The DCI Architecture: A New Vision of Object-Oriented Programming

by Trygve Reenskaug and James O. Coplien

March 20, 2009

Summary

Object-oriented programming was supposed to unify the perspectives of the programmer and the end user in computer code: a boon both to usability and program comprehension. While objects capture structure well, they fail to capture system action. DCI is a vision to capture the end user cognitive model of roles and interactions between them.

Objects are principally about people and their mental models—not polymorphism, coupling and cohesion

Object oriented programming grew out of Doug Englebart's vision of the computer as an extension of the human mind. Alan Kay's Dynabook vision,[1] often regarded as the progenitor of modern personal lap-tops, was perhaps the epitome of this vision: a truly personal computer that was almost a companion, an extension of self. He would later create a language, Smalltalk, to carry that vision into the very source code. In fact, the goal of object-oriented programming pioneers was to capture end user mental models in the code. Today we are left with the legacy of these visions in the blossoming of interactive graphical interfaces and the domination of object-oriented languages in programming world-wide.

When a user approaches a GUI, he or she does two things: *thinking* and *doing*. For a smooth interaction between man and machine, the computer's "mental" model (also the programmer's mental model) and the end user's mental model must align with each other in kind of mind-meld. In the end, any work that users do on their side of the interface manipulates the objects in the code. If the program provides accurate real-time feedback about how user manipulations affect program state, it reduces user errors and surprises. A good GUI provides this service. Using an interactive program is like being a doctor trying to navigate a probe through a patient's bronchial tubes: just as you can't see the objects in program memory, you can't see the actual probe in the patient's body. You need some external representation of the program structure, or of the bronchial probe, to guide your interaction with a program.

We've been good at the mind-meld of structure

Both object-oriented design and the Model-View-Controller (MVC) framework grew to support this vision. MVC's goal was to provide the illusion of a direct connection from the end user brain to the computer "brain"—its memory and processor.

In some interfaces, this correspondence is obvious: if you create a circle on a PowerPoint® slide, the circle in your mind directly maps onto its representation in computer memory. The rows and columns of a spread sheet ledger map onto the screen rows and columns in a spreadsheet program, which in turn map onto the data structures in the program. Words on a text editor page reflect both our model of a written document and the computer's model of stored text. The object approach to structuring makes such alignment possible, and human thinking quickly aligns with the computer's notion of structure.

MVC is about people and their mental models—not the Observer pattern

Most programmers think of MVC as a fancy composition of several instances of the Observer pattern. Most programming environments provide MVC base classes that can be extended to synchronize the state of the Model, the View, and the Controller. (Model, View and Controller are actually roles that can be played by the objects that the user provides—we'll talk more about roles later.) So it's just a housekeeping technique, right? To think of it that way is to take a nerd's perspective. We'll call that perspective "Model-View-Controller." More deeply, the framework exists to separate the representation of information from user interaction. In that capacity we'll call it "Model-View-Controller-*User*," capturing all four of the important actors at work—MVC-U for short.

It can serve us well to define additional terms more precisely. MVC-U is all about making connections between computer *data* and stuff in the end user's head. *Data* are the representation of information; in computers, we often represent them as bits. But the bits themselves *mean* nothing by themselves: they *mean* something only in the mind of the user when there is an interaction between them. The mind of the end user can interpret these data; then they become *information*. *Information* is the term we use for interpreted data. Information is a key element of the *end user mental model*.[2]

This mapping first takes place as an end user approaches an interactive interface, using it to create the path between the data from which the interface is drawn, and his or her model of the business world. A well-designed program does a good job of capturing the information model in the data model, or at least of providing the illusion of doing so. If the software can do that then the user feels that the computer memory is an extension of his or her memory. If not, then a "translation" process must compensate for the mismatch. It's at best awkward to do this translation in the code (and it shouldn't be necessary if the coder knows the end user cognitive models). It is painful, awkward, confusing, and error-prone for the end user to perform this mapping in their head in real time. To unify these two models is called the *direct manipulation metaphor*: the sense that end users are actually manipulating objects in memory that reflect the images in their head.

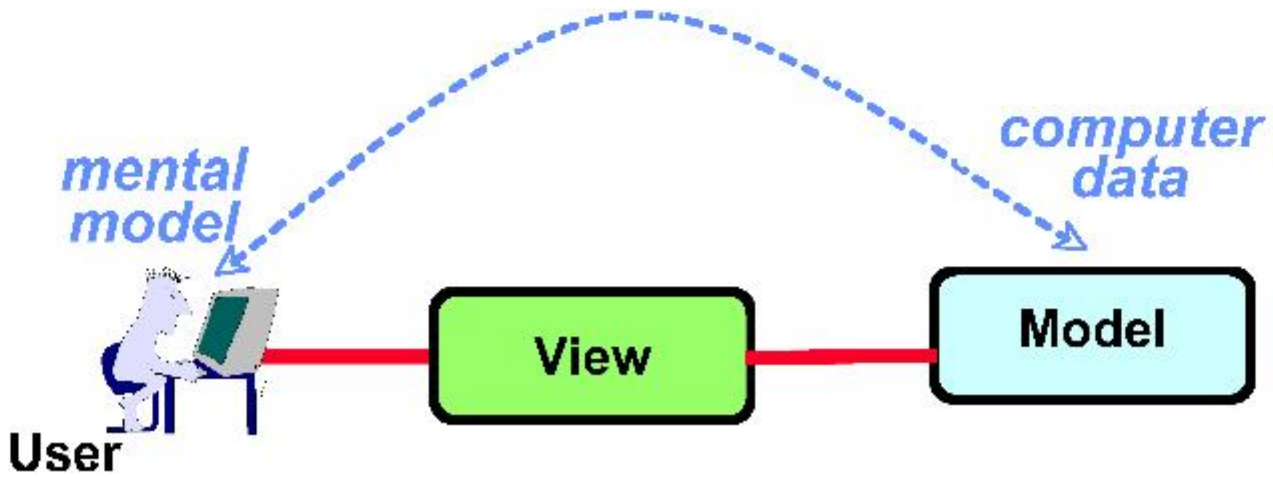

Figure 1. Direct Manipulation

The Direct Manipulation Metaphor

We want the system to provide a short path from the information to the data that represents it in the program (Figure 1). The job of the *model* is to "filter" the raw data so the programmer can think about them in terms of simple cognitive models. For example, a telephone system may have underlying objects that represent the basic building blocks of local telephony called *half-calls*. (Think about it: if you just had "calls," then where would the "call" object live if you were making a call between two call centers in two different cities? The concept of a "half-call" solves this problem.) However, a telephone operator thinks of a "call" as a thing, which has a duration and may grow or shrink in the number of parties connected to it over its lifetime. The Model supports this illusion. Through the computer interface the end user feels as though they are directly manipulating a real thing in the system called a "Call." Other Models may present the same data (of a half-call) in another way, to serve a completely different end user perspective. This illusion of direct manipulation lies at the heart of the object perspective of what computers are and how they serve people.

The View displays the Model on the screen. View provides a simple protocol to pass information to and from the Model. The heart of a View object presents the Model data in one particular way that is of interest to the end user. Different views may support the same data, *i.e.*, the same Models, in completely different ways. The classic example is that one View may show data as a bar graph while another shows the same data as a pie chart.

The Controller creates Views and coordinates Views and Models. It usually takes on the role of interpreting input user gestures, which it receives as keystrokes, locater device data, and other events.

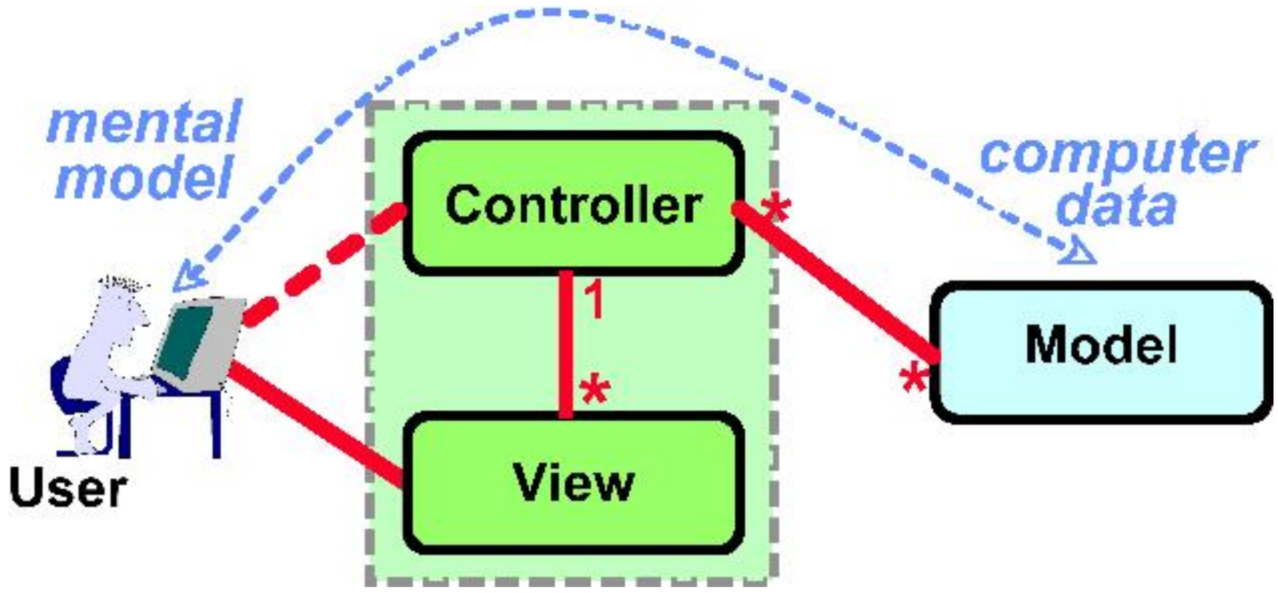

Figure 2. Model-View-Controller-User

Together, these three *roles* define interactions between the objects that play them—all with the goal of sustaining the illusion that the computer memory is an extension of the end user memory: that computer data reflect that end user cognitive model (Figure 2). That summarizes Model-View-Controller-User: it does a good job of supporting the *thinking* part of computer/human interaction.

... but in spite of capturing structure, OO fails to capture behavior

Unfortunately, object orientation hasn't fared so well to capture how we reason about *doing*. There is no obvious "place" for interactions to live, either on the GUI or in the code. There are exceptions to this rule, particularly for simple actions that involve only a single object. For example, a good interface might allow us to use a well-placed paint brush to change the color of a circle on the screen. In the program, the code for re-coloring the circle is itself part of the circle. In these simple cases the end user mental model, the code, and the screen all align. But for a spreadsheet we can't see the sum over a column. Instead, we need to invoke some set of mystical incantations to bring up a sub-window or other field that recovers an earlier constructed formula. With appropriate screen design and interaction design we can limit the damage for the end user, and some user interfaces are surprisingly good at making these actions visible. Still, it is far too often that such interfaces are shrouded in mystery. Consider the totally opaque ceremony that takes place in a popular word processor between a picture and a paragraph as you strive to insert one into the other.

As if things aren't bad enough for the end user, they are as bad or even worse for the programmer. Programmers are people, too, and we want them to be able to map from their understanding of user needs to their understanding of the code. Object-oriented programming languages traditionally afford no way to capture collaborations between objects. They don't capture algorithms that flow over those collaborations. Like the domain structure captured by object instances, these collaborations and interactions also have structure. They form part of the end user mental model, but you can't find a cohesive representation of them in the code. For example, users have expectations for their interactions with a spell-checker in a word processor and have a preconceived notion of its interactions with the text, with some dictionary, and with the end user. Which object should encapsulate the spell-checking operation in a word processor: The editing buffer? The dictionary? Some global spell-checker object? Some of these options lead to poor cohesion of the object that hosts spell checking while other options increase the coupling between objects.

In this article, we'll show how to combine roles, algorithms, objects, and associations between them to provide a stronger mapping between the code and the end-user mental model. The result is an architecture based on the object *D*ata, the *C*ollaborations between objects, and the way that Use Cases scenarios comprise *I*nteractions between roles: the DCI architecture.

Where did we go wrong?

We can trace much of our failure to capture the end user mental model of *doing* to a kind of object mythology that flourished doing the 1980s and into the first half of the 1990s. Some buzzwords of this mindset included *anthropomorphic design*, *smart objects*, and *emergent system behavior*. We were taught that system behavior should "emerge" from the interaction of dozens, hundreds or thousands of local methods. The word of the day was: think locally, and global behavior would take care of itself. Anyone caught writing a method that looked like a procedure, or caught doing procedural decomposition, was shunned by the OO community as "not getting it."

In fact, most GUI problems start with the programmer's inability to capture the end user cognitive model in the code. The MVC framework makes it possible for the user to reason about what the system *is*: the *thinking* part of the user cognitive model. But there is little in object orientation, and really nothing in MVC, that helps the developer capture *doing* in the code. The developer doesn't have a place where he or she can look to reason about end user behavioral requirements.

Back in the 1960s, we could take the behavioral requirements for a program, and the FORTRAN code that implemented them, and give both of them to an office mate—together with a big red pen—to review whether the code matched the requirements. The overall form of the code reflected the form of the requirements. In 1967, software engineering took away my ability to do this: the algorithm had to be distributed across the objects, because to have a large method that represented an entire algorithm was believed to not be a "pure" object-oriented design. How did we decide to split up the algorithm and distribute its parts to objects? On the basis of coupling and cohesion. Algorithms (methods) had to be collocated with the object that showed the most affinity for the algorithm: optimizing cohesion.

That works fine when an algorithm lives within a single object, as might be true for changing the color of a circle on the screen, or adding a typed character to a word processor's text buffer. However, interesting business functionality often cuts across objects. The spell-checker in the text editor involves the screen, some menus, the text buffer, and a dictionary. Even for a shapes editor, the problem of calculating overlapping regions belongs to multiple objects. Object-orientation pushed us into a world where we had to split up the algorithm and distribute it across several objects, doing the best piecemeal job that we could.

Back into the Users' Head

If the goal of object-orientation was to capture end users' conceptual model of their worlds, it might serve us well to journey back into that space to find out what lurks there. We'll start with familiar territory: the data model, which most nerds today call *objects* (but then, to our puzzlement, model and discuss only as *classes*) and then move on to more dynamic concepts called *roles* and *collaborations*. All three of these—the data model, the role model, and the collaboration model—are conceptual concerns independent of programming language. But, of course, one of our goals is that the programming language *should* be able to express these things. So we'll also look at programming concepts that express these concepts in code. One of these concepts is called a *class* (and we're again on familiar ground), and the second is called a *role*.

Data: representing the user's mental model of *things* in their world

Managing data is arguably the second oldest profession in computer science (we'll talk about the oldest profession below). The old Data Flow Diagram (DFD) people used to tell us that the data are the stable part of design. This truism carried forward into objects, and object designers were encouraged to look for stable object structures.

A particularly simplistic rule of thumb in early object-oriented design was: nouns (e.g. in the requirements document) are objects, and verbs are methods. This dichotomy naturally fit the two concepts that programming languages could express. Object-oriented programming languages—particularly the "pure" ones—expressed *everything* in terms of objects or methods on objects. (Of course, most programming languages used classes to do this. The point is that nothing was supposed to exist outside of an object framework.) So if I looked at a Savings Account object, the fact that it was an object led us to capture it as such (or as a class). The fact that it could both decrease its balance and could do a withdrawal were lumped together as methods. Both are behaviors. However, these two behaviors are radically different. Decreasing the balance is merely a characteristic of the data: what it *is*. To do a withdrawal reflects the *purpose* of the data: what it *does*. Being able to handle a withdrawal—which infers transaction semantics, user interactions, recovery, handling error conditions and business rules—far outstrips any notion of a data model.

Withdrawal, in fact, is a behavior of the system and entails system state, whereas reducing the balance is what makes an account an account and relates only to the object state. These two properties are extremely different in kind from the important perspectives of system architecture, software engineering, and maintenance rate of change. Object-orientation lumped them into the same bucket.

The problem with this approach is this: If objects are supposed to remain stable, and if all of the code is in objects, then where do I represent the parts that change? A key, longstanding hallmark of a good program is that it separates what is stable from what changes in the interest of good maintenance. If objects reflect the stable part of the code, there must be a mechanism other than objects to express requirements changes in the code, supporting the Agile vision of evolution and maintainability. But objects are stable—and in an object-oriented program, there is no "other mechanism."

Stuck with these artificial constraints, the object world came up with an artificial solution: using inheritance to express "programming by difference" or "programming by extension." Inheritance is perhaps best understood as a way to classify objects in a domain model. For example, an exclusive-access file may be a special kind of disk file, or magnetic, optical and mechanical sensors might be different implementations of the more general notion of sensor. (You might object and say that this is subtyping rather than inheritance, but few programming languages distinguish the expression of these two intents). Because inheritance could express variations on a base, it quickly became a mechanism to capture behavioral additions to a "stable" base class. In fact, this approach became heralded as an honorable design technique called the *open-closed principle*: that a class was closed to modification (i.e., it had to remain stable to capture the stability of the domain model) but open to extension (the addition of new, unanticipated code that supported new user behaviors). This use of inheritance crept out of the world of programming language into the vernacular of design.

Somewhere along the line, statically typed languages got the upper hand, supported by software engineering. One important aspect of static type system analysis was the class: a construct that allowed the compiler to generate efficient code for method lookup and polymorphism. Even Smalltalk, whose initial vision of objects and a dynamic run-time environment was truly visionary, fell victim to the class compromise. The class became the implementation tool for the analysis concept called an object. This switch from dynamics to statics was the beginning of the end for capturing dynamic behavior.

Inheritance also became an increasingly common way to express subtyping, especially in Smalltalk and C++. You could cheat in Smalltalk and invoke a method in any class in an object's inheritance hierarchy whether or not a default implementation appeared in the base class. It would work, but it exacerbated the discovery problem, because the base class interface wasn't representative of the object's total behavior. The statically typed languages created a culture of inheritance graphs as design abstractions in their own right, fully represented by the base class interface. But because programming by extension took place at the bottom of the hierarchy, newly added methods either didn't appear in the base class—or, worse, needed to be added there (e.g., as pure virtual functions in C++) every time the inheritance hierarchy was extended to incorporate a new method.

The alternative was to take advantage of static typing, and to let clients of a derived class have access to the class declaration of classes that were added for programming-by-extension. That preserved the "integrity" of the base class. However, it also meant that statically typed languages encouraged cross-links between the buried layers of class hierarchies: an insidious form of violating encapsulation. One result was global header file proliferation. The C++ world tried to respond with RTTI and a variety of other techniques to help manage this problem while the community of dynamically typed languages shrugged and noted that this wasn't a problem for them.

The rhetoric of the object community started turning against inheritance in the mid-1980s, but only out of a gut feel that was fueled by a few horror stories (inheritance hierarchies 25 deep) and the resulting software engineering nightmare of trying to trace business behavior back into the code.

In the end, this whole sordid story suggests that extension by derivation was a less-than-ideal solution. But, in fact, inheritance wasn't the most infested fly in the ointment. Most such code changes could be traced back to behavioral requirements changes, and most such changes were driven by end users' desire for new behaviors in the code. Software is, after all, a service and not really a product, and its power lies in its ability to capture tasks and the growth and changes in tasks. This is particularly credible in light of the argument (well-sustained over the years) that the data model is relatively stable over time. The discord between the algorithm structure and domain structure would be the ultimate undoing of classes as units of growth; we'll get back to that below.

There is another key learning that we'll carry forward from this perspective: that domain classes should be dumb. Basic domain objects represent our primordial notions of the essence of a domain entity, rather than the whole universe of processes and algorithms that burdens traditional object-oriented development as Use Cases pile up over time. If I asked you what a Savings Account object can do, you'd be wise to say that it can increase and decrease its balance, and report its balance. But if you said that it can handle a deposit, we're suddenly in the world of transactions and interactions with an ATM screen or an audit trail. Now, we've jumped outside the object and we're talking about coupling with a host of business logic that a simple Savings Account has no business knowing about. Even if we decided that we wanted to give objects business intelligence from the beginning, confident that we could somehow get it right so the interface wouldn't have to change much over time, such hopes are dashed by the fact that initial Use Cases give you a very small slice of the life-time code of a system. We must separate simple, stable data models from dynamic behavioral models.

Roles: a (not so) new concept of *action* that also lives in users' heads

So let's go back into the user's head and revisit the usual stereotype that everything up there is an object. Let's say that we're going to build an ATM and that one Use Case we want to support is Money Transfer. If we were to ask about your fond remembrances of your last account funds transfer, what would you report? A typical response to such a question takes the form, "Well, I chose a source account and a transfer amount, and then I chose a destination account, and I asked the system to transfer that amount between the accounts." In general, people are often a little less precise than this, using words like "one account" and "another account" instead of "source account" and "destination account."

Notice that few people will say "I first picked my savings account, and then an amount, and then picked my investment account..." and so forth. Some respondents may actually say that, but to go to that level artificially constrains the problem. If we look at such scenarios for any pair of classes, they will be the same, modulo the class of the two accounts. The fact is that we all carry, in our heads, a general model of what fund transfer means, independent of the types of the account involved. It is that model—that *interaction*—that we want to mirror from the user's mind into the code.

So the first new concept we introduce is *roles*. Whereas objects capture what objects *are*, roles capture collections of behaviors that are about what objects *do*. Actually, it isn't so much that the concept of roles is new as it is unfamiliar. Role-based modeling goes back at least to the OORAM method, which was published as a book in 1996.[3] Roles are so unfamiliar to us because so much of our object thinking (at least as nerds) comes from our programming languages, and languages have been impoverished in their ability to express roles.

The interactions that weave their way through the roles are also not new to programming: we call them algorithms, and they are probably the only design formalism that predates data as having their own vocabulary and rules of thumb. What's interesting is that we consciously weave the algorithms through the roles. It is as if we had broken down the algorithm using good old procedural decomposition and broken the lines of decomposition along role boundaries. We do the same thing in old-fashioned object modeling, except that we break the lines of procedural decomposition (methods) along the lines of object boundaries.

Unfortunately, object boundaries already mean something else: they are loci of encapsulated domain knowledge, of the data. There is little that suggests that the stepwise refinement of an algorithm into cognitive chunks should match the demarcations set by the data model. Old-fashioned object orientation forced us to use the same mechanism for both demarcations, and this mechanism was called a *class*. One or the other of the demarcating mechanisms is likely to win out. If the algorithmic decomposition wins out, we end up with algorithmic fragments landing in one object but needing to talk to another, and coupling metrics suffer. If the data decomposition wins out, we end up slicing out just those parts of the algorithm that are pertinent to the topic of the object to which they are assigned, and we end up with very small incohesive methods. Old-fashioned object orientation explicitly encouraged the creation of such fine-grain methods, for example, a typical Smalltalk method is three statements long.

Roles provide natural boundaries to carry collections of operations that the user logically associates with each other. If we talk about the Money Transfer example and its roles of Source Account and Destination Account, the algorithm might look like this:

1. Account holder chooses to transfer money from one account to another
2. System displays valid accounts
3. User selects Source Account
4. System displays remaining valid accounts
5. Account holder selects Destination Account
6. System requests amount
7. Account holder inputs amount
8. <u>Move Transferred Money and Do Accounting</u>

The Use Case Move Transferred Money and Do Accounting might look like this:

1. System verifies funds are available
2. System updates the accounts
3. System updates statement information

The designer's job is to transform this Use Case into an algorithm that honors design issues such as transactions. The algorithm might look like this:

1. Source account begins transaction
2. Source account verifies funds available (notice that this must be done inside the transaction to avoid an intervening withdrawal!)
3. Source account reduces its own balance
4. Source account requests that Destination Account increase its balance
5. Source Account updates its log to note that this was a transfer (and not, for example, simply a withdrawal)

6. Source account requests that Destination Account update its log
7. Source account ends transaction
8. Source account informs Account Holder that the transfer has succeeded

The code for this algorithm might look like this:

```
template <class ConcreteAccountType>
class TransferMoneySourceAccount: public MoneySource
{
private:
 ConcreteDerived *const self() {
    return static_cast<ConcreteDerived*>(this);
 }
 void transferTo(Currency amount) {
    // This code is reviewable and
    // meaningfully testable with stubs!
    beginTransaction();
    if (self()->availableBalance() < amount) {
      endTransaction();
      throw InsufficientFunds();
    } else {
      self()->decreaseBalance(amount);
      recipient()->increaseBalance (amount);
      self()->updateLog("Transfer Out", DateTime(),
                amount);
      recipient()->updateLog("Transfer In",
             DateTime(), amount);
    }
    gui->displayScreen(SUCCESS_DEPOSIT_SCREEN);
    endTransaction();
 }
```

It is almost a literal expansion from the Use Case. That makes it more understandable than if the logic is spread over many class boundaries that are arbitrary with respect to the natural organization of the logic—as found in the end user mental model. We call this a *methodful role*—a concept we explore more thoroughly in the next section.

Whole objects, each with two kinds of know-how

At their heart, roles embody generic, abstract algorithms. They have no flesh and blood and can't really *do* anything. At some point it all comes down to objects—the same objects that embody the domain model.

The *fundamental problem* solved by DCI is that people have two different models in their heads of a single, unified thing called an object. They have the what-the-system-*is* data model that supports *thinking* about a bank with its accounts, and the what-the-system-*does* algorithm model for transferring funds between accounts. Users recognize individual objects and their domain existence, but each object must also implement behaviors that come from the user's model of the interactions that tie it together with other objects through the roles it plays in a given Use Case. End users have a good intuition about how these two views fit together. For example, end users know that their Savings Accounts take on certain responsibilities in the role of a Source Account in a Money

Transfer Use Case. *That*, too—the mapping between the role view and data view—is also part of the user cognitive model. We call it the *Context* of the execution of a Use Case scenario.

We depict the model in Figure 3. On the right we capture the end user role abstractions as interfaces (as in Java or in C#; in C++, we can use pure abstract base classes). These capture the basic architectural form, to be filled in as requirements and domain understanding grow. At the top we find roles that start as clones of the role abstractions on the right, but whose methods are filled in. For a concept like a Source Account in a Money Transfer Use Case, we can define some methods independent of the exact type of object that will play that role at run time. These roles are *generic types*, analogous to Java or Ada generics or C++ templates. These two artifacts together capture the end user model of roles and algorithms in the code.

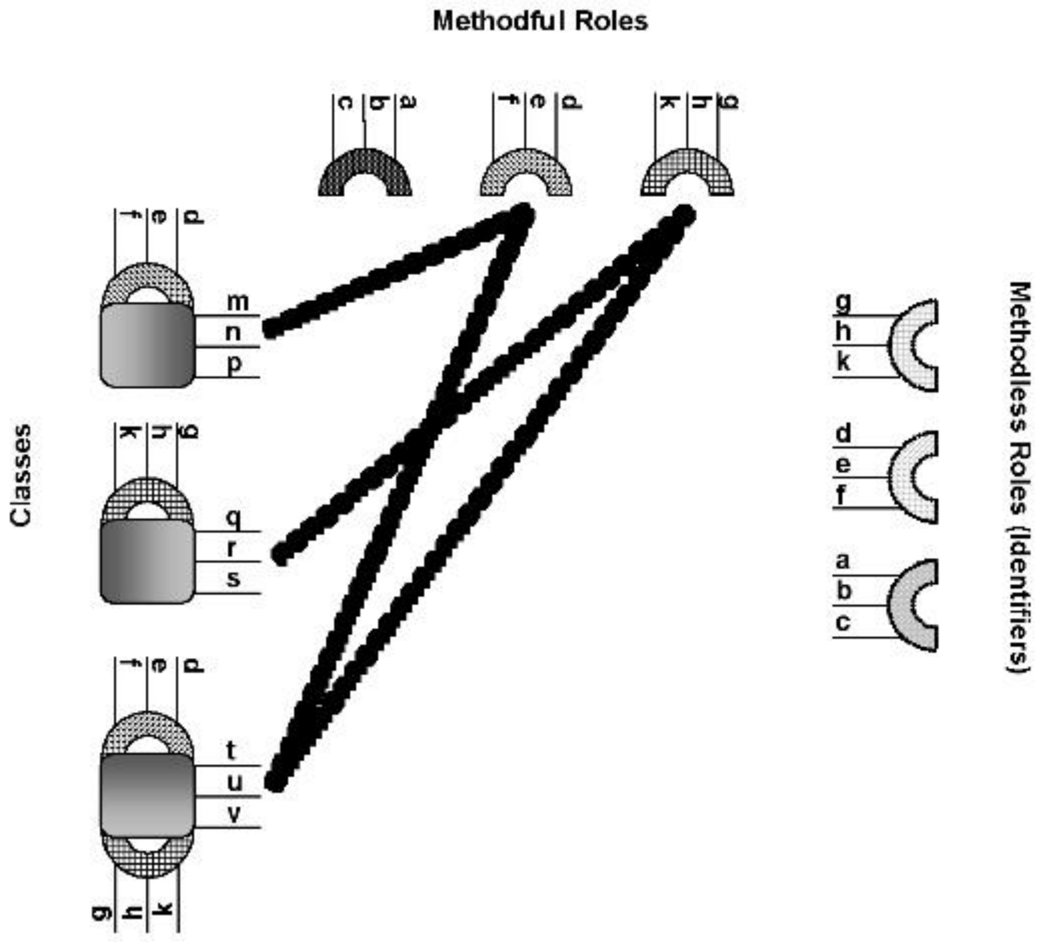

Figure 3. Combining Structure and Algorithm in a Class

On the left we have our old friends, the classes. Both the roles *and* classes live in the end user's head. The two are fused at run time into a single object. Since objects come from classes in most programming languages, we have to make it appear as though the domain classes can support the business functions that exist in the separate source of the role formalisms. At compile time programmers must face the end user's models both of Use Case scenarios and the entities they operate on. We want to help the programmer capture those models *separately* in two different programming constructs, honoring the dichotomy in the end user's head. We usually think of classes as the natural place to collect such behaviors or algorithms together. But we must also support the seeming paradox that each of these *compile*-time concepts co-exists with the other at *run* time in a single thing called the object.

This sounds hard, but even end users are able to combine parts of these two views in their heads. That's why they know that a Savings Account—which is just a way of talking about how much money I can access right now through a certain key called an account number—can be asked to play the role of a Source Account in a Money Transfer operation. So we should be able to snip operations from the Money Transfer Use Case scenario and add them to the rather dumb Savings Account object. Figure 3 shows such gluing together of the role logic (the arcs) and the class logic (rounded rectangles). Savings Account already has operations that allow it to carry out its humble job of reporting, increasing, or decreasing its balance. These latter operations, it supports (at run time) from its domain class (a compile-time construct). The more dynamic operations related to the Use Case scenario come from the roles that the object plays. The collections of operations snipped from the Use Case scenario are called *roles*. We want to capture them in closed form (source code) at compile time, but ensure that the object can support them when the corresponding Use Case comes around at run time. So, as we show in Figure 4, an object of a class supports not only the member functions of its class, but also can execute the member functions *of the role it is playing at any given time* as though they were its own. That is, we want to *inject* the roles' logic into the objects so that they are as much part of the object as the methods that the object receives from its class at instantiation time.

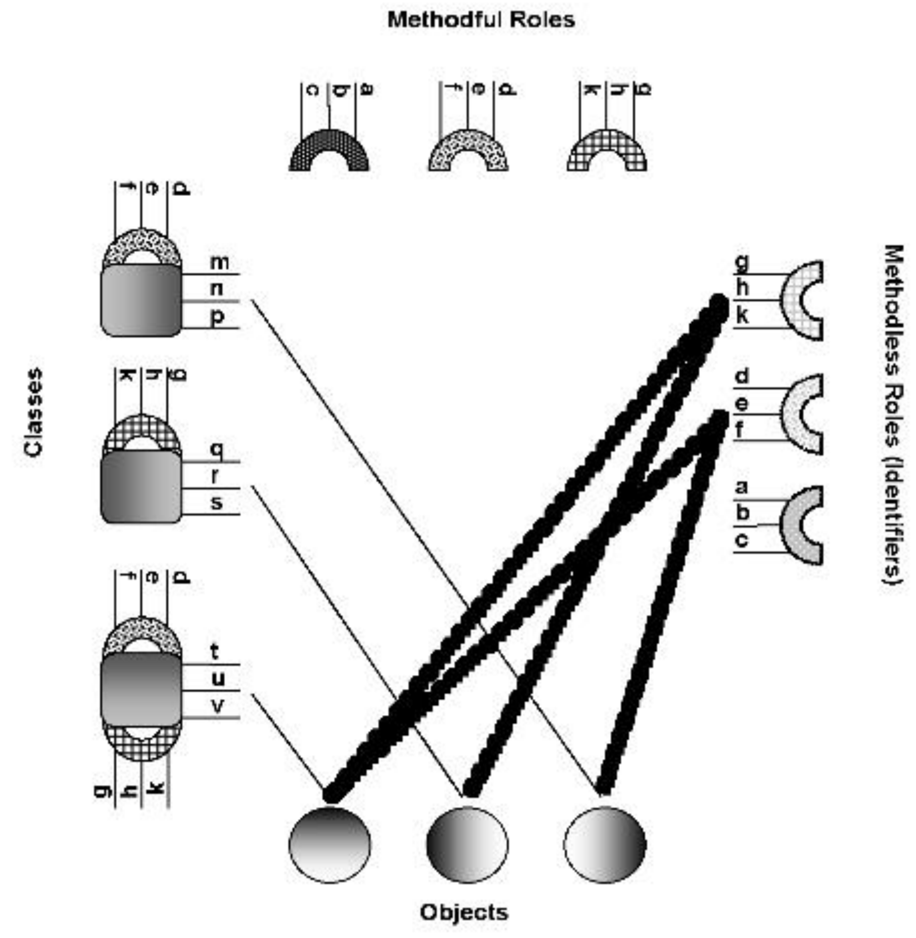

Figure 4. Combining Structure and Algorithm in an Object

Here, we set things up so each object has all possible logic at compile time to support whatever role it might be asked to play. However, if we are smart enough to inject just enough logic into each object at run time, just as it is needed to support its appearance in a given role, we can do that, too.

Roles working together: Contexts and Interactions

When I go up to an ATM to do a money transfer, I have two objects in mind (let's say that they are My Savings Account and My Investment Account), as well as a vision of the process, or algorithm, that takes money from some Source Account and adds it to some Destination Account in a way that is agreeable to both me and the bank. (It's probably true that My Savings Account isn't actually an object in a real bank, but it probably *is* an object within the realm of the ATM. Even if it isn't, there are some nice generalizations in DCI that cause it not to matter.) I also have a notion of how to map between these two. I establish that mapping, or context, as I interact with the ATM.

First, I probably establish that I want to do a funds transfer. That puts a money-transfer scenario in my mind's "cache," as well as bringing some kind of representation of the roles and algorithms into the computer memory. We can capture these scenarios in terms of roles, as described above.

Second, I probably choose the Source Account and Destination account for the transfer. In the computer, the program brings those objects into memory. They are dumb, dumb data objects that know their balance and a few simple things like how to increase or decrease their balance. Neither account object alone understands anything as complex as a database transaction: that is a higher-order business function related to what-the-system-does, and the objects individually are about what-the-system-is. The higher-level knowledge doesn't live in the objects themselves but in the roles that those objects play in this interaction.

Now I want to do the transfer. For the transfer to happen, I need My Savings Account to be able to play the role of Source Account, and the My Investment Account object to play the role of the Destination Account. Imagine that we could magically glue the member functions of the roles onto their respective objects, and then just run the interaction. Each role "method" would execute in the context of the object into which it had been glued, which is exactly how the end user perceives it. In the next section of this article we'll look exactly at how we give the objects the intelligence necessary to play the roles they must play: for the time being, imagine that we might use something like delegation or mix-ins or Aspects. (In fact each of these approaches has at least minor problems and we'll use something else instead, but the solution is nonetheless reminiscent of all of these existing techniques.)

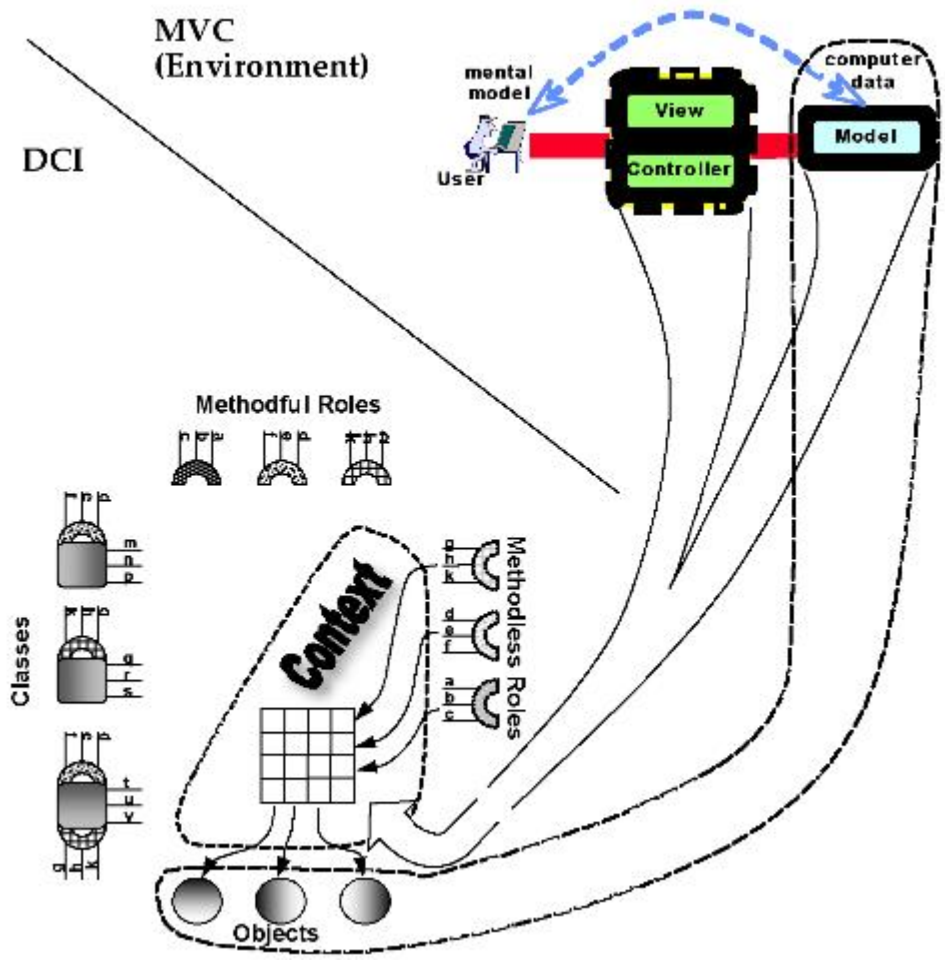

Figure 5. Mapping Roles to Objects

The arrow from the Controller and Model into the Context just shows that the Controller initiates the mapping, perhaps with some parameters that give hints about the mapping, and that the Model objects are the source of most mapping targets. The Methodless Roles are identifiers through which application code (in the Controller and in Methodful Roles) accesses objects that provide services available through identifiers of that type. This becomes particularly useful in languages with compile-time type checking, as the compiler can provide a modicum of safety that ensures, at compile time, that a given object can and will support the requested role functionality.

By this time, all the objects necessary to affect the transfer are in memory. As indicated above, the end user also has a process or algorithm in mind to do the money transfer in terms of the roles involved. We need to pick out code that can run that algorithm, and then all we have to do is line up the right objects with the right roles and let the code run. As shown in Figure 5, the algorithm and role-to-object mapping are owned by a *Context* object. The Context "knows" how to find or retrieve the objects that become the actual actors in this Use Case, and "casts" them to the appropriate roles in the Use Case scenarios (we use the term "cast" at least in the theatrical sense and conjecturally in the sense of some programming language type systems). In a typical implementation there is a Context object for each Use Case, and each Context includes an identifier for each of the roles involved in that Use Case. All that the Context has to do is bind the role identifiers to the right objects. Then we just kick off the trigger method on the "entry" role for that Context, and the code just runs. It might run for nanoseconds or years—but it reflects the end user model of computation.

Now we have the complete DCI architecture:

- The *data*, that live in the domain objects that are rooted in domain classes;
- The *context* that brings live objects into their positions in a scenario, on demand;
- The *interactions*, that describe end-user algorithms in terms of the roles, both of which can be found in end users' heads.

As shown in Figure 5, we can think of the Context as a table that maps a role member function (a row of the table) onto an object method (the table columns are objects). The table is filled in based on programmer-supplied business intelligence in the Context object that knows, for a given Use Case, what objects should play what roles. A method of one role interacts with other role methods in terms of their role interfaces, and is also subject to the role-to-object mapping provided by the Context. The code in the Controller can now deal with business logic largely in terms of Contexts: any detailed object knowledge can be written in terms of roles that are translated to objects through the Context.

One way of thinking about this style of programming is that it is a higher order form of polymorphism than supported by programming languages. In fact, all of the polymorphism can be under programmer control: roles are *explicitly* mapped to objects, and every role method invocation can be statically bound. This makes the code straightforward to analyze and understand statically. Compare that with the usual implementation of method dispatch in an object-oriented programming language, where it is in general impossible to determine where a method invocation will end up through a static analysis of the code.

In some implementations the Context also does the injection of the business logic methods into the domain objects. This is particularly true in implementations based on dynamic languages such as Python and Ruby. In C++ and C# we usually "pre-load" all of the business logic methods by injecting them at the class level, which can be done even at compile time. In Scala we can achieve a hybrid when creating an object from a domain class by injecting the role methods as part of the instantiation. (Scala is really doing the same thing as C++ and C#, but it has a nice syntax of specifying mixins at instantiation points. The Scala compiler will generate an anonymous class that pre-loads all of the business logic methods, and that class is intantiated just at that one point.) When the object comes into existence it has a hybrid type that offers the behaviors both of the base domain class as well as the Use Case roles.

Nested Contexts

One can imagine building rich Context objects that define whole subgraphs of self-contained role relationships: relationships so stable that they constitute a kind of domain in their own right. If these Context objects have a small number of public methods they can behave like domain objects. Consider a Savings Account, which is often wrongly used as an example of a class in simple courses on object orientation. A Savings Account is really a collection of behaviors on roles, where the roles are transactions, transaction logs, and audit trails. If Savings Account is a Context, it can map these roles onto the right objects for a given method (e.g., to calculate the balance of the account or to generate a monthly statement) and then kick off the computation on the suitable role. The Savings Account Context can be used as a domain object by "higher-level" Context objects, and it can call on Context objects below it. This is a powerful concept supporting a multi-tiered domain model.

Traits as the design trick to combine characteristics and purpose

The question is: how do we do this? And the punch line is a concept called a *trait*. If a *role* is an analysis concept (from the mind of the end user), then a *trait* is a general design concept that represents the role, and its implementation varies from programming language to programming language. For example, we can represent traits in C++ as templates whose member functions are composed with those of a concrete class at compile time, so that the object exhibits both the class and template behaviors at run time.

```
. . . .
template <class ConcreteAccountType> class TransferMoneySourceAccount {
public:
   void transferTo(Currency amount) {
        beginTransaction();
        if (self()->availableBalance() < amount) {
        . . . .
}

. . . .

class SavingsAccount:
    public Account,
    public TransferMoneySourceAccount<SavingsAccount> {
public:
    void decreaseBalance(Currency amount) {
        . . . .
    }
}

. . . .
```

In Scala, traits are implemented by a language construct called, curiously enough, a *trait*, whose methods can be injected into an object at instantiation time.

```
. . . .

trait TransferMoneySourceAccount extends SourceAccount {
  this: Account =>

  // This code is reviewable and testable!
  def transferTo(amount: Currency) {
    beginTransaction()
    if (availableBalance < amount) {
        . . . .
    }
}

. . . .
val source = new SavingsAccount with TransferMoneySourceAccount
val destination = new CheckingAccount with TransferMoneyDestinationAccount
. . . .
```

In Squeak Smalltalk, we implement methodful roles with Squeak Traits used according to certain conventions pioneered by Schärli[4] and inject a trait's methods into appropriate Data classes by adding its methods to the class method tables at compile time.

```
. . . .
```

```
RoleTrait named: #TransferMoneySource
   uses: {}
   roleContextClassName: #MoneyTransferContext
   category: 'BB5Bank-Traits'
   . . . .

TransferMoneySource>>transfer: amount
   self balance < amount
      ifTrue: [self notify: 'Insufficient funds'. ^self].
   . . . .

Object subclass: #Account
   uses: TransferMoneySource
   instanceVariableNames: 'balance'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BB5Bank-Data'
```

DCI implementations also exist in C#/.Net (Christian Horsdal Gammelgaard), Ruby (Steen Lenmann), Python (David Byers and Serge Beaumont), and Groovy (Lars Vonk). The Qi4J environment (Richard Öberg and Steen Lehmann) is pushing forward the ability to express traits in a Java environment.

Properties of DCI

- *We use roles to capture the main user concepts that participate in a Use Case requirement.* Roles are first-class components of the end user cognitive model, so we want to reflect them in the code. Semantically, these roles map closely to the concept of *interfaces* in Java or .Net. However, we use interfaces to capture only the overall form of the behavioral design. Ultimately, our goal is to capture the Use Cases in code, and we'll use other language features to do that. The approach varies with programming language. In Squeak and Scala, we can use *traits* directly. In C++ we can simulate traits using templates. In other languages, we can use classes together with some tricks that associate methods of one class with an object of another.
- *We use objects to capture the deep domain concepts that come from experience and tacit knowledge, as* barely *smart data*. In the old days we distributed responsibilities to classes using CRC cards (*C*lasses, *R*esponsibilities, and *C*ollaborations). But it isn't classes that exhibit responsibilities: roles do. We find this when we ask people to elicit their recollection of some activity: people talk about the task of ordering a book, of transferring money between accounts, and most such transactions as involving roles rather than classes.
- *The software exhibits the open-closed principle.* Whereas the open-closed principle based on inheritance alone led to poor information hiding, the DCI style maintains the integrity of both the domain classes and the roles. Classes are closed to modification but are opened to extension through injection of roles.
- *DCI is a natural fit for Agile software development*. It allows programmers to connect directly with the end user mental model (going beyond just customers to engage end user interactions instead of processes and tools). We can therefore use shared customer vocabulary and iterate the code side by side with them (customer collaboration over contract negotiation). We can reason about the form of task sequencing (which greatly raises the chance of delivering working software—because at least the programmer can understand it, and the translation distance to the end user mental model is much shorter). And, last but not least, it separates the rapidly changing Use Case part from the stable domain part so that we embrace change. Each of these benefits ties directly to a provision of the Agile Manifesto (http://www.agilemanifesto.org).

Other bits

There are certainly other models in the user's head. One common darling of some software engineering camps is *business rules*. DCI doesn't provide a convenient home to capture rules; that is perhaps a weakness in the same way that the failure to capture interactions was a weakness of primordial object orientation. Many other formalisms, such as states and state transitions, can be viewed as derived models that come from the data and usage models. For example, I know that it makes sense to depress the accelerator on my car only if I am in a state where the gearbox is engaged; the state machine representation of this constellation would show an allowable accelerator "message" to occur only in the gearbox "state." However, this transition can also be viewed as a sequence of steps that are described in terms of roles (accelerator, gearbox, engine). A quick check with our intuition suggests that this latter model is a better fit for our intuition, while the state machine model may be a better fit for a nerd-centric view.

However, we offer no firm research evidence for such conclusions. In the interest of full disclosure, this is an area where we believe additional research could bear fruit. However, lacking a complete picture is probably not a good reason to move to a more faithful picture, and we view DCI as an important step in that direction.

DCI fulfilling a bit of history

DCI is in many respects a unification of many past paradigms that have appeared as side-cars to object-oriented programming over the years.

- Though aspect-oriented programming (AOP) has other uses as well, DCI meets many applications of AOP and many of the goals of Aspects in separating concerns. In line with the fundamental principles underlying AOP, DCI is based on a deep form of reflection or meta-programming. Unlike Aspects, Roles aggregate and compose nicely. Contexts provide a scoped closure of association between sets of roles, while Aspects only pair with the objects to which they are applied.
- In many ways DCI reflects a mix-in style strategy, though mix-ins themselves lack the dynamics that we find in Context semantics.
- DCI implements many of the simple goals of multi-paradigm design, in being able to separate procedural logic from object logic. However, DCI has much better coupling and cohesion results than the more brute-force techniques of multi-paradigm design offer.

End notes

1. *A Personal Computer for Children of All Ages*, Alan Kay, Xerox Palo Alto Research Center, 1972 (http://www.mprove.de/diplom/gui/Kay72a.pdf)

2. IFIP-ICC Vocabulary of Information Processing; North-Holland, Amsterdam, Holland. 1966; p. A1-A6.

3. Working with Objects: The Ooram Software Engineering Method., Reenskaug, Trygve, Wold, P., Lehne, O. A., Greenwich: Manning Publications, 1996.

4. Traits: The Formal Model, N. Schärli, Nierstrasz, O; Ducasse, S; Wuyts, R; Black, A; “Traits: The Formal Model,” Technical Report, no. IAM-02-006, Institut für Informatik, November 2002,

Technical Report, Universität Bern, Switzerland, Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA

Share your opinion

Have an opinion on the ideas presented in this article? You can discuss this article in the Articles Forum Topic, The DCI Architecture.

About the authors

Trygve Reenskaug has 50 years experience with the development of professional software and software engineering methodologies. He is now a researcher and professor emeritus of informatics at the University of Oslo. 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, structured programming and a data base oriented architecture (1960). Object oriented applications (1973). Model-View-Controller, the world's first reusable object oriented framework (1979). OOram role modeling method and tool (1983). The premier book on role modeling (1995). He was a member of the UML Core Team, adding parts of the role modeling technology under the name of Collaborations. He has develped the DCI paradigm for high level programming of object system state and behavior. He is currently working on BabyIDE; a companion development environment for working with a program as seen in different perspectives such as Data, Communication and Interaction (DCI).

Jim Coplien is a Senior Agile Coach and System Architect at Nordija A/S, doing international consulting in organization structure, software patterns, system architecture, as well as software development in electronic design automation, telecom, and finance. In this 'blog, he reflects on his academic career pursuant to his visiting professorship at University of Manchester Institute of Science and Technology, his appointment as the 2003-2004 Vloebergh Chair at Vrije Universiteit Brussel, two years as an Associate Professor at North Central College in Naperville, Illinois, and extensive work developing some of the first C++ and OOD training materials. He is well-known for his foundational work on object-oriented programming and on patterns, and his current research explores the formal foundations of the theory of design, foundations of aesthetics and beauty, and group theoretic models of design structure. He most recent book "Organizational Patterns of Agile Software Development", co-authored with Neil Harrison, culminates a decade of research. His book "Advanced C++ Programming Styles and Idioms" defined design and programming techniques for a generation of C++ programmers, and his book "Multi-Paradigm Design for C++" presents a vision for the evolution of current OO design techniques into a more general and better grounded theory of software construction.