Coarse-grained Components from a Color Model

By David J. Anderson

Introduction

I am often asked, what is the relevance of domain modeling in an age of distributed application frameworks and service-oriented architecture? One answer and a simple one is that there is no replacement for good analysis. Domain modeling in color helps you to understand a problem, analyze it thoroughly and communicate it clearly. It gets everyone on the same page. However, if that were all then you would only ever need a simple drawing tool for models and there would be no need for a sophisticated tool such as Together Control Center which keeps code and models in synch with continuous round-trip engineering.

However, as Tate et al [Tate 2003] demonstrate in Bitter EJB, a good component architecture should have a normalized, domain driven, functional architecture within a set of coarse-grained components whilst exhibiting a service-oriented architecture externally between components. This means that domain modeling in color can be used with application framework development and specifically it can be used to design coarse-grained Beans within an EJB framework.

Component Definition Ambiguity

What tends to happen at the beginning of the architecture work on a distributed system is the team makes a high level attempt to define a set of components from a sketchy understanding of the domain. Then in a traditional modeling approach, they will try to define the interface or set of services provided by each component – again this is being done with a sketchy understanding of the domain. Typically, this component definition is done before domain modeling and may in larger companies be done by a different team. It's a top-down approach. The components definitions may later be distributed to geographically dispersed teams who are asked to develop the interior of the component. This technique of defining interfaces is considered a best practice because it mitigates risk and encapsulates the implementation of individual components. It allows for parallel development with low risk. It is based on the underlying assumption that the interfaces do not need to change and that the component boundaries are correctly drawn in the first instance.

What gets many distributed computing projects into trouble is that these component boundaries are often ill-conceived and either become a burden, or a drag on development and an impediment to reuse, or they require refactoring later. Why is this so?

As each team begins to analyze the requirements for their piece of the system in depth, they struggle with understanding precisely what fits within their scope resulting in debate about the responsibilities of their component. Often as they gain a deeper and

Copyright David J. Anderson 2004

better understanding, they decide it is better to delegate some responsibilities to other components. This results in communication across teams and more than likely a refactoring of the agreed interface. This causes confusion, delay and may impact quality, which in turn causes further delay. I've had better results with a bottom-up approach to component definition and this article explains how.

Postponement and Lean Software Development

In manufacturing industry, the concept of postponement is well understood. Postponement means leaving a decision as late as possible. This has been shown in lean manufacturing to minimize inventory levels, and reduce lead times, whilst improving customer satisfaction. Postponement means designing a product such that fickle end user choices can be committed as late as possible such as color, or shape, or fabric, or pattern, or user interface such as language settings. No manufacturer wants to be left with a large inventory of purple widgets because they were caught by a consumer sentiment change and purple is suddenly last year's color.

Mary Poppendieck introduced the principles of lean manufacturing to software development [Poppendieck 2003]. One of those principles is "Decide as late as possible" or stated another way, "Keep options open until the last responsible moment". Poppendieck was specifically referring to software lifecycle processes but the principle can and should be applied to systems architecture. Decisions about architecture deeply affect the likelihood of re-use, of future refactoring, and of flexibility or adaptability. Architectural decisions can and do affect the future velocity of development teams whether they are using an agile or a traditional approach. Postponing architectural decisions to the last responsible moment allows for flexibility, adaptability and responsiveness but most of all it allows the maximum time to acquire the most information about the system being built. More information means less uncertainty and less uncertainty means less likelihood of change becoming necessary later.

The DNC facilitates postponement of component boundaries

One fear held by developers, is that, if they do not do a high level, top down, component definition but simply jump in to a requirements wide domain model, then they will simply be left with one big monolithic system which cannot be sub-divided readily without massive refactoring – and there will never be time for that.

However, the Domain Neutral Component and the color modeling technique are designed for optimal loosely coupled systems which can easily be componentized. With the DNC, every class is treated as a component and every method on every class is a service that class is providing. Through application of the Law of Demeter [Anderson 2004], each class holds dependencies only to its immediate neighbors. In a typical DNC model, that means that most classes hold relationships only to 2 or 3 others. This means that it is easy to define coarse-grained component boundaries later. All that is required is minor impact refactoring to resolve any two-way dependencies. The interface of the coarse-grained component will inherit the methods from the classes on its boundary which are available to classes which now reside in a different coarse-grained component. Figure 1 shows the guidelines for packaging or componentizing a DNC model.

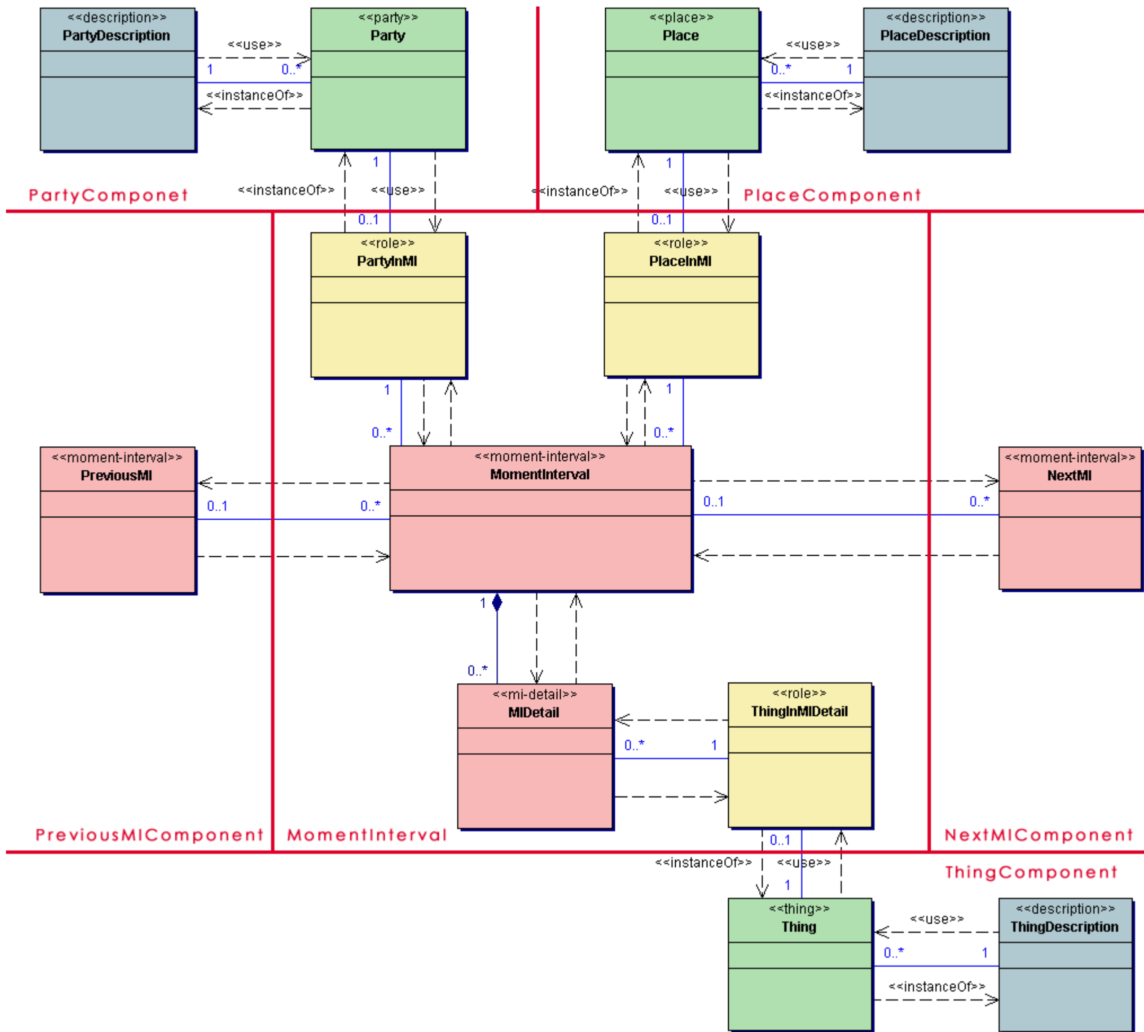

Figure 1. The Domain Neutral Component indicating possible component boundaries

Figure 2. shows how this would look as a UML Component diagram after any two-way dependencies have been resolved. Note that the dependency direction is from the transactional components which contain the <<Moment-Interval>> and the <<Role>>s classes of the <<Party>>, <<Place>> or <<Thing>> components on which it relies to complete a transaction. Any two-way dependencies which might emerge from a sequence of transactional <<Moment-Interval>> components can be resolved with the inclusion of a <<Factory>> component which makes (constructs) the subsequent (or next) <<Moment-Interval>> in the chain.

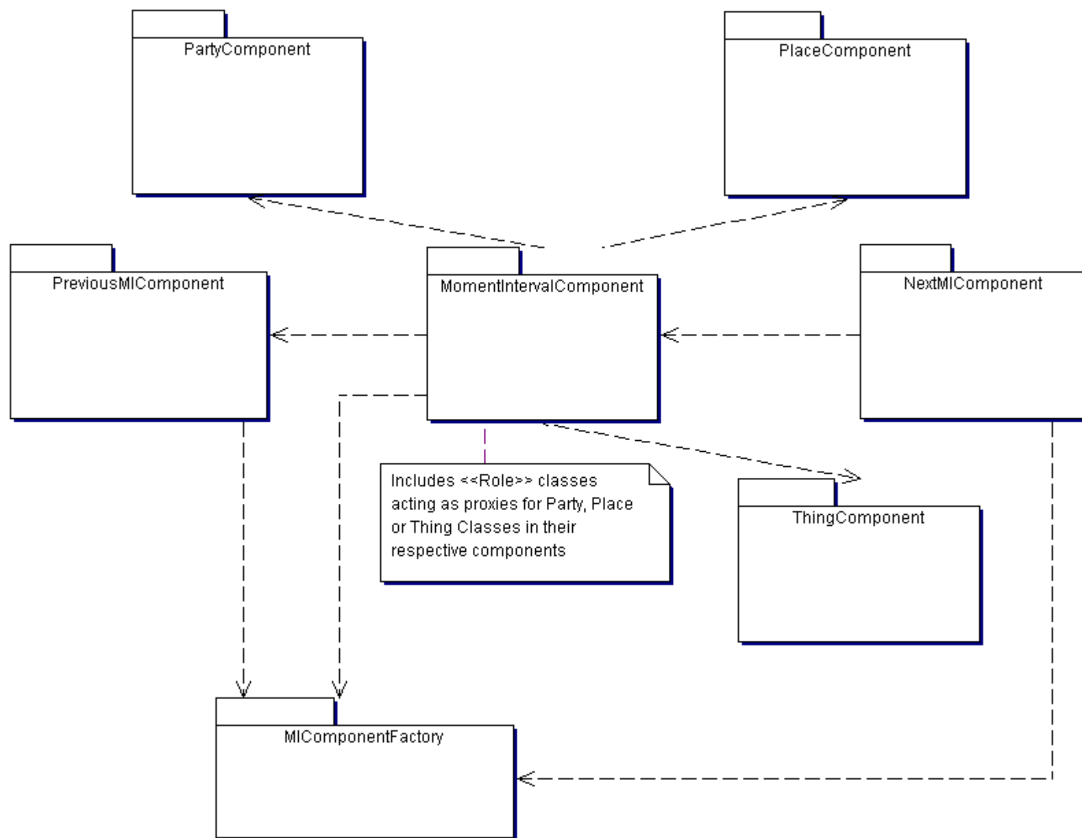

Figure 2. DNC showing Dynamic Dependencies

There probably three other relatively likely schemes for packaging/componentization. One would be to make a component from each layer of a DNC model, i.e. package the whole <<Moment-Interval>> graph with its associated Parties, Places and Things. However, this strategy is risky because it denies easy reuse of the Parties, Places or Things to other components. Only use this strategy when you are very positive about the domain and the limited need for reuse of the green classes within this DNC graph.

Figure 3 shows another more granular strategy which separates out across the whole-part relationship between a <<Moment-Interval>> and its <<MI-Detail>>. You would use this strategy where the <<MI-Detail>> needs to be re-used across generations or versions of its containing <<Moment-Interval>> or the <<Moment-Interval>> and <<MI-Detail>> need to vary independently for some other reason. This might be common where the M-I and its detail are controlled by two different industry specifications which develop independently. I have seen this pattern in telecom applications.

Figure 4 shows another approach where the entire chain of <<Moment-Intervals>> in a workflow is encapsulated as a single component. Note that all the <<Role>> classes go inside this component. This pattern would be appropriate where the

workflow is well understood or where it is likely to vary as a whole and the individual steps do not need to be reused independently.

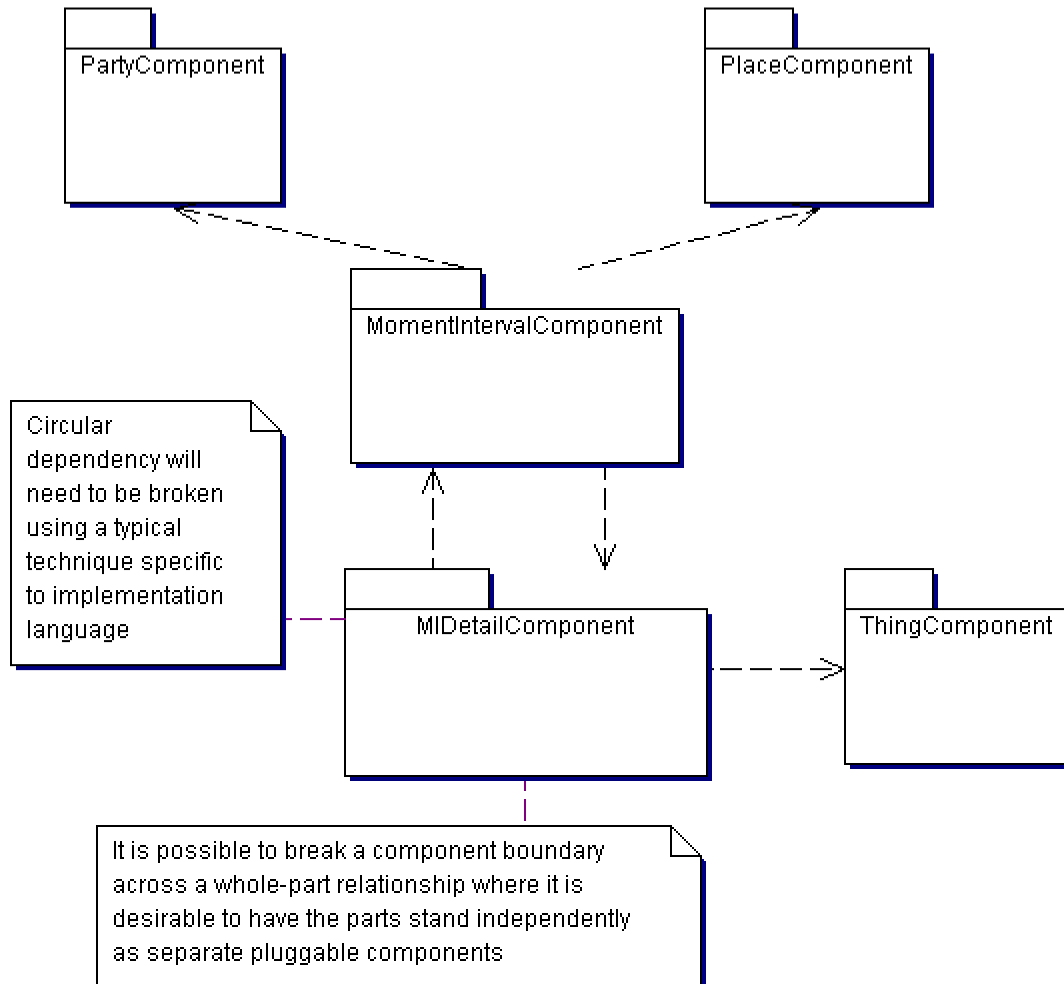

Figure 3 MI-Detail is separated as its own component

Note in all three component diagrams, the dependency direction is from the component with the <<Moment-Interval>> and the <<Role>> out towards the components which hold the green and blue classes. This makes sense because ultimately it is the <<Party>>, <<Place>> or <<Thing>> classes which we want to re-use in other applications and systems. Their respective <<Description>> classes may also be re-usable or extendable through <<plug-in>> points to cope with flexible re-use. Plug-in points will be discussed in another Coad Letter later this year.

Copyright David J. Anderson 2004

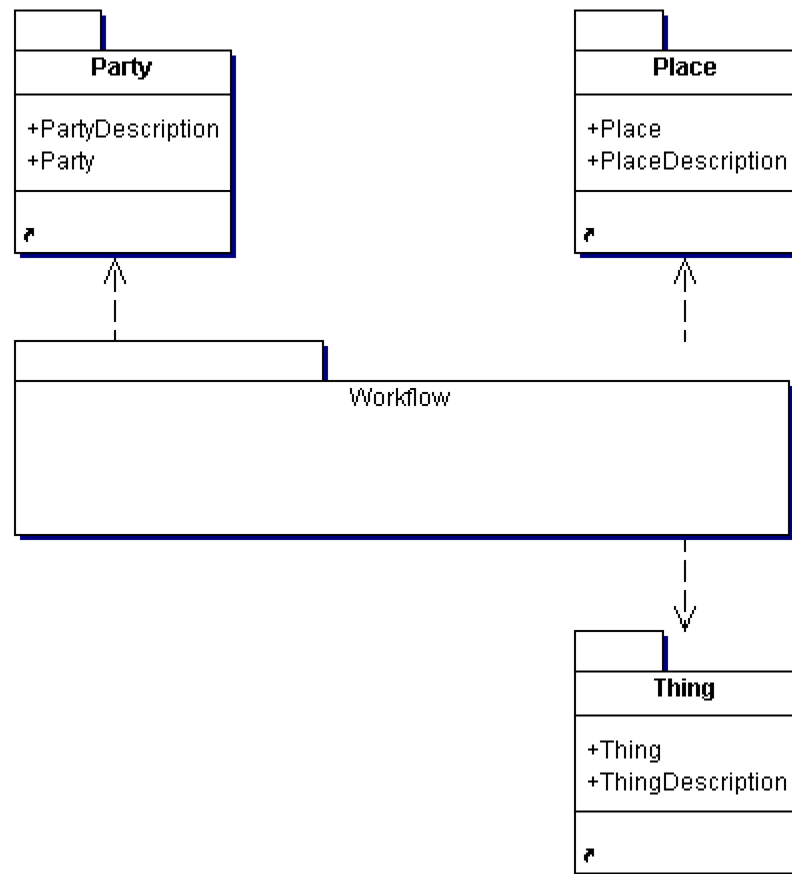

Figure 4. Series of <<Moment-Intervals>> encapsulated as a workflow component

Resolving 2-way Dependencies

Figure 5 shows a generic approach to resolving 2-way dependencies in color model for the simple task of packaging in Java. This example shows how to resolve a dependency across a whole-part relationship by introducing a collector interface. Figure 6 provides a domain specific example from a hotel and event management application. The ICollectSession interface is placed inside a common package and both the **conference** package which contains the Conference class and the **session** package which contains the Session class have a one-way dependency down to the **common** package which contains the ICollectSession interface. This will allow libraries to be created from the classes in a color model and consequently reduce compile times.

Despite the recent trend to continuous integration, I still strongly believe in packaging and library usage. Using libraries encourages developers to think carefully about responsibilities and to clearly identify when a class has been affected by a change. If a library needs recompiled then it clearly needs to be regression tested. If it is unchanged then it need not be tested. Use of libraries encourages cleaner, loosely coupled code which in turn often leads to higher quality and greater re-use.

Copyright David J. Anderson 2004

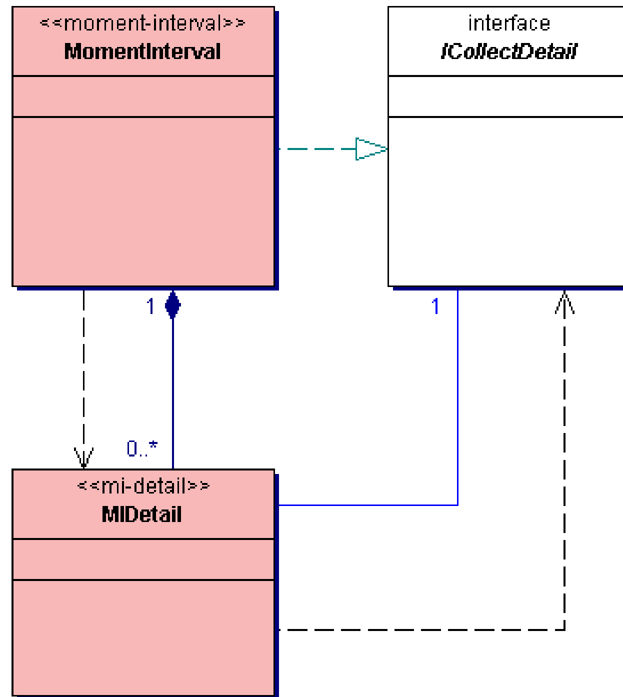

Figure 5. Generic approach to resolving 2-way dependency in Java

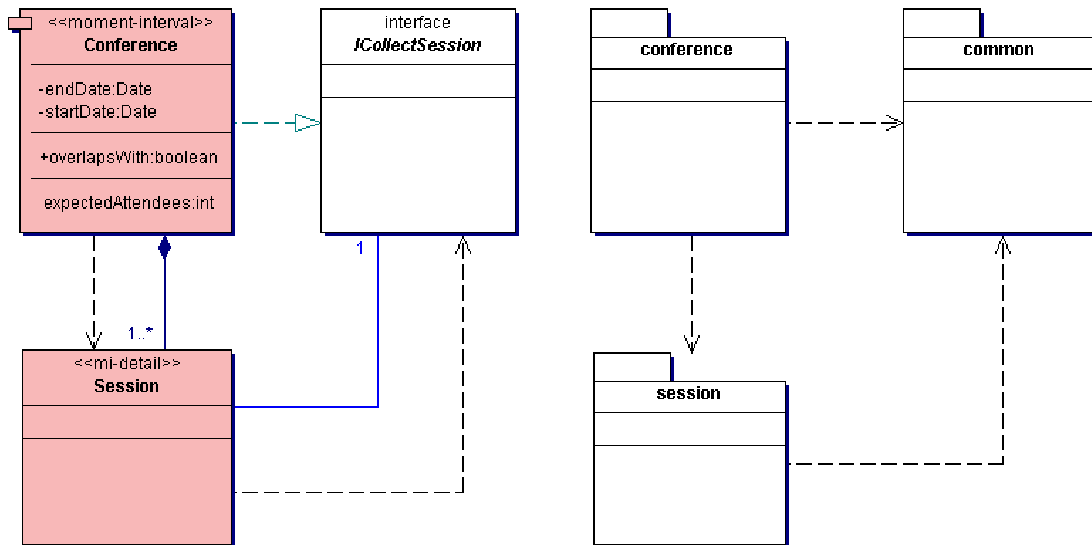

Figure 6. Eliminating a 2-way Dependency with a common package

Figure 7 shows a similar technique employed between a <<Role>> and its <<Place>>. Note how the interface defines the methods which the green class needs to call on its yellow <<Role>> class. Only the methods called across the boundary of the package should be exposed in the interface but all of the methods required must be exposed in the interface. Again, the interface would be packaged in the **common** package as shown in Figure 8.

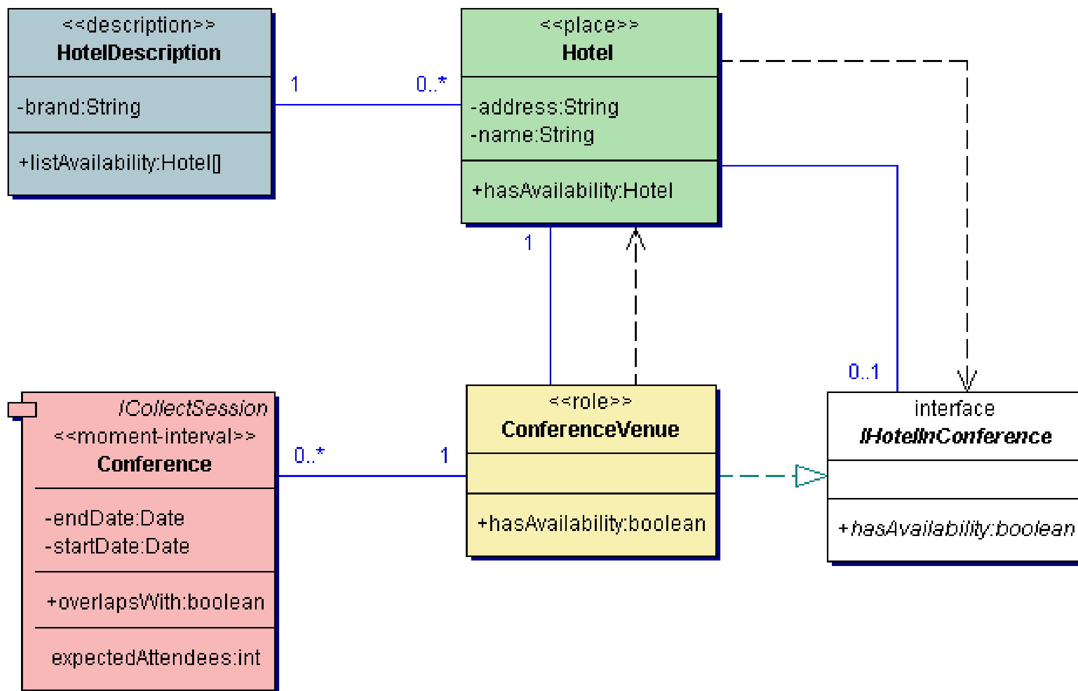

Figure 7. Resolving 2-way dependency between a <<Place>> and its <<Role>>

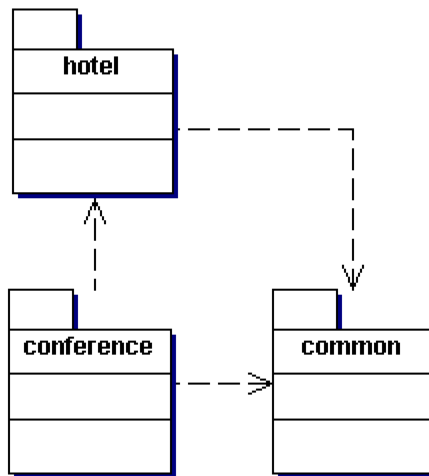

Figure 8. Package dependencies for Figure 7

A longer discussion of how to apply this within a component framework or application server environment will be covered in a future Coad Letter.

Summary

Domain modeling in color modeling with the Domain Neutral Component still has relevance in a world of component frameworks and application servers. Domain modeling allows for accurate and revealing discovery of a problem domain which

enhances understanding. It also provides an ability to achieve bottom-up coarse-grained component definition. As each class in a DNC model is treated as a component in its own rite, aggregation of components is a natural thing which provides for better more accurate assignment of responsibilities and reduces or eliminates refactoring of the coarse-grained component boundary and its interface.

Using the a color model and the DNC pattern as a foundation for component framework architecture leads to faster architecture, with reduced rework, better intra-team communication, and ultimately more re-usable, loosely coupled components. It is well documented that re-use is the no.1 method for achieving high velocity software development. Using a color DNC model as a foundation for your architecture is an important step in being agile and highly productive.

About the author

David J. Anderson is the author of the recent book, "Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results" published in Peter Coad's series by Prentice Hall PTR in September 2003. He is a program manager with Microsoft in Redmond, Washington. David was one of the team which created the popular agile development method, Feature Driven Development. He has introduced FDD at two Fortune 100 companies Sprint (a telecommunications operator in the United States) and Motorola. He holds a degree in Computer Science and Electronics from the University of Strathclyde.

David publishes new thinking regularly through his Agile Management weblog, http://www.agilemanagement.net/

Email: dja@agilemanagement.net

References

Anderson, David J., ***Color Models and the Law of Demeter***, The Coad Letter, Borland Developer Network, 2004
Coad Peter, Eric Le Febvre & Jeff De Luca, ***Java Modeling in Color with UML – Enterprise Components and Process***, Prentice Hall, Upper Saddle River, NJ, 1999
Palmer, Stephen R., ***A New Beginning***, The Coad Letter, Borland Developer Network, 2002
http://www.thecoadletter.com/article/0,1410,29697,00.html
Poppendieck, Mary and Tom Poppendieck, ***Lean Software Development – an Agile Toolkit***, Addison Wesley, New York, NY, 2003
Tate, Bruce, Mike Clark, Bob Lee, Patrick Linskey, ***Bitter EJB***, Manning Publications, 2003