DCI in 1585 words.

Most articles about DCI are long, somewhat complicated and often philosophical. It didn't take me very long at all to grasp the concepts behind object oriented programming (OOP), when I read about it. And it shouldn't take long to grasp the concepts behind DCI either. So instead of wasting words, let's get on…

Two of the most important things about creating a computer program are a) the data and b) the algorithms. With procedural programming, data cannot be programmed particularly well (no abstraction, no inheritance, sometimes even poor support for data structures), but algorithms can be programmed fine. With OOP, it's the other way around, it's great for programming all kinds of real world data structures, but it isn't so great for programming algorithms, because the algorithms described in use cases, tend to get fragmented into lots of classes, because behaviour is kept close to its data (i.e. "coherence"). The relationship between algorithms and use-cases gets lost quite easily. It doesn't have to, you are free to program procedurally in most modern OO languages, but if you do OOP the way it was intended, fragmentation happens. OOP also suffers when people over-engineer the inheritance trees. It doesn't have to, but it depends upon how good the designer is. The complex Bridge Pattern from the Gang of Four was created to help reduce inheritance.

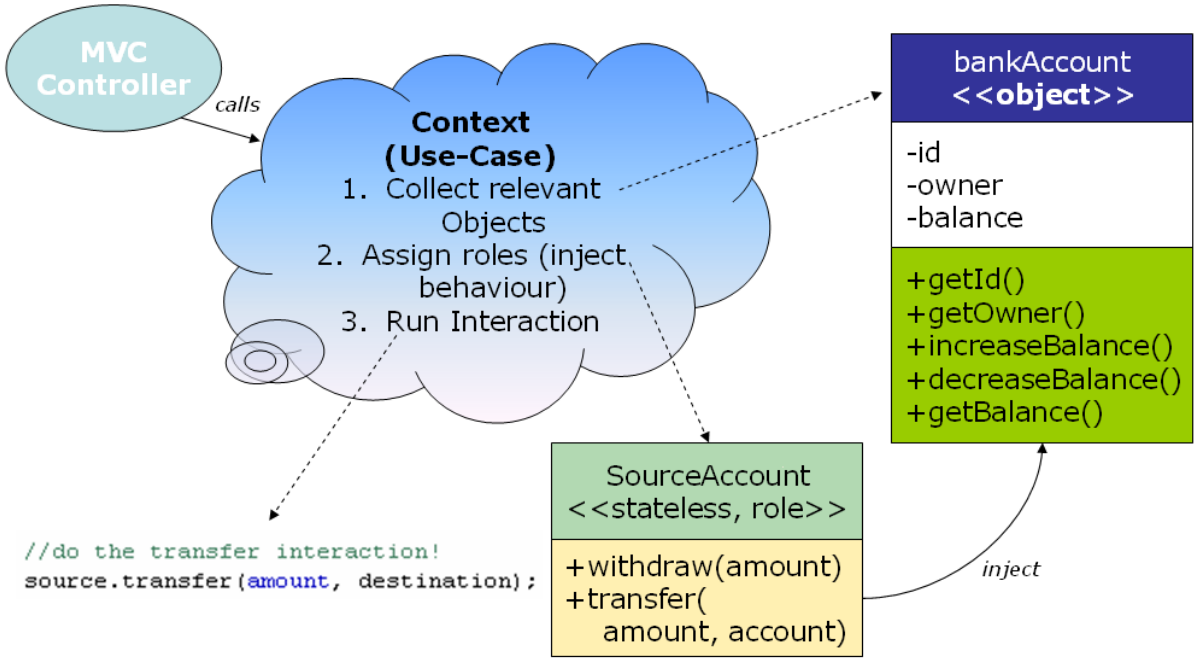
The fragmentation can be bad for two reasons. Firstly, it can make reviewing use-cases hard. Either you give the reviewer a set of entire classes, including a bunch of stuff unrelated to what they want to review – or you spend time cutting and pasting methods and data out of source code into some kind of document to give to the reviewer, and they end up with parts of more complex classes and they end up confused, and you spend way too long creating that useless document. Both ways makes reviewing hard. If reviewing cannot be relied on for catching bugs or other problems, then only testing can be, and testing is expensive. And most people are lazy and don't like testing. The second reason fragmentation can be bad is that it's basically the same as mixing up algorithms and data. And that can be bad, because, especially in the agile world, algorithms change faster than data. It's quite easy to capture an entire data model from the requirements and to program it up. It doesn't often change much, it may get extended. But capturing the functionality (algorithms) is hard, and they are guaranteed to change, as the requirements grow from iteration to iteration or as scope creep occurs. And because we love our friends who create the requirements (they give us money to *program*[1]), we hate telling them that they can't have functionality which they need, because adding it would break our well designed software. So we hack around and add functionality, and our code starts to rot. Alternatively, we refactor, and that costs a lot. If we could split the data from the algorithms, our code might rot much slower.

Services (ala Service Oriented Architecture) have been around for a decade, and do this splitting really well. But SOA struggles to be similar to what the user has in his head when creating requirements – when was the last time you read a requirements document or user story with the word "service" in it? The problem is that service design does not dictate a close mapping to the use-cases. So there is a mapping between the user's mental model and the software (the programmer's mental model). And that can be bad, because anyone reviewing the code has to understand the mapping. And anyone picking up the source code after a few years has to understand the mapping. And that mapping is either undocumented, or in a design document, which not actually necessary, because the mapping itself is not necessary. That design document also gets forgotten when we start adding functionality, or when another programmer starts adding code. And it gets worse. If during analysis, you use CRC cards or similar techniques, you end up with two mappings – one from the user's mental model to the CRC cards (OO world), and a

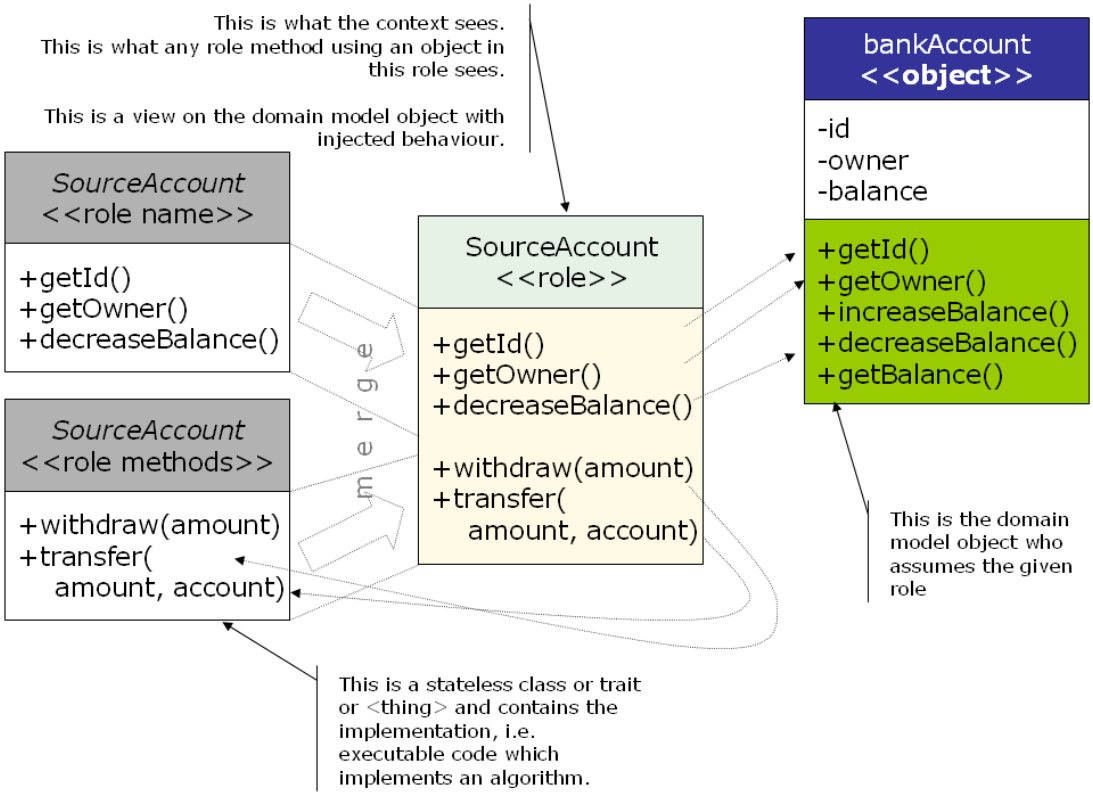
[1] If only they knew that wasn't necessary, we love programming! But don't tell them that.

second, from CRC cards to services! In this respect, OOP is better, because you only need one unnecessary mapping.

So some clever people living in Scandinavia, using their own ideas and some others from around the world, created something called DCI. It stands for Data, Context, and Interaction. And it takes separating data from algorithms a few steps further, by using the idea of roles. It goes like this: data objects (aka "D") require different functionality, depending upon the context (aka "C") in which they are used. Within a given context, they interact (aka "I") with other objects. Instead of letting complicated objects interact with other complicated objects, objects are assigned roles, to make them less complicated. The roles also provide the objects with the algorithms they need to be functional in that context. The context (which starts the interaction), or indeed other objects within an interaction, only know about objects in their particular role – that is, they have a smaller (narrowed) interface when compared to them being plain data objects. They only expose the things important to the role they are playing, as well as the new functionality which is part of the role. So, all these words and no pictures?! 895 so far, so we are more than half way! In the following picture, start at the MVC controller in the upper left.



And here's another picture:



Those two pictures had some interesting words on them: "inject", "role name" and "role method".

In order to have algorithms, one needs to code them somewhere, and we said above, that this wouldn't be in the data classes (that's what OOP does). So we code the algorithms in "role methods", i.e. methods in stateless classes, or if your language of choice supports them, "traits".

Above, I said that complicated objects need to be made simpler… that's where "role names" come in. These are a definition of methods which the object will expose while playing the role.

So, that leaves the word "inject". This word is about providing the functionality to the object so that it can play the role. It's controversial, because how do you inject algorithms into data objects, using main stream languages? Well, if your language of choice doesn't let you do it, you can pretend it does, using things like dynamic proxies, or just the proxy pattern, or other mechanisms like composites. You have to be careful to avoid "object schizophrenia", because in pure DCI, an object assuming a role *is* the data object, not a new one, and if you hack this by using a proxy, the object exposing the role methods is *not* the data object. But the important point is that anyone reading the code reads it so that they are telling the same story which the use case tells. All these words (1162 so far, three quarters done) and some pictures, and now you want to see some code? OK. Imagine a design with some Model-View-Controller in it, and the user does something to cause the controller to get called by an event. That controller creates a context in order to do something useful, rather than do something useful itself. That context is responsible for firstly casting objects into relevant roles, and secondly for running the interaction. Role methods are the inner workings of the context and the context may also contain algorithms which hare not directly the responsibility of roles. OK, OK! Here's the code:

```
def withdraw(accountId: Int) {

    var source = new BankAccount with SourceAccount_Role

    source.withdraw(100)

    println(source getBalance)
}
```

Prefer that in Java, rather than Scala? OK, but be warned, it is not pure DCI[2].

```
// prepare the injector (dynamic proxy)
BehaviourInjector behaviourInjector = new BehaviourInjector();

// convert the domain object into a role, and inject the relevant
// role methods into it
ITransferringSourceAccount_Role source = behaviourInjector.inject(
    sourceAccount, //domain object
    TransferringSourceAccount_Role.class, //class providing all the impl
    ITransferringSourceAccount_Role.class); //the role interface

//now do the withdraw interaction, get some cash and go shopping!
source.withdraw(amount);
```

The last bit is then to look at what happens inside the role method called "withdraw":

```
/**
 * check its the logged in users account, check the account has available funds,
 * withdraw money from the account and write ledger entry.
 */
public LedgerEntry withdraw(BigDecimal amount) throws InsufficientFundsException {

    checkSecurityForWithdraw();

    checkAvailableFunds(amount);

    //decrease balance
    self.decreaseBalance(amount);

    //create ledger entry
    String comment = "Withdrawal";
    return createLedgerEntry(self.getUid(), amount, self.getBalance(), LedgerEntrySide.DEBIT, comment);
}

/** checks whether the user is a {@link Roles#CUSTOMER}, and whether they own the account */
private void checkSecurityForWithdraw() {
    //check security
    if (!sc.isCallerInRole(Roles.CUSTOMER)) {
        throw new SecurityException("Wrong role!");
    }
    if (!sc.getCallerPrincipal().getName().equals(self.getParty().getLogin())) {
        throw new SecurityException("Not account holder");
    }
}

/** checks if the account has a balance above zero */
private void checkAvailableFunds(BigDecimal amount) throws InsufficientFundsException {
    if(!self.hasAvailableFunds(amount)){
        throw new InsufficientFundsException();
    }
}
```

[2] See http://www.maxant.co.uk/tools.jsp for details of the `BehaviourInjector` class which creates a dynamic proxy.

```
protected LedgerEntry createLedgerEntry(int accountId, BigDecimal amount, double newBalance,
                                                    LedgerEntrySide side, String comment) {
    //need to create an object, so that JPA can set the foreign key
    BankAccount account = new BankAccount();
    account.setUid(accountId);

    LedgerEntry le = new LedgerEntry();
    le.setAccount(account);
    le.setAmount(amount.doubleValue());
    le.setNewBalance(newBalance);
    le.setSide(side);
    le.setWhenDt(new Date());
    le.setComment(comment);

    em.persist(le);

    return le;
}
```

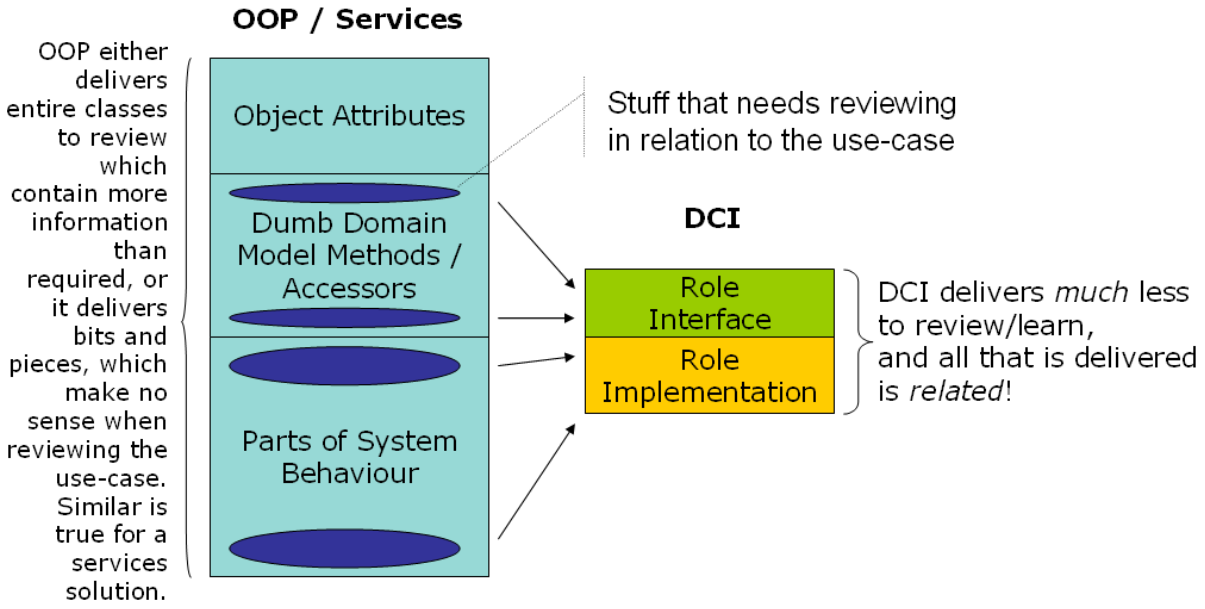
Hey, what's "self", in the code `self.decreaseBalance(amount)`?? Self, is the object, currently playing the role being executed. It's a bit like "this", but "this" would refer to the instance of the class implementing the role methods. Where does it come from? In Java, in this given implementation of DCI, it's injected at the time when the data object is cast into the role:

```
public class TransferringSourceAccount_Role {

    /** injected by the {@link BehaviourInjector}. */
    @Self
    protected ITransferringSourceAccount_Role self;
```

You might not agree with the code, i.e. what it does. But this is an introduction to DCI, not a discussion about what the use case is, how it should be defined, or how it should be coded. The important point is that you see how to implement DCI, and to understand what its benefits and disadvantages are.

See how this code is reviewable, compared to OOP? See how reading the code sounds identical to the use-case?[3]



[3] The actual use case has not been printed here, because this is a succinct article, but if you read the code, and write it out, that would be the same as the use-case ☺.
Code and actual use case are downloadable from www.maxant.co.uk/whitepapers.jsp

Let me close with pretty bullet points.

DCI is a paradigm used in computer software to program systems of communicating objects. Its goals are:
To improve the readability of object-oriented code by giving system behavior first-class status;
To cleanly separate code for rapidly changing system behavior (what the system *does*) from that for slowly changing domain knowledge (what the system *is*), instead of combining both in one class interface;
To help programmers reason about system-level state and behavior instead of only object state and behavior;
To support an object style of thinking that is close to peoples' mental models, rather than the class style of thinking that overshadowed object thinking early in the history of object-oriented programming languages.

DCI makes reviewing / reading / understanding code easier by...
...formally separating the Domain Model (what-the-system-is) from System Behaviour (what-the-system-does);
...narrowing each object involved in an interaction so that anyone reading the code only need to know about that narrowed interface;
...formally relating implementation directly to use-cases;
...relating unit tests, directly to use-cases because when unit tests are written for roles, they automatically test use-cases, rather than unit testing random parts of use-cases;
...making it very clear where you need to invest your testing effort, i.e. unit test the roles.

DCI...
...reduces the size of the inheritance tree of the domain model, when compared to OOP;
...makes reading the code sound more like the users mental model, when compared to a services solution;
...can be combined with frameworks or application servers which consider cross-cutting concerns like transactions and security, as well as other concerns like resource management, concurrency, scalability, robustness and reliability – it is the context which the container knows deals with.

So, hopefully you now know what DCI is about. DCI is not a one-size-fits-all solution and it is not suited to every application. But if you value reviewable code, and want code to map directly to use-cases, DCI empowers you to do that. DCI does not need to be used everywhere in your code either, it can be used to solve local problems. Even if you don't use DCI, DCI formalises a few important things:

- Separating System Behaviour from Data is a good thing,
- System Behaviour is first class, just like Data,
- System Behaviour should be use-case-centric, rather than class- or service-centric,
- Reviewing code is better than just testing it,
- Reviewing code can be easier, if the data objects involved have a narrowed interface.

For more information, check any of these useful links:

http://en.wikipedia.org/wiki/Data,_Context,_and_Interaction	http://scg.unibe.ch/archive/phd/schaerli-phd.pdf
http://www.artima.com/articles/dci_vision.html	http://vimeo.com/8235394
http://heim.ifi.uio.no/~trygver/2010/DCIExecutionModel.pdf	http://vimeo.com/8235574
http://architects.dzone.com/videos/dci-architecture-oberg	http://www.maxant.co.uk/whitepapers.jsp
http://heim.ifi.uio.no/~trygver/2009/commonsense.pdf	http://groups.google.com/group/object-composition