The DCI Architecture

Lean and Agile at the Code Level

James O. Coplien

Gertrud&Cope, Mørdrup, Denmark

Scrum Training Institute

The misguided, CASE-heavy practices of the 1980s fueled the proto-Agile rhetoric of the 1990s and survived full-force into the advent of Agile practices such as Scrum and XP in the past decade. Part of that rhetoric has been to go as far from the sins of the 1980s as possible by discarding up-front requirements (instead we have a promise for a future conversation between a developer and a customer) and architecture (instead, we have had a succession of short-lived ideas including "metaphor" and TDD). Experience and recent research have both borne out the value of architecture in software development in general, as well as its value in sustaining high velocity and change resiliency in Agile projects. In this talk, Agile expert Jim Coplien will provide tips for putting architecture back into your Agile project without dragging it back into the dark ages -- and all within the framework of the Agile Manifesto.

1 hour -- 4 minutes per slide, 15 slides

What is architecture?

- Architecture is the essence of structure: *form*
  - Structure obfuscates form!
- Lean architecture: just-in-time delivery of functionality, just-in-time pouring material into the forms
- Agile architecture: one that supports change, end-user interaction, discovery, and ease of comprehension (of *functionality*)

Let's start with a few basic definitions to establish common ground.

This is a talk about Lean and Agile architecture. What is architecture? Throughout the ages and even in the field of building construction, architecture has always been about form. Form is the essence of structure. The word in English for a piece of material, into which molten metal or plastic are poured or injected to make a new something, is called a form. We focus on form because if we look at structure, it actually makes it difficult to understand the essence by adding a lot of detail which is irrelevant early in design.

What, then, is Lean Architecture? The principles of Lean include just-in-time delivery, careful organization, up-front planning, avoiding waste, and overall consistency. Architecture is in principle about consistency. If we have a consistent system early on that is stable over time, it avoids rework and waste. That takes up-front planning and careful organization. That in place, we have a guide — a form — into which we can write code when the need comes along.

A good architecture supports Agile development. A good architecture is Agile if it supports change and has just enough documentation to do the job — the documentation that really makes a difference. Good architecture supports the Agile values: support for change, a good model that supports the end user's mental model of the system and that helps the developer's discovery process. Here, we will focus on helping the developer reason about functionality in the code — functionality that reflects the end user wishes.

What is its value?

- Architecture supports "what happens there"
- Habitable code — by the people who develop it and the people who use it
- Architecture is what makes code feel familiar
- A good architecture reduces waste and inconsistency — muda and mura
  - Less rework
  - System consistency

Architecture is what is, but architecture support what the system does. A good architecture makes the code "comfortable" to have around and to use. We use the word habitable, a term borrowed from the pattern community (which emerged from building architecture).

A good architecture reduces waste and inconsistency. These are key Lean values: muda and mura. It means better overall consistency and less rework and waste.

Architecture and OO

- OO is a paradigm — a way of talking about form
- OO's foundations: to capture the end user's mental models in the code
- OO captures
    - The entities (objects) that users know about
    - The classes that serve as sets of such objects

4

Switching gears a bit, let's turn to the JaOO theme: objects. Object orientation is a way of taking about form. It's not the only way, but let's start there.

What is object orientation anyhow? Instead of starting with the tired old coupling-and-cohesion definitions (which apply equally well to modules or procedures) or polymorphism (even overloading is a form of polymorphism), let's go to the goals of its originators. Dahl and Nygaard were striving to capture the end user's mental model in the code. Good OO captures the objects that users know about. Most object-oriented programming languages uses classes as a kit for building objects; that is largely an issue of history, and comes from the fact that Dahl and Nygaard chose to base their simulation language in Algol. (Well, they were paid to write an Algol compiler and were allowed to build Simula into the product.) But the goal is still focused on the user and the user mental model.

What is that model? The traditional OO view is that it's all about getting the objects right.

MVC: The Embodiment of the OO Vision

- User model -> into the code -> presented back to the user
- The goal of *views* is direct manipulation

The goal of the *controller* is to coordinate multiple views

mental model

computer data

User

Model

mental model

computer data

Controller

1

*

View

*

*

Model
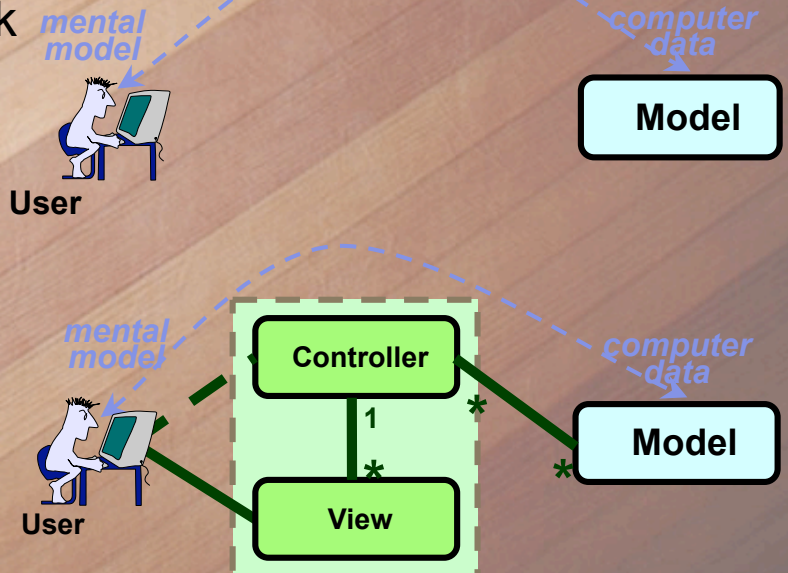
5

A quick side note: One of the other people in the wings in the early days of OO was Trygve Reenskaug. Trygve was disenchanted with Simula because of its lack of encapsulation (it was difficult to send a message to a receiver that you didn't create) and with Smalltalk because of the domination of classes in the programming environment — when it should have been about objects.

In 1978 Trygve brought the user into the picture with a four-part pattern called Model-View-Controller-User. Its first goal is to let users interact directly with the code that itself was desigend as a reflection of their own mental models: that's what View is for.. Its second goal was to allow the user to coordinate several simultaneous views of the same model; that's what Controller is for — to create and coordinate those views.

This, again, is based on the simple model that it is sufficient to model the end user's notion of the system objects.

Architecture is more than that

- **The form of the business domain**
  - What the system *is*
  - Domain *model* (as in *MVC*)
  - What the programmer cares about
  - Deliver as abstract base classes
  - Eric Evans' *Domain-Driven Design*, *Multi-Paradigm Design for C++*

- **The form of the system interactions**
  - What the system *does*
  - *Role* models: *OORAM*
  - What the end user cares about
  - Has long eluded the OO crowd
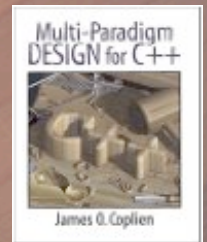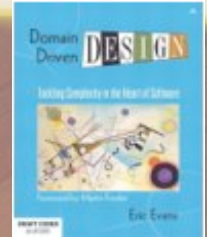
However, there is a half that is missing. Even in his early work Trygve started using roles. Roles were collections of responsibilities on objects that corresponded to the end user's view of those objects. An object could fill many roles; any given role could be filled by a number of different objects. Note that roles are themselves one kind of form. These roles captured a bit of what-the-system-does, rather than just what it is. This is the model that the end user cares about. Except for Trygve's roles, this notion of object-oriented modeling escaped the computing community for years, and in particular the programming language folks.

In the mean time, object orientation inspired further work on the what-the-system-is part of form. Model-View-Controller-User was a way to bring the what-the-system-is picture out to the user. It provided the programmer tools for organizing and coordinating the interaction between the graphical interface and the objects inside the program, though it did nothing to enforce that the objects should reflect the end-user mental model.

Over time, this same idea would appear in the analysis and design space (instead of the programming space) as domain engineering. Domain engineering is somewhat more general than OO in some respects and somewhat less general in others.

Multi-paradim design was another stab at domain engineering. Early forms of Agile architecture started to advocate the use of abstract base classes (form) rather than classes (structure) as the basis for architecture. The GOF book also strongly advocated programming to interfaces, rather to code. This started the slide into lean and Agile design. But the what-the-system-does part still hadn't come of age.

Back to OO: Other forms in the end-user's head

- **Users think more about the *roles* played by the objects than the objects**
  - What-the-system-does again!
  - Money transfer from a bank account: the roles are Source Account and Destination Account
  - Savings, checking, investment account objects can all take on these roles — so can your phone bill
- **The association from roles to objects, for a given use case, is also part of the end user model**

There are other forms in the user's head beyond what-the-system-is. As described on the previous slide, users care more about the responsibilities that are carried out inside the system: where the action is. Related responsibilities tend to group. Their grouping from the user perspective forms something called a role, which loosely corresponds to an interface in Java or C#. Think of a money transfer at an ATM: you want to transfer money between two accounts. Your model of the program is that there is a source account and a destination account. You don't think of "being source for a transfer" as being a property of a Savings Account, nor do you view this property as being duplicated in Checking Accounts. It may not be part of the Account hierarchy at all, but may extend to investments and to a concept called your telephone bill or gas bill to which amounts are transferred monthly. This role structure Is part of the end user mental model

But users know, in a given transaction, that these roles must bind to real objects. If I want to transfer money from my savings account to my checking account, then I bind the role of "being source for a transfer) to my savings account and "being the destination for a transfer" to my checking account. These mappings are also part of the end user mental model.

This is getting a bit more rich than your grandfather's architecture, which was just a bunch of objects!

Yet a few more forms!

- How about the algorithm?
    - The algorithm also has form in the user's head
        1. Start transaction
        2. Debit Source Account
        3. Credit Destination Account
        4. End transaction
    - In FORTRAN I could argue the correctness of program functionality; I can't do that in Java
    - Object orientation has served the programmers (the discovery process, architecture) but not the end users and customers — and not quality (Hoare)

8

There's more! The roles interact with each other to carry out Use Cases. In the code, Use Case scenarios are algorithms. Consider a Use Case for the money transfer; it might look as above. Where do I find that Use Case in the code?

Most OO architectures fragment the algorithm across many objects. It's easy to find the objects, and a programmer can eventually track down the bits of the algorithm, but it's difficult to see the whole. The end user, on the other hand, continues to see this algorithm, this Use Case scenario, as a whole. There is a disconnect between the end user mental model and the structure of the software. This is not in concert with the original goals of OO. In general, software with this kind of disconnect leads to poor human interfaces; Brenda Laurel writes much about this in her book "Computers as Theatre," and it follows by intuition from the direct manipulation interface metaphor. It can also confuse architecture in the long term, particularly when feature evolution (Use Case scenarios, algorithms) are important.

In the old days of FORTRAN and Pascal, I could convert a Use Case scenario into code, give it to my office mate with the scenario description, and ask him or her to desk check it. Today, after the polymorphism and object orientation of 1967, I can't do that. Our ability to reason about systems has been lost. We've tried to compensate with testing, but as Tony Hoare said, "There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies." The deficiencies of today's software are opaque.

9
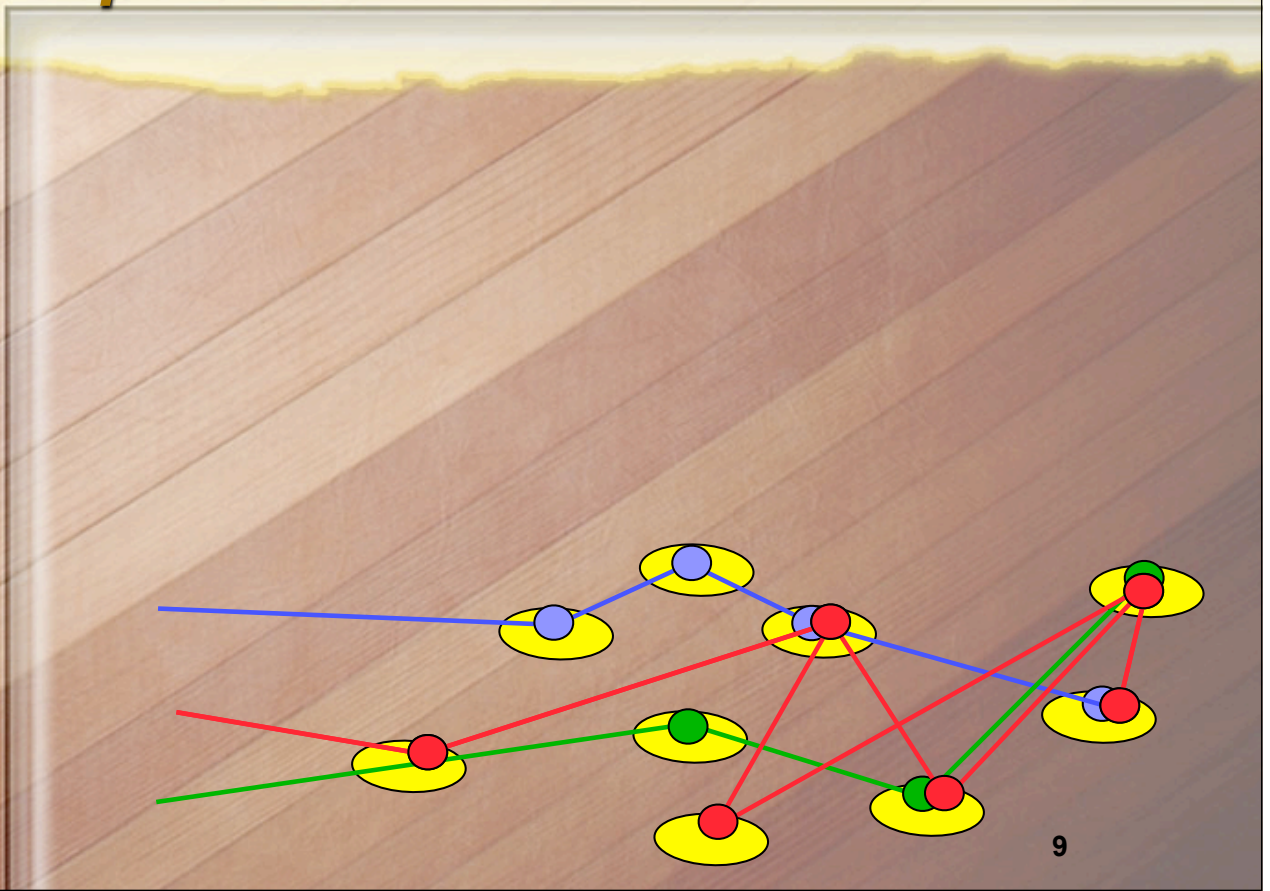
Computers can:
> store and retrieve data
> + transform data
> + ***communicate!!!***

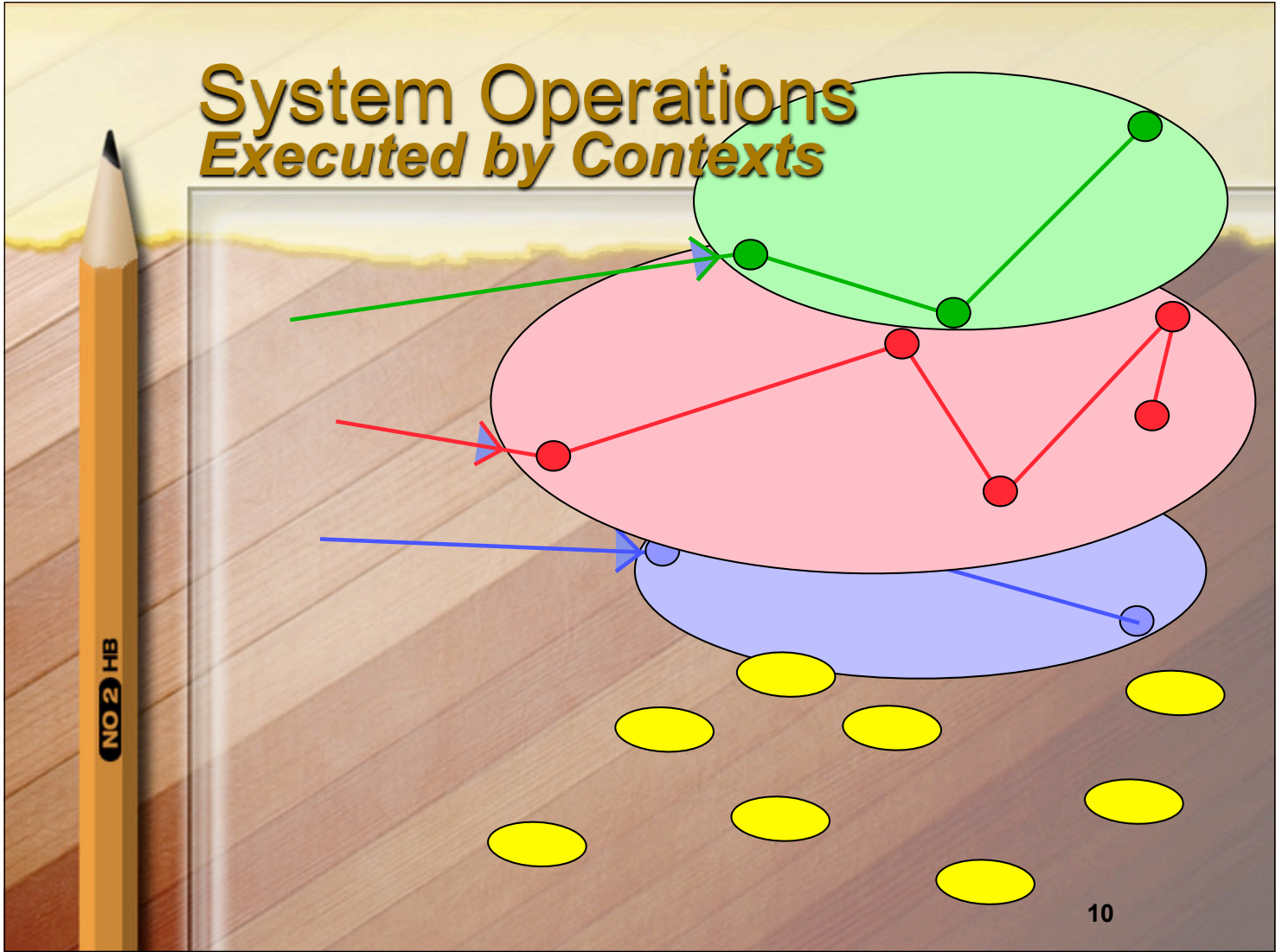Communication becomes first class citizen in computing

Show execution of the three tasks (on clicks)

Class oriented programming: NOODLES

DCI: Separation of concern:
> Each task coded separately (animated on clicks)

System Operations
Executed by Contexts

A Context (an instance of a context class)
  receives a message
  is reponsible for a use case/task/system operation
  triggers a method in the first role
  execution continues as specified in role methods

  functional decomposition context responsibility/role responsibilities

50 years ago:
  Data store / applications / functional decomposition
  + Red circles: access routines

DCI:
  Data objects / contexts / methodful roles
  + Context binds roles to objects:
    mobilize the objects that actually do the work

NOTE
  Binding role/object not shown,
 late (runtime) binding

A link to patterns?

And finally, of course, I want to paint a picture which allows me to understand the patterns of events which keep on happening in the thing whose structure I seek. In other words, I hope to find a picture, or a structure, which will, in some rather obvious and simple sense, account for the outward properties, for the pattern of events of the thing which I am studying.

— Christopher Alexander, The Timeless Way of Building, Chapter 5, 1979

11

These forms beg a new architecture

Methodful Roles

Use Case

Classes

Identifiers and Methodless Roles
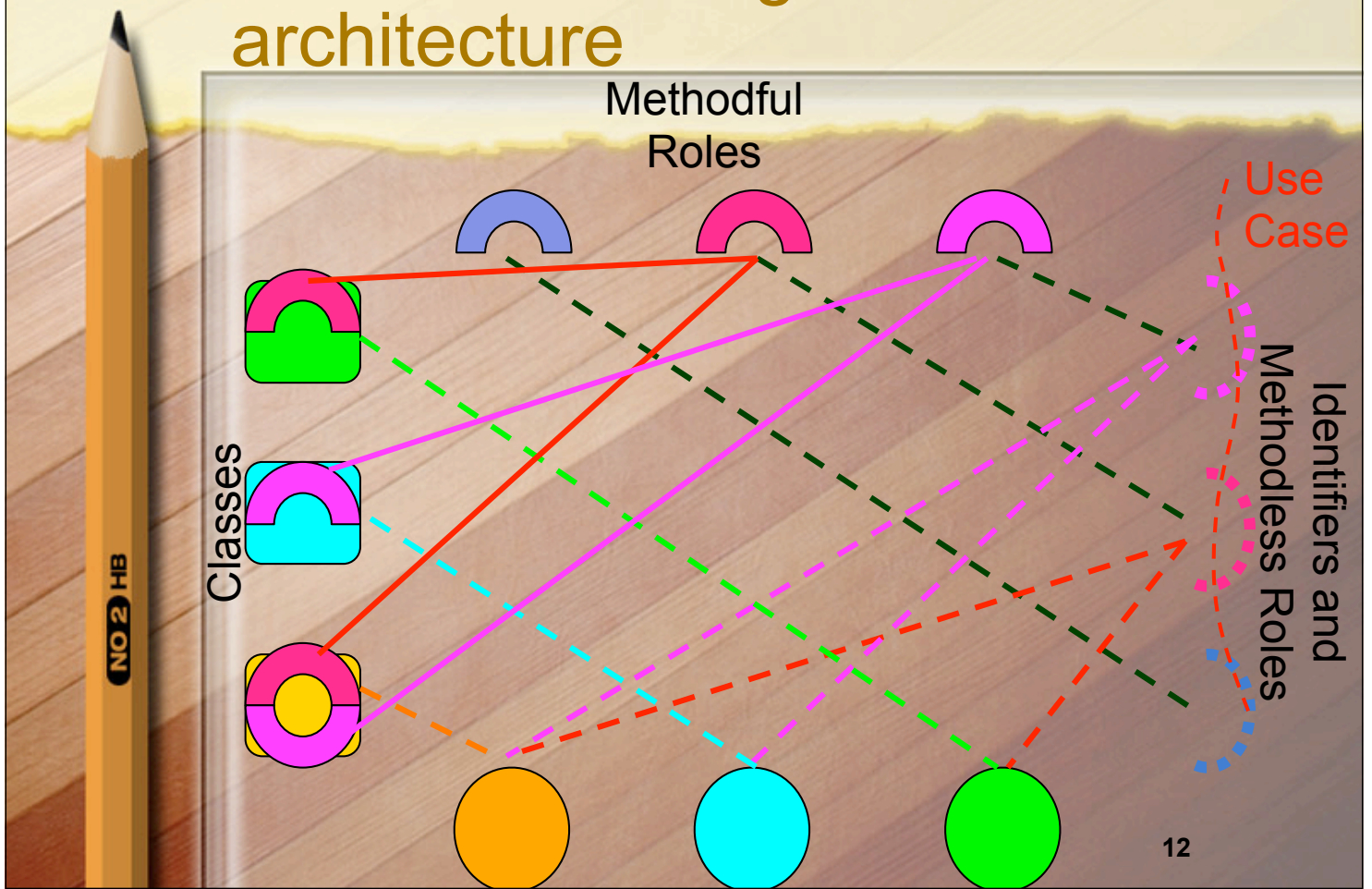
Trygve Reenskaug has come up with a new architecture to support the end user's view of Use Cases and to localize algorithm in an encapsulated role. There are four major forms in this architecture. First is the basic behavioural form, or architecture, that is defined just in terms of the protocols of the roles. Those are at the right, and we call them methodless roles. We obviously have the objects; they capture the end user's mental model of their world. Classes are an optimization for holding similar stuff that is common to many similar objects.

What Trygve adds are roles with methods. These roles are compliant with the method-less roles. Each one contains Use Case information in the form of methods. These methods can invoke methods of self or of another role. These roles are implemented as classes in most programming languages, though Scala can implement them as traits, and C++ as templates. They are pure, generic logic without state and with a fuzzy notion of the type of self or this. Each method-ful role is injected into each class whose objects play that role at some time during their lifetime. Think of roles as a kind of class, and think of the domain classes (on the left) also as classes. This "injection" is a kind of gluing operation, or more like a smashing operation, where the logic of the method-ful roles is added to the logic of the classes. Each class will then behave as though the methods of the method-ful roles had been copied into it. Of course, we retain one, single closed copy of each Use Case scenario in the methodful roles.

A Use Case scenario is implemented as an interaction of the algorithms between roles. It's a kind of procedural decomposition, much as end users decompose complex tasks into subtasks. We divide up those subtasks on a per-role basis, consistent with the end user view. At run time we associate a method-less role (like an object identifier or object pointer in C++) with an object that can support the methods of that role, and then we just let the scenario run.

Tricks with Traits

- Need to compose the generic algorithms of method-ful roles with the classes whose objects play those roles
- This is a simple class composition
- Can use Traits (à la Schärli) to glue classes together
  - Extra "hidden" field in Smalltalk classes
  - Current Squeak implementation maps the method name into every class using it
  - Trivial application of templates in C++

How do we glue classes together? Schärli has pioneered a use of traits in Smalltalk (in Squeak) to change the way method lookups work, so that two classes can be made to appear as though they are one. He uses an extra field in each Smalltalk class to hold a descriptor of the classes that are injected with it. This gives the illusion of composing classes, and we can use it to compose the generic algorithms of the role classes (method-ful roles) with the domain classes. In C++, we can do the same thing with templates. In Scala, we can use the traits language feature.

Domain objects: Ruby

```
class SavingsAccount <
      AccountWithPrintableBalance
   def initialize(balance) . . . . end
   def availableBalance . . . . end
   def decreaseBalance(amount) .... end
   def increaseBalance(amount) .... end
end
```

```
class InvestmentAccount <
   AccountWithPrintableBalance
   def initialize(balance) . . . . end
   def availableBalance . . . .  end
   def increaseBalance(amount) . . . .
   end
   def decreaseBalance (amount) . . .
   end
   def dividend . . . . end
end
```

(dumb)

Here is the injection: adding the roles into the domain classes like SavingsAccount and InvestmentAccount. Of course, we also could have injected the role TransferMoneySource into either or both of these accounts, too (in practice, we probably would). The classes re relatively dumb: they are little more than smart data. All of the action -- the Use Case logic -- is in the method-ful roles. The method-ful roles call on these rather dumb data operations to make the system fly.

The Role Code: Ruby

```ruby
module TransferMoneySink
    include MoneySinkAPI, ContextAccessor

    # Object role behaviors

    def transferFrom
        self.increaseBalance amount
        self.updateLog 'Transfer in', Time.now,
                        context.amount
    end
 end

module MoneySourceAPI
    def transfer_out; end
    def pay_bills; end
    def destination_account
        context.destination_account
    end
    def amount; context.amount end
end
```

```ruby
module TransferMoneySource
    include MoneySourceAPI, ContextAccessor

    # Object role behavior

    def transferTo
      beginTransaction
      raise "Insufficient funds" if balance <
                                amount
      self.withdraw amount
      source_account.deposit amount
      self.updateLog "Transfer Out", Time.now,
                                amount
      source_account.updateLog
          "Transfer In", Time.now,
          amount
      gui.displayScreen
          SUCCESS_DEPOSIT_SCREEN
      endTransaction
    end
end
```

15

Here is the code. We have roles (method-ful roles) for a money sink and a money source, each one of which represents a role in a money transfer Use Case. We can read the transfer algorithm from the TransferMoneySource perspective. It is readable. It is testable. One could stub it off and formally test it using system tests derived directly from Use Case requirements; if we did that, we might call it Behavior-Driven Development.

(The code has been tested, and compiles and runs.)

Injection

```
. . . .
sourceAccountForTransfer = SavingsAccount.new(account)
sourceAccountForTransfer.extend TransferMoneySource
context.setAmount amountToBeTransferred
sourceAccountForTransfer.transferTo

. . . .
```

Here is the injection: adding the roles into the domain classes like SavingsAccount and InvestmentAccount. Of course, we also could have injected the role TransferMoneySource into either or both of these accounts, too (in practice, we probably would). The classes re relatively dumb: they are little more than smart data. All of the action -- the Use Case logic -- is in the method-ful roles. The method-ful roles call on these rather dumb data operations to make the system fly.

The Code: C++

```cpp
template <class ConcreteDerived>
class TransferMoneySink: public MoneySink
{
public:
    void transferFrom(double amount, MoneySource *src) {
        deposit(amount);
        updateLog("Transfer in", DateTime(), amount);
    }
};




#define SELF static_cast<const ConcreteDerived*>(this)

#define RECIPIENT \
    ((static_cast<TransferMoneyContext*>(this)      \
                    )->destinationAccount())


template <class ConcreteDerived>
class TransferMoneySource: public MoneySource
{
public:
```

```cpp
// Role behaviors
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();
}
public:
    TransferMoneySource(void)  { }
};
```

Here is the code. We have roles (method-ful roles) for a money sink and a money source, each one of which represents a role in a money transfer Use Case. We can read the transfer algorithm from the TransferMoneySource perspective. It is readable. It is testable. One could stub it off and formally test it using system tests derived directly from Use Case requirements; if we did that, we might call it Behavior-Driven Development.

(The code has been tested, and compiles and runs.)

Injecting the roles into classes

```
class SavingsAccount:
    public Account,
    public TransferMoneySink<
                SavingsAccount> {
public:
    Currency availableBalance(void);
    void decreaseBalance(Currency);
    void increaseBalance(Currency);
    void updateLog(
            string, DateTime, Currency);
    Currency interest(void) const;
};
```

```
class InvestmentAccount:
    public Account,
    public TransferMoneySource<
                InvestmentAccount> {
public:
    Currency availableBalance(void);
    void decreaseBalance(Currency);
    void increaseBalance(Currency);
    void updateLog(
            string, DateTime, Currency);
    Currency dividend(void) const;
};
```

(dumb)

Here is the injection: adding the roles into the domain classes like SavingsAccount and InvestmentAccount. Of course, we also could have injected the role TransferMoneySource into either or both of these accounts, too (in practice, we probably would). The classes re relatively dumb: they are little more than smart data. All of the action -- the Use Case logic -- is in the method-ful roles. The method-ful roles call on these rather dumb data operations to make the system fly.

What do I get?

- Polymorphism is gone
- All objects that play the same role process the same messages with the same methods
- Algorithms read like algorithms rather than fragments
- Rapidly evolving functionality is separated from stable domain logic
- Can reason about system state and behavior, not just object state and behavior

What do I get with this?

In general, you can make virtual functions disappear at the level of the Use Case scenarios. When reading the interactions between the roles there is no dynamic dispatch: what you see is what you get. You can read the algorithm as though it were a FORTRAN algorithm. The algorithms read like algorithms; you don't have to track down their fragments all over the system.

Furthermore, the more rapidly evolving business logic (in the method-ful roles) is separated from the more slowly moving domain logic (in the domain classes). This gives good coupling and cohesion that supports change.

We have also embraced the end user and his or her mental model, and captured it in the program. That is likely to enhance usability. It also makes it easier to review functionality with end users more directly, or at least through a more direct comparison with Use Cases.

Last, we have risen to the system level. This paradigm -- called the DCI paradigm -- is about system state and behavior. Simple OO is about simple object or class state or behavior. That's inadquate to reason about system properties or even about what-the-*system*-does.

Or, from an Agile perspective:

- Allows me to connect with the user mental model
  - Users & interactions instead of processes and tools
- Can employ shared customer vocabulary
  - Customer collaboration, not contracts
- Can reason about form of task sequencing
  - More likely to deliver working software
- Exposes the changing part for ready update
  - Embracing change

Because it allows me to connect with the end user mental model, it supports the user/interaction Agile objective. Because it builds on shared customer vocabulary of Use Case scenarios, it allows me to work better in terms of customer collaboration. Because I can reason about task sequencing from the code, I am more likely to deliver working software. And because it exposes the changing part for ready update it is all about embracing change.

Learn more at:

- Baby IDE:
  - http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html
- Agile Architecture, the book: manuscript:
  - http://sites.google.com/a/gertrudandcope.com/info/Publications/LeanArchitecture.pdf
- Two Grumpy Old Men:
  - ROOTS 2008

Lasse Koskela

- cope@gertrudandcope.com