand re-modularization according to [10] has the same roots as Object Teams. Therefor, it shouldn't surprise that both approaches have similar capabilities of a-posteriori integration of two complex structures, namely view and model. By introducing an explicit interface between a connector and a collaboration [10] suggests to reuse connectors for dynamic composition with different collaborations (Teams in our terminology). The work in [10] only mentions as future plans the integration of our callin (in [10]: callbacks) and activation mechanisms. Also, no practical experience with that model exists.

Outside the field of AOSD, support for MVC is also provided by GUI builders. Such tools either emit source code or declarative descriptors which capture the structure of windows and their widgets. For binding such a generated GUI to a model several alternatives exist:

1. The invocation of model operations has to be hand-coded at specific points within generated source code.
2. Actions can be defined by strings which are evaluated through reflective capabilities of the programming language. Interpreter based environments may simply evaluate the stored string.
3. Actions may be encoded by user defined names (also strings), which have to be interpreted explicitly within client code. This is comparable to the style of properties change listeners of Java Beans.

Option (1) provides no satisfactory separation of a view and its binding to a model. Options (2) and (3) are not type safe because such string based interfaces circumvent the type system. Thus even simple typos cannot be detected by the compiler.

Different approaches for building interactive applications have been mentioned in the introduction. A good discussion on this behalf can, e.g., be found in [12].

8. STATUS, SUMMARY AND FUTURE

Object Teams [6, 11] have their roots in Aspectual Components [9]. Two prototypical implementations exists. LAC [5] was the first practical demonstration, that the concepts proposed in [9] can in fact be implemented and operate as expected. After further refinements of concepts and terminology, Object Teams have been implemented on the basis of Ruby [13]. This prototype has a fully functional integration to the GTK widget library. Most of the experience reported in this paper has been gathered using Ruby Object Teams and GTK. Implementation of a Java-based compiler and runtime system for Object Teams is progressing and the examples in this paper have also been elaborated using ObjectTeams/Java [1, 7] and the Swing library.

This papers has shown, how the technical requirements of the model-view-controller paradigm can easily be mapped to constructs of the Object Teams model. Object Teams achieve a level of independence between those three components, that, to the best of our knowledge, has not been achieved before in a statically typed language. Such independence is possible due to a-posteriori binding of classes and methods. It is essential to support method bindings in both directions, callout and callin. Consistent refinement of complex widget structures and their application to complex model structures are achieved by the module concept of Teams.

Aside from technical considerations, it turns out, that the metaphor of role objects within a context, which are views of underlying base objects, is extremely helpful for designing good architectures in the style of MVC. This confirms our impression that Object Teams are not only a powerful combination of techniques, gathered from the research of many different groups and researchers (see [6] for a discussion), but also provide an appropriate metaphor which enables developers to naturally use the new mechanisms without thinking in terms of mechanisms only but in terms of powerful abstractions that hide much of the complexity of the underlying execution model. On the other hand, the examples shown in this paper also report that a system built with the said abstractions does not exhibit any unpleasant surprise, i.e., the metaphors are also sound and imply semantics that are actually precisely met by the implementation.

A first medium sized case study is currently in progress [14], where a project management information system is being developed, using Ruby Object Teams. It evaluates the capabilities of a Team-based design, which separates common aspects like persistence, access control or GUI visualization from domain code. The user interface of that system consistently applies the MVC paradigm in the style presented in this paper. The possibility of encapsulating different crosscutting (large scale) aspects and their synthesis in a domain model shows the versatility of Object Teams which appear ready for real world problems. This kind of case study helps to identify remaining open issues. Features like Team freezing demonstrate how advanced requirements can be met by new API functions of the infrastructure, instead of new language constructs. From such observations, we consider the language ObjectTeams/Java quite stable, shifting the focus of further development and research on infrastructure, tools, practical experience, method and tutorials.

Acknoledgements

We would like to thank Robert Hirschfeld and the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] Christof Binder. Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken. Diploma Thesis (German), TU Berlin, 2002.

[2] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. position paper at the workshop "Aspects and Dimensions of Concerns", ECOOP 2000, June 2000.

[3] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In *Proc. AOSD*, pages 76–85, 2002. ACM Press.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[5] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*, 2001.

[6] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proc. Net Object Days 2002*, www.netobjectdays.org, 2002.

[7] Christine Hundt. Bytecode-Transformation zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/Java. Diploma Thesis (German), TU Berlin, 2003 (in preparation).

[8] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, Aug./Sept. 1988.

[9] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999.

[10] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proc. of OOPSLA'02*, 2002.

[11] Object Teams home page. `http://www.ObjectTeams.org`.

[12] R. Taylor and G. Johnson. Separations of concerns in the chiron-1 user interface development and mangagement system. In *Proc. of INTERCHI'93*, pages 367–374. ACM, 1993.

[13] D. Thomas and A. Hunt. Programming Ruby: A Pragmatic Programmer's Guide, Addison Wesley, 2000.

[14] Matthias Veit. Evaluierung modularer Softwareentwicklung mit "Object Teams" am Beispiel eines Projektmanagementsystems. Diploma Thesis (German), TU Berlin, 2002.

[15] Matthias Veit. Ruby Object Teams. `http://sourceforge.net/projects/robjectteam`.

[16] Gimp ToolKit `http://www.gtk.org/`