Six wise men and the elephant
(with apologies to John Godfrey Saxe)

Trygve Reenskaug's DCI architecture has emerged in the past year as an exciting advance in object-oriented software design. It is the culmination of more than a decade of work, work that arguably stands on the shoulders of giants. Trygve and I often give presentations or write papers about DCI and are invariably asked, "Isn't that just dependency injection?" or are told, "That's just mix-ins!" I'm writing this article first to clarify the distinctions between DCI and its siblings, and second to honor those who have been exploring these same areas for the past 40 years.

A long time ago (in the 1970s) in a land far away, the first elephant was born. It was a software entity called an object and it was born in a land in the heart of the former Danish empire, and was called Simula 67. Its origins are the lore of yore, but few people living today have seen one of these objects firsthand.

While Simula 67 was often heralded as the primordial object, history in fact would remember it much differently, as a kind of wooly mammoth, a shadow of what was to come. The real elephant existed largely as an ideal in the minds of the original elephant trainers Kristin and Ole-Johan. This was a vision of real-world objects like ships and cars, captured in terms that the program could represent, unifying the end-user and programmer mental models.

Though Simula 67 objects were true elephants in their conception and in the core of their DNA, the language itself was dominated by class constructs that belied the true vision behind it, giving it more the appearance of a wooly mammoth than of an elephant. It was unable to inspire others to the object way of thinking. Smalltalk was an evolutionary mutant from the left coast of the land of the Nacirema tribe [Nacirema] that produced the same kind of elephants, but with none of the external features of traditional programming languages. Its conceptualization and deep genetic structure were brilliant; each elephant even had a recursive meta-genetic structure. Alan, a leading Nacirema elephant trainer, would protest loudly for years that the Nacirema elephants were the true elephants and that their European counterparts were as old as the wooly mammoth and unproductive [Kay]. Nonetheless, it still had the hairy class-oriented-programming gene and so, in the end analysis, was just a mastodon — a Nacirema variant of the wooly mammoth. We credit Alan with naming the upbringing and care of these elephants as object-oriented programming.

There was also a right-coast Nacirema tribe of elephant trainers descended from the ancient Danish masters, named Bjarne, who would take the class-oriented concept even further. This led to a right-coast Nacirema population of mastodons, and they were much hairier than their left-coast conterparts.

Nonetheless, both of these varieties of Elephant had many admirers and keepers. Most of the admirers were little better than harmless and came around just to use the elephant dung for their gardens of programming. They, too, called their style of gardening "object-oriented programming" and, indeed, each such application carried a little essence of Elephant. Elephant-dung gardens sprung up everywhere as the object label became a sine qua non of the day's software development.

A few others wanted to train the new elephants to do old tricks, like persistent storage of large-scale data (object-oriented databases), GUI design (object-oriented GUIs, whatever that means), and even to serve as the foundation of software methods (like object-oriented analysis, whatever that means). While enough was written about these ideas to fill a university library with books, no one ever saw real elephants that could be credibly traced to any of these ideas. Yet an elephant is just an elephant, and there in an industry with all the solemnity of a circus, the pressure for the elephants to do tricks grew and grew.

Six wise keepers

Six wise keepers of the elephant were disenchanted with how far the elephant species had evolved since its birth in 1967. There were some problems with the elephant: it was sometimes too weak, sometimes too clumsy, sometimes too stupid, to carry out its tasks. Those out in the elephant-dung gardens didn't even notice, but these trainers did. These individuals were certainly the ones to know, as they were among the best elephant keepers and trainers of the day. They decided to blind themselves to the superficial external beauty of the elephant and to dig into its roots, its genetic code, to understand the very fundamental nature of its makeup. A fine and wise elephant (as all elephants are) named Horton offered his DNA for the task.

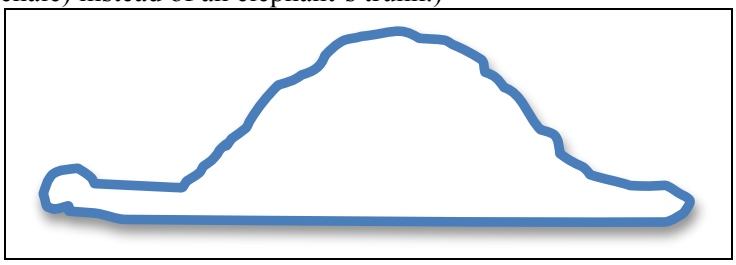
Each of the trainers analyzed the elephant DNA to identify places where it could be improved. One of the trainers, named Howard, analyzed DNA relating to the features of the hind part of the elephant. He found the genome for its tail and noted, "I have never thought of the elephant as being like a simple rope. Good elephants are complex things, and we need to think about the rope in terms of its individual strands. Instead of using pre-made, simple ropes, we want to weave together the cords that we want, to build a strong and yet flexible system." So he set about weaving elephants together from building blocks called mix-ins. Most of the mix-ins had interfaces that allowed multiple gardeners to regard the elephant as being whatever flavor they wanted. With his tool, called Flavors, gardeners were able to create chocolate, strawberry, and raspberry elephants. The basic elephant was vanilla. This experiment failed, however, among the dominant gardeners of the time (many of whom were tending C gardens before the elephant became widely known) due to too many jokes about vanilla (white) elephants and strawberry (pink) elephants.

In about 1990, an elephant trainer named Guy discovered a genome from around the elephant's head. He discovered the roots of the canvas-like structure of its earflaps. He noted, "The essence of elephant-ness is this structure similar to that of a circus tent. Elephants don't live in these tents, but they do tricks there in groups. People come to pay for the tricks, not the elephants. The tricks have an identity all their own, and we should take care that the right elephants together can do the right tricks." So Guy pioneered a technique variably called multiple dispatch or multi-methods that could orchestrate the dance of multiple elephants. Guy would later genetically engineer elephants into a tropical variant that became prolific but which lacked exactly the right meta-genetic structure that could have allowed it to adapt further.

In about 1993 two trainers named Dave and Randy picked up some genetic material, also from the head area of the elephant, and found the genome for tusks. The tusk is a spear-like structure used, among other things, for defense. One could very easily stereotype the elephant as a violent and warlike animal on the basis of this knowledge, and the elephant had unfortunately established that reputation. Now, not all elephants have tusks; it varies depending on their kind, their gender and their age. Dave and Randy said that if we were really to be concerned with helping elephants, we should treat them as individuals. After all, elephants are people, too, and elephants have been stereotyped for years: though they have good memories, they are conservative and escalate wars like Republicans. It's time to stop stereotyping elephants — even on the basis of their genetic possibility — and to start treating them as individuals. Dave and Randy would refine their work for many years to come, and they produced some of the most hairless elephants as history. Some of their work actually came round to some genetic engineering that helped elephants run faster. (It was an interesting trick: if an elephant had been in the same place twice under the same circumstances, it knew that it could do just the same thing as last time. And an elephant never forgets). But as for the core of their work, people had become so used to the hairy class-oriented coats of the mastadons that they found Dave and Randy's objects to be somewhat naked.

Another, called Martin, noted that elephants were noble and awesome creatures and, aside from their size, Martin strove to discover the source of their ability to inspire fear. In the genetic structure of the elephant he noted a gene for a long, narrow appendage. "Aha!" he

exclaimed, “elephants have a structure in common with the boa constrictor snake, so these species must be related — perhaps descended from a common ancestor. Perhaps snakes and elephants used to live in symbiosis, and the symbiosis became so great that the snake became part of the elephant.” He noted that problems arose when objects from more than one type hierarchy tried to work together separately. Perhaps, in the old days, elephants and snakes were spending far too much time in school learning about each other to sustain this symbiosis. Too much of the time, each had to know too much about the concrete class of the other. He continued, “Each class hierarchy can create its own object and inject it, like this snake, into another one of the objects, like an elephant, in a way that the two objects can be joined as one in symbiosis. He called it dependency injection,[Depedency injection] and his perspective developed many followers. Legend says that this elephant-snake relationship, later called the snabel, would become the foundation of Email addressing among the descendants of the original elephant herders. (There is nonetheless lingering confusion about whether it is really a pig’s tail (grisehale) instead of an elephant’s trunk.)



Two other keepers named Jim and Tim approached the elephant from the side and felt its tough, leathery skin. “Ah, this is so homogeneous and boring. I never thought of elements that way. This elephant is like a wall. Everything is combined into one, flat, undifferentiated surface — both the use case part and the domain behaviors are combined in the same interface. We need to differentiate the use case parts out into procedures — in fact, those aren’t very elephant-like at all, and we should probably even go beyond elephants for this part of the system.” So they each came up with a vision of multi-paradigm design. While Tim focused on differentiating computational models and Jim focused on differentiating structuring techniques, they both agreed that there should be different ways to express the procedural parts and the object parts. They both started talking about multi-paradigm design and programming in about 1993 or 1994.

Yet that division seemed too coarse. Even though the separation was good, it was difficult to put the parts back together. Gregor learned of this failure and proposed that it owed to the strict partitioning and coarseness of the division of classes and procedures. He noted that mix-ins had the same problem with respect to method granularity. His analysis of the elephant’s genetic makeup discovered the genes for legs. Discerning its shape with his hands, said, “This elephant is like a tree and, like a tree, has a strictly hierarchical structure. In fact, layered or hierarchical tree structures are too simplistic for the complexity of the software world.” He said that a clean separation of function and structure didn’t recognize the essential, complex cross-cutting nature of their relationship. He said that, instead, we should be using reflection to inject this functionality into the objects at finer levels of granularity. He called this the aspect technique: a way to do gene splicing at fine levels of granularity. Since most gardeners were C++ and Java gardeners, they didn’t understand this notion of “reflection” or, if they did, dismissed it as an academic toy that belonged to academic languages. The Nacerima radical right (the “east coast” programmers) abhorred it as a practice akin to vivesection. So in 1997 Gregor and his co-trainer Christina showed how it could be done even in the toy tropical language that Guy had created, with hopes that people would see how ugly the implementation was and, out of horror and repentance, would become true Lisp programmers. To his surprise, these tropical gardeners loved it and made it their own.

A common thread

In fact, what each of the wise men was able to see in his mind's eye was the limitation of some kind of strict, Cartesian classification scheme. Howard Cannon's Flavors [Flavors] were an attempt to move beyond a strict classification that forced every object to be of one class at a time, to one that permitted the class itself to be a composition of multiple class-like things. Multiple dispatch was an attempt to stop classifying methods in terms of their method selector and the single type of a single object, but instead to classify each method as potentially belonging partly to several classes at once. The self language [Self] tried to destroy the very notion of classification as found in a traditional class, and to return to the object foundations that drew objects from the end-user mental model. Dependency injection strove to blend the functionality of two objects into one. Multi-paradigm design refused to view the world according to a single classification scheme, making it possible to carve up different parts of the system in different ways. The goal of AOP is similar to that of mix-ins, except its cross-cutting units are more invasive at a finer level of granularity. They are like multi-paradigm design in that they allow a degree of separation of function and structure, but aspects' functional structure is much richer. It is more like having multiple knives carving up the same part of the system at the same time, whereas multi-paradigm design ensured that the knives didn't cross.

As all these fads came and went, this noble notion of Elephant perhaps lived on, unnoticed beneath the circus trappings of all the pretenders to object-oriented-dom. Though the brilliant folks in our industry each could see a part of the elephant, the idea of the whole eluded most of them. And we haven't even mentioned all of them. Delegation was another.

One stubborn old elephant trainer had known the first elephant but had difficulty finding it in any of the programming language circuses over the years. His name was Trygve. He was intrigued by the findings of the six wise men and was even more intrigued about how those findings related to his own memories of that dearly beloved first elephant. Maybe that elephant had just been a collective hallucination of other great men of the time, like Ole Dahl, Kristin Nygaard, and Alan Kay. No, he was pretty sure that the real elephant had once existed and had long since been forgotten.

The Smalltalk thinkers of the 1970s had a vision of a computation model that supported objects, and those thinkers struggled to express that vision in a language. Unfortunately, the language ended with building blocks called classes, which failed to capture the dynamics or richness of relationships in the objects. The real death of Smalltalk came when VisualWorks turned it from a framework for conceptualization into a GUI-builder.

(Photo by Steve Bloom. http://www.allposters.dk/)

What intrigued Trygve were those parts of the elephant that rarely made it into any circus, into any language or programming environment. Yes, as the six wise men found, the elephant did have parts: legs and trunks and tails and ears and tusks and skin. Just considering the four legs, however, it is laughable to think of them as tree trunks. Legs are for running. The complex motions of the legs of a charging elephant don't come from the legs; they come from an invisible brain and nervous system inside the elephant. Or at least that's true at least part of the time; sometimes, they are just to stand on, and sometimes they help the elephant swim (elephants are remarkably good swimmers). It is this charging and standing and swimming that intrigued Trygve more than the tree-trunk metaphor.

The last hurrah

A group, alleged to come from China, thought it would be worthwhile to dress up the elephant by describing all of these things in a book. Martin's notion of the trunk would become Template Method and Abstract Factory or Factory Method. Guy's ideas would be named Visitor. It was like dressing up the elephant. Some of them even confused the dependency-injected snake for a hat. It was one of the best-selling clothes line for elephants ever. It didn't change the elephant, but there is no arguing that it was one of the most enduring fashions in what is arguably the greatest fashion industry on earth. But it was a bit confused, because Howard's ideas also had to be implemented using Template Method, and many elephants ended wearing shoes on their head and hats on their feet.

DCI: the re-birth of the elephant

Trygve conceived of a paradigm of development called DCI that was rooted in the original notions of object-orientation. DCI is faithful to the original elephant.

Trygve carefully considered that there are two worlds: the dynamic world of the running program, and the static world of the program under construction. The former is tied closely to the programmer mental model, the latter is closely tied to the end-user mental model (which in turn is a learned part of the programmer's mental model). In the interest of being Agile, we want to accommodate the individuals in the system. We shouldn't limit ourselves to the end users. Programmers are people, too.

End-users tend to think in objects. Mette thinks of her kitchen telephone as an object. She may think of it in a way that is somewhat free of the context of its use at any given time. While in general it has something to do with communication (calling friends or reporting a fire in the kitchen) it can also be used to hold down a recipe while she is cooking so the breeze coming through the kitchen window doesn't blow it away, or she may use its address book facility to look up addresses. I think of my savings account as having a particular balance and as being in a particular bank, without any particular regard for how I am or am not using it a the moment, and with no regard to anyone else's savings account. When I am working as a circus worker struggling to raise a tent pole under a tent, I care about this pole: its dimensions (does it fit?), its weight (should I get an elephant to help me raise it into place, or is it a small enough tent can I do it myself?). The general properties of tent poles disappears.

Whereas end users tend to think in objects, today's programmers tend to think in classes. The evolution of programming languages has all but stamped out the notion that gives first-class standing to any notion of task sequencing. Java completely outlaws it, forcing all activities to take place in the context of some class. In a world where classes are all that we have at programming time, they have to capture a compression of all end-user concepts. Contemporary programmers use classes for at least two things: to capture that part of the end-user mental model that relates to the concrete things in their world, and to capture bits of how the system behaves in the real world. So we find classes like Savings Accounts that characterize my savings account; Telephones like Mette's telephone; and tent poles like one in a camping tent.

Yet in a world where everything is a class, and where we want to capture key mental models in the code, the classes must also capture the system activity. This is like the nervous system of the elephant. But classes don't capture activities like "running:" the capture only the primitive joints and muscles of the legs and perhaps capture their orchestration that helps propel one quarter of an elephant. Those activities might be aspects or mix-ins — but we've seen their limitations. And then there is the coordination between the legs, which a methodologist might ascribe to the grand, nebulous concept of elephant. That's wrong, too, because elephants are not the only four-legged creatures with those gaits and cadences. Running is a concept unto itself; it is not a method of an elephant object. Even if it is, such knowledge doesn't help us understand or manage its relationship to the legs.

What DCI does is to bring object-orientation to the current class-based world of programming.[1] The idea is simple. First, we divide code into the program's basic domain data on one hand and its functionality, or behavior, on the other. The domain structure is like legs and trunks, and the functionality is like running or taking a bath. The domain structure we call Data, the D of DCI. They are just barely smart data. Most of the "smarts" belong in the roles rather than in the data, and object orientation has long made the mistake of mixing these two in the same interface. The roles capture our mental chunking of the behavior into algorithms, which are the archetypical Interactions between the objects. We represent these as Interactions between roles, generalizing above the object level. Interactions are the I in DCI.

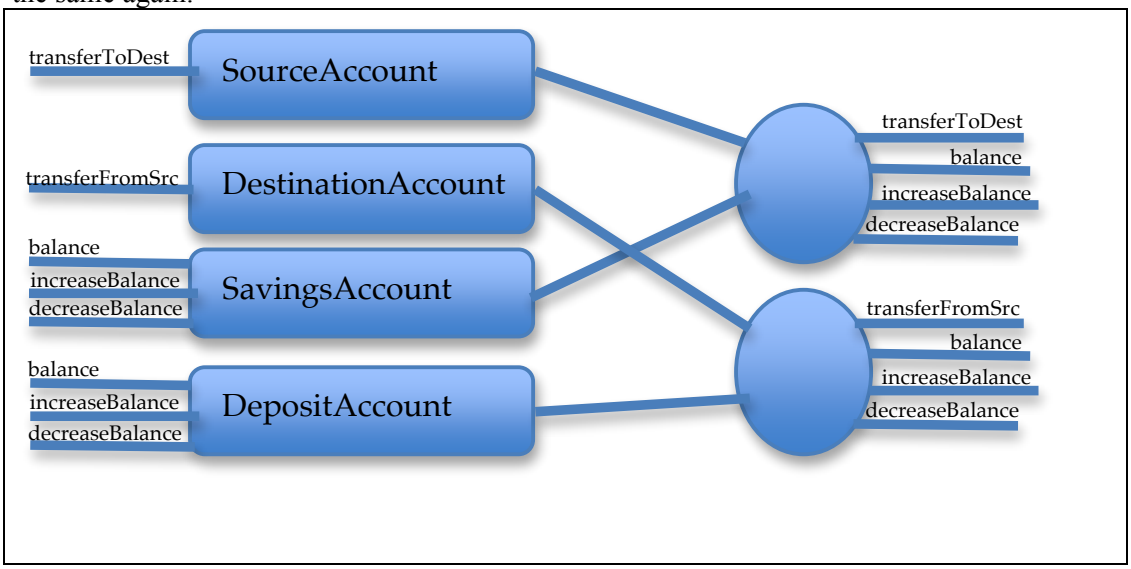
Second, we partition each of these spaces. Partitioning is inevitable because human minds can understand small things better than large things, so we build on a basic psychological skill called chunking. In the domain structure we use the same, time-honored chunks as we always have in object orientation: classes. In the world of behavior, we divide up long algorithms in terms of sequences of operations on roles. This is a good fit for human mental models. When we conceive of how to transfer money from some source account to a destination account, we indeed think of the concepts source account and destination account

[1] There is a more general way of looking at DCI which doesn't depend on traits. In this article we are trying to take current programmers from what they know to what they might want to know. Because most contemporary programmers are working in classful languages, we investigate that corner of the DCI world here.

— not the classes SavingsAccount or DepositAccount. Either of those objects can play any of the roles. So we subdivide the system's algorithms along role boundaries. Roles tend to collect multiple related behaviors

We have to map this conceptualization onto programming languages. We often work in programming languages where classes are the only construct for grouping objects (as in C#, Smalltalk, C++, and so forth). That means that classes have to stand in for roles.

Next, we need to be able to "teach" specific objects how to play specific roles. Consider a system that is based on this separation of functionality and domain knowledge. Domain object are sitting around in memory waiting for work to do, and a use case arrives. Think of it like the start of a play. The use case is the script. Each object will be asked to play a role. So we need to educate each object how to play its role. That means injecting the role logic into the objects that need it. It's like hooking up Neo or his girlfriend to the Matrix and downloading a program into him that will allow him to fly a helicopter. Now, the curtain rises, and: Lights, camera, action. Each actor can play its role. The curtain falls, there is applause, and the play has been carried out. Neither the actors nor the audience will ever be the same again.



Teaching each object its role in real time is the tricky bit. If the script fragments are inscribed in class code, and if each actor is an instance of a class, we must compose those classes together — at least for those of their objects which will play those roles. Schärli [Traits] describes this approach as traits. Each chunk of role functionality is a trait that can be injected into the objects that need to play that role. Traits appear in different forms in different programming languages. In Scala they are first-class citizens. In Ruby, we simulate them using modules instead of classes. In C++ they are templates. In some languages this injection happens at run-time, at the last microsecond; in others like C++ it is arranged at compile time. Scala, which actually does all the work at compile time, it still gives the illusion and feeling of doing the work at run time:

```
val source = new SavingsAccount with SourceAccount
val destination = new DepositAccount with DestinationAccount
```

Last, we need code that knows how to find the right objects for a use case (to hire the right actors for the play) and to inject the right role logic into each one (to have each actor learn its part of the script). This code is called the Context. It is the C in DCI.

Yes, there is hope that languages will give more prominence to objects and less to classes over time. In Python and to some degree in Ruby we see the ability to freely mix object and class operations with a uniform syntax and semantics. Javascript and Groovy are similar. It is possible for a single object to have a unique method independent of its class. Scala helps support this illusion, though there is actually class support behind the scenes. And

this illusion can be supported even in static languages like C++. With some instructive use of the MOP, we can make even the Smalltalk elephant dance.

Mix-ins

Howard noticed that elephants could be complex things with many behaviors that we think of independently. Furthermore, depending on which of these behaviors we chose, we could make many different kinds of elephants. It was a brilliant notion and very close to Trygve's notion of DCI.

However, DCI is much more than just mix-ins. What mix-ins lack is a framework to tie them together, and in particular to tie the mix-ins of one object with the mix-ins of another. Mix-ins reflected a hierarchical composition. They worked well one elephant at a time, but you couldn't use them to make elephants follow each other around the circus ring or to stand all at the same time, let alone put their forelegs on the back of the elephant in front of them.

Roles are in fact mix-ins that have a class-like feel to them. Each role is a collection of behaviors; what it means to be of a certain role is to be able to be grouped by that behavior. Anything that is large, grey, has a trunk, and lives in trees must be able to play the role of an elephant.[2] Perhaps a mastodon with a haircut could play the role of an elephant in a play. Roles are like classes in that they are groupings of behavior, but they differ from classes in that they are partial classifications while classes are total classifications. So every elephant object is of class Elephant. However, even though a shorn mastodon is of class Mastodon it can technically play the role of an elephant according to the above definition.

Multiple dispatch

Guy noticed that the actions of a system depended on the type of more than one kind of thing in the system. More important, he could see the algorithm's high-level behavior standing as a separate concept independently of the objects or classes involved. He did this out of necessity: some of the more complex operations involved multiple objects, so no one of them could own the operation. The method "run" in the context of a human being likely applies to two objects of type leg, whereas it applies to four leg objects in the context of the elephant. The two algorithms are quite different, and we should choose the right variant.

However, DCI is much more than just multi-methods. What Guy missed is that large parts of these algorithms are often invariant across the context of invocation, and other parts could be factored into the objects themselves. There is a fundamental, universal process of running that is invariant with respect to the objects to which it is applied, and multi-methods don't really capture that. We need to reinvent running from scratch every time.

Multiple dispatch is an advanced form of run-time polymorphism. Most pundits of old-fashioned object-oriented programming hold polymorphism to be one of its most fundamental features. Most class-oriented programming language support it directly by a special facility of class-oriented programming. A programmer can arrange for all objects of a derived class to override the function of the same name in the base class. It is a nice way for the programmer to organize these functions at run time. However, this style provides that the method or member function, and not even the class or object, is the unit of polymorphism. If a class supports methods A and B, one may be polymorphic while the other isn't. That means that the programmer needs to look at multiple classes in the inheritance hierarchy to weave together a picture of the object's business behavior. Furthermore (though unrelated to multiple dispatch), the programmer must repeat this exercise for the class hierarchy of each object involved in a given use case. If we know at run time that method A in object O1 calls method B in object O2, we don't in general know how to find the source code of either function.

---

[2] We apologize for lieing about the trees. See Ron Brachman, *'I Lied about the Trees' or, defaults and definitions in knowledge representation*. AI Magazine, 6(3), 1985.

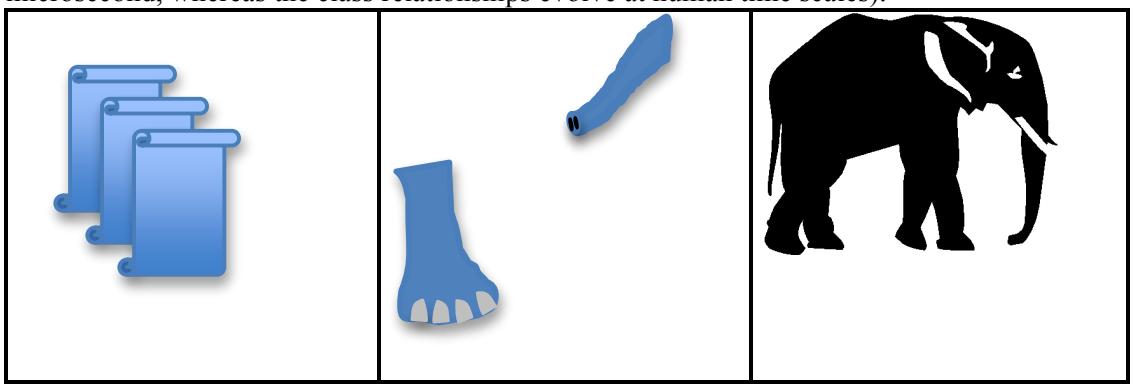
DCI minimizes this kind of polymorphism to the simple, primitive operations of the domain objects. If the use case is a complicated simulated annealing, then we might override methods like + and / on objects like ComplexNumber or Integer. This kind of polymorphism usually doesn't cause programmers to stumble as they try to understand code. The logic supporting the business use case is in the roles and does not use this kind of polymorphism.

Classless programming

Dave and Randy noted that classes are only documents that described what an elephant looked like. Like Trygve, they understood that there traditionally had been a distinction between two worlds: the programming world and the execution world. They suspected that the dichotomy might be artificial. The also knew that the instance perspective was important, and that it was more in line with the original elephant than class-oriented programming languages.

However, DCI is more than just giving stature to objects. Whereas Dave and Randy could see the objects that made up the elephant, Trygve noted that it was also important to see the whole elephant. That means understanding the elephant as a system. To focus on individual objects is to lose the system view, and you can't see the elephant for the trees. Furthermore, though Dave and Randy tried hard to expunge classes from their system, there are still class-like entities down at the bottom. They don't call them classes but they are still the programmatic constructs that developers ultimately rely on, statically, in a way that compresses the run-time semantics down to something that fits into the programmer's head.

In the end, we find that the programming / execution dichotomy isn't a wholly artificial one. The two environments vary wildly both in scale (dozens of classes compared with millions of objects) and stability (the object graph is being re-created microsecond by microsecond, whereas the class relationships evolve at human time scales).



So what makes an elephant a system? Of course, it has what we recognize as "parts" (though, as in software, the parts are not as separable as children's building blocks) or objects: legs and trunk and tail and tusks, for example. But there are parts we can't recognize from the outside, such as the elephant's nervous system — the locus of coordinated activity between the legs of a charging, standing or swimming elephant. This nervous system provides the context for the objects to interact. The nervous system is, of course, also another one of the parts of the elephant. But it's not a "part" in the common English sense of a highly de-coupled element that has locus or identity. It is, in fact, a reification of the coupling itself — a higher-order object than those other parts.

Instead of focusing just on objects as the loci of all things important, DCI realizes that we need several perspectives that cut across each other. Yes, we still have the object view. But we also have a nervous system, called a Context. Each Context is home for a coordinated activity between objects. It establishes connections between objects that work with each other to make the system do what it does; the legs and trunk and tail and tusks are just what it is. For an elephant, this might mean running or standing or swimming. For a piece of software, this "nervous system" is the use case that ties multiple objects together. Like use cases, Contexts describe their activity in terms of roles. A role is a way of looking at a particular

elephant while wearing blinders: we see only as much of the elephant at a time as we need to carry out the work. In fact, each Context encodes the object relationships in terms of the interactions between the roles they play. These "interactions" are the I of DCI.

Dependency Injection

We can look at dependency injection as one way to combine the base domain code of an object, and the code of the role it plays, into one. One can choose from many roles in the what-the-system-does world and associate each chosen role's behaviors with a kind of object to play that role at run time. Dependency injection can relax the static dependency between these two worlds so we can achieve a more dynamic vision of object orientation than with the more static mechanisms of most programming languages.

However, DCI is more than just dependency injection. Dependency injection leaves us with two objects and an identity crisis; DCI does not. The difference is that DCI changes the behavior of an object by changing its type run time (or of its class at compile-time in a way that anticipates the run-time need). Dependency injection changes behavior through patterns like Abstract Factory or Factory Method, each of which bind the variant behavior to instantiation. We want instead to bind and re-bind the behavior of a given object. It's true that mix-ins are a weak form of dependency injection that allow us to do this, but they alone aren't enough.

In fact, some DCI implementations use a form of mix-ins to represent roles (as we described above). Most programming languages have no first-class concept called role, but represent roles as classes. To do the mix-in thing we need to be able to inject the roles into the objects that play them. We can do that at compile-time using mix-ins. We can do it at run-time by manipulating the method table. Whether they are injected at run time or compile time, the role logic we inject is called a trait. We tend to use the word role for the analysis concept and the word trait for the programming constuct.

The Scala programming language has traits as full first-class citizens.

Multi-paradigm design

The multi-paradigm approach of Jim Coplien presumed, much like Howard, that behaviors could be represented more like the trainers that train the elephants than like the elephants themselves. Just put the behavioral part of the system into its own collection of procedures, and it will be just fine. It is essentially an ad-hoc version of multi-methods. This puts one part of the system in charge over another: the imperative parts in one place and the domain parts in another.

However, DCI is much more than just separating form from function. The roles are missing. Roles establish a structure over the functionality, whereas Coplien's approach establishes a structure over the domain data but says nothing about structuring the algorithm. More seriously, Coplien said nothing about the mapping between them — and that's where the real architectural issues of coupling and cohesion arise.

Aspect-oriented programming

Gregor almost got it. Whereas multi-methods His initial instinct about the importance of reflection is probably one of the great unsung insights of programming history. Dependency injection fails to achieve DCI's goals because it is at the level of the program instead of the meta-program. The run-time type handling beneath multiple dispatch is too weak to achieve DCI's needs, and perhaps reflection could help. And unlike most of the other approaches, aspects retain object locus and identity.

However, DCI is much more than just aspects. Where Aspects went wrong is in their one-sided focus on the imperative side of design. In common use, aspects tend to capture elements that are more part of the programmer model than of the end-user model. A banker may talk about logging a transaction, but doesn't design a log with respect to the detailed actions that can deposit things in it.

DCI is designed to make code readable, and in so doing, raise the likelihood that it is correct. AOP tends to be a good short-term convenience for the programmer, in making it possible to quickly inject small snippets of code in multiple places across the program. However, cutpoints [right term?] (see if they make the code more unreadable).

A bigger picture

Aspects, dependency injection, multiple dispatch, and mix-ins have become the modern trappings of programming and replace the primitives that dominated programming discussions of yesteryear: procedures, classes, inheritance, and so forth. Yet these are just trappings of programming. Object-oriented programming — the whole elephant — is more than that. Alan Kay, who coined the term "object-orientation," tells us:

> In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computer all hooked together by a very fast network.

If we take a historical perspective on what went wrong, it probably goes back to the Smalltalk programming language and its class model. Smalltalk, along with Simula 67, were the programming language standards against which all other object-oriented programming languages were judged. Smalltalk overshadowed Simula and earned the reputation of a more highly evolved elephant than its predecessor. However, in spite of having a much more highly evolved run-time object model than Simula, its language still clung to a Simula-like class model for programmers instead of taking a more distinctive evolution path. Though C++ was based more on the older Simula than on Smalltalk, a more distinguished evolution path might have allowed it to overpower C++ in the market. The C heritage of C++ became the meme for the whole menagerie of 1980s programming language and, as Richard Gabriel notes, the end of programming ensued [The End of Programming].

Don't think about elephants

Indian and African elephants

Kay: The Computer revolution hasn't happened yet.
http://video.google.com/videoplay?docid=-2950949730059754521#

Nacirema: Horace Miner, Body ritual among the Nacerima,
https://www.msu.edu/~jdowell/miner.html

Dependency injection: Jakob Jenkov http://tutorials.jenkov.com/dependency-injection/index.html,

Flavors:

Self: Self, the power of simplicity. http://research.sun.com/self/papers/self-power.html

Traits: Schärli, Nathaniel, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black, "Traits: Composable Units of Be- havior," Proceedings of European Conference on Ob- ject-Oriented Programming (ECOOP'03), LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248—274.

The end of programming. Gabriel, Richard, Good News, Bad News, and How to Win Big,
http://www.ai.mit.edu.docs/articles/good-news/good-news.html