DesignScript @ Domain Specific Modelling 2016
Robert Aish, Bartlett/UCL and Emmanuel Mendoza, ARM

DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation.

In this presentation we are focussing on the application domain, the challenges this presents and how DesignScript address these challenges. This is based on our paper

http://www.dsmforum.org/events/DSM16/Papers/Aish_Mendoza.pdf

A discussion of the design decisions behind DesignScript and how it is implemented will be presented tomorrow at DSLDI 2016
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

We are addressing the domain of architecture. We can describe this domain by two different types of buildings and how computer based applications are used in their design.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

In fact we can describe these differences using three characteristic dimensions:

1. Size and complexity
2. Domain Specific to Abstract
3. The level of computational skill required
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

…but what we are really interested in is the gap in the middle…
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

... and the possible paths a novice user might take to use more advanced computation to design more interesting architecture
Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

The challenge is to create programing tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems. Hence the ‘challenge’.
The challenge is to create programming tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems.

Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

and here is an example
The challenge is to create programming tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems.

Marina Bay Bridge, Singapore
http://2.bp.blogspot.com/-TkamurQBnYA/VO0pTt1Y23I/AAAAAAAAAOE/Zety_u2abkU/s1600/helix%2Bbridge.jpg
GenerativeComponents: but not an isolated example
https://communities.bentley.com/products/products_generativecomponents/w/generative_components_community_wiki
and another example...

Velodrome at the 2012 London Olympics

http://www.designboom.com/cms/images/vido/vel01.jpg
Rhino Grasshopper
http://www.grasshopper3d.com/
or examples from Rhino Grasshopper
Such as this from the GoldSmiths’ company... not a building but it could be.
or examples from Dynamo
but here is the problem..

Visual data flow programming is a technique which is initially easy to learn and use but does not necessarily scale. It is not that the application fails, but rather as the program becomes more complex it becomes less clear and less useful.
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems.

Increasing skills required
Increasing complexity of result

So this is the argument.
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems.

2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow.

Increasing skills required

Increasing complexity of result

So this is the argument.
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems.

2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow.

3. The skill level required for programming in conventional imperative high level languages is increasing.

So this is the argument.
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems.

2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow.

3. The skill level required for programming in conventional imperative high level languages increases.

4. The increase in skills required to move from data flow programming to regular high level languages may present ‘abstraction barriers’ and be beyond the range of novice programmers.

So this is the argument.
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems.

2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow.

3. The skill level required for programming in conventional imperative high level languages.

4. The increase in skills required to move from data flow programming to regular high level languages may present ‘abstraction barriers’ and be beyond the range of novice programmers.

5. Encapsulation, via user functions [Figure 7]

6. C# classes added using ‘zero touch’ [Figure 4]

**DesignScript** is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation.

... and this is a possible solution
DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation. DesignScript implements a series of intermediate programming techniques between visual data flow programming and regular text based programming. This provides an abstraction gradient which allows the gradual introduction of more advanced programming concepts and notation.

... and this is a possible solution
This is an example of a simple visual data flow program for a Fairly abstract ‘proto-architectural’ geometry model.
but actually behind each node is a DesignScript statement, so we can use the ‘node-to-code’ functionality to replace the visual program with a text based data flow program.
and here is the same program written using Imperative code. Ultimately this is more flexible and expressive but requires more skill from the user.
Debugging is also considered. With a pure data flow program, the user can trace through the graph of connected nodes and inspect the values created. But with Imperative code, which can be iterative and recursive there are internal states which do not manifest themselves external to the nodes. This is where a conventional IDE is important to allow the user to inspect the internal behaviour of the code.
1. Double click to create an empty code block node and type the program statement

```
d = (a+b)*c;
```

In this sequence we show how the ‘code block’ node works
1. Double click to create an empty code block node and type the program statement:

```
Code Block
\[ d = (a+b)*c; \]
```

2. Clicking outside the code block node and input and output ‘ports’ are automatically generated:

```
Code Block
\[
\begin{array}{l}
a \\
b \\
c \\
d = (a+b)*c; > \\
\end{array}
\]
```

In this sequence we show how the ‘code block’ node works.
1. Double click to create an empty code block node and type the program statement

```plaintext
d = (a+b)*c;
```

2. Clicking outside the code block node and input and output ‘ports’ are automatically generated

3. Connect up the input and output ports to provide sufficient data to run the program

This sequence show how the ‘code block’ node works
1. Double click to create an empty code block node and type the program statement.

\[ d = (a+b) \times c; \]

2. Clicking outside the code block node and input and output ‘ports’ are automatically generated.

3. Connect up the input and output ports to provide sufficient data to run the program.

But let’s focus on how a ‘code block’ node works internally.
In this sequence we focus on how a ‘code block’ nodes works internally.
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports.

In this sequence we focus on how a ‘code block’ nodes works internally.
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports.

2. The expression is evaluated according to the precedence of the operators and brackets.

In this sequence we focus on how a ‘code block’ nodes works internally.
1. The independent variables \((a, b, c)\) in the expression are assigned their respective values from the ‘input’ ports.

2. The expression is evaluated according to the precedence of the operators and brackets.

3. The value of the expression is assigned [right to left] to the dependent variable \((d)\).

In this sequence we focus on how a ‘code block’ nodes works internally.
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports.

2. The expression is evaluated according to the precedence of the operators and brackets.

3. The value of the expression is assigned [right to left] to the dependent variable (d).

4. The value of the dependent variable (d) is made available to the output port.

In this sequence we focus on how a ‘code block’ nodes works internally.
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports.

2. The expression is evaluated according to the precedence of the operators and brackets.

3. The value of the expression is assigned [right to left] to the dependent variable (d).

4. The value of the dependent variable (d) is made available to the output port.

So in a code block node we are fitting a conventional program statement that operates right to left into the data flow convention that operates left to right.
This sequence focuses on the way collections are created and controlled. First we start with visual programming.
We use a ‘Range’ node to create a sequence of coordinate values. This sequence focuses on the way collections are created and controlled. First we start with visual programming.
We use a ‘Range’ node to create a sequence of coordinate values...which feeds into the Point.ByCoordinates node.

This sequence focuses on the way collections are created and controlled. First we start with visual programming...
We use a ‘Range’ node to create a sequence of coordinate values which feeds into the Point.ByCoordinates node. This sequence focuses on the way collections are created and controlled. First we start with visual programming. ...which creates the 2D array of Points. ...which feeds into the Point.ByCoordinates node.
We are going to show how the same 2D array of points can be create with the text based data flow language, but first let’s look at how a single point is create. Here we use single values for the XYZ coordinates.
We are going to show how the same 2D array of points can be create with the text based data flow language, but first let’s look at how a single point is create. Here we use single values for the XYZ coordinates...
We can switch the single X coordinate to a range expression which creates a 1D array of coordinate values. This is called replication. So anywhere where a single value is expected the user can present an array of values. Think of this as ‘map’ function which is built into the language.

...which ‘automatically’ creates a 1D array of Points. This is called replication. So anywhere where a single value is expected the user can present an array of values. Think of this as ‘map’ function which is built into the language.
We can switch the single X coordinate to a range expression which creates a 1D array of coordinate values. This is called replication. So anywhere where a single value is expected the user can present an array of values. Think of this as ‘map’ function which is built into the language.

Internally, the `Point.ByCoordinates` method is called once for every X coordinate value, and the resulting points together as the returned collection and assigned to the ‘point1’ variable.
If both the X and Y coordinate are a 1D array of coordinate values… then…

...we get a 1D array of Points, but ‘zipped’ so i’th X coordinate is matched to i’th Y coordinate.
We now introduce the concept of a replication guide. This the `<1>` in this example or more generally `<n>` where `n` controls how the different sets of inputs are combined to create the ‘cartesian product’ set of the resulting output.
We now introduce the concept of a replication guide. This the \(<1>\) in this example or more generally \(<n>\) where \(n\) controls how the different sets of inputs are combined to create the ‘cartesian product’ set of the resulting output.

The replication guides \(<n>\) are the only new syntax [and semantics] which DesignScript introduces. Indeed one of the fundamental rules of language design is only very sparingly introduce new syntax and only where there is an overwhelming reason. Where possible re-use well established syntax.
In this example, the 1D array of X coordinates has \(<1>\) replication guide. This means that DesignScript will [first] iterate over the array of X coordinates and then for every value of X it will [second] iterate over the array of Y coordinates.
In this example, the 1D array of X coordinates has <1> replication guide. This means that DesignScript will [first] iterate over the array of X coordinates and then for every value of X it will [second] iterate over the array of Y coordinates. So the replication guides control the order in which the input collections will be used to build the output collection and the dimension of that output collection. In this case we get a 2D array of points.
We now draw curves through the points. Each curve requires a 1D array of points. We are offering a 2D array of points.....therefore we end up with a 1D array of curves.
Switching the replication guides so that now Y coordinates are <1> and X coordinates are <2> results in the 2D array of points being built with Y as the 1st dimension.

....therefore we end up with a 1D array of curves built in the opposing sense.
It is of course possible to build exactly the same example using Imperative programming. So what conclusion can we draw?
The visual data flow language is highly simplified as there are no explicit flow control statements [because flow control is provided by the dependency graph]. New syntax is added to control replication [allowing the use of collections without explicit iteration]. The text-based data flow language is succinct but has defined limitations.
The imperative language gives most flexibility, expressibility, but is less succinct and requires more programming skills.
Building are composed of collections of components. This example demonstrates the value of being able to directly operate on collections.
Using the hybrid visual and text based programing to model the MIPS microprocessor pipeline.
overview of the design and implementation

Please see the companion presentation that we gave at DSLDI 2016
In DesignScript, we have implemented visual and text based data flow programming and text based imperative programming, but not visual imperative programming... This cell is empty.
There have been attempts to retrofit aspects of imperative programming into visual data flow systems, but again we may be falling into the trap of making visual programming too complex and unreadable. The effort to explain how this work might be better spent teaching a regular text based imperative language.
Other system use the ‘jig-saw’ puzzle visual approach for imperative programming.
Our target users are not school students, but professionals who happen to be novice end-user programmers. Our sense, is that by the time the users have progressed from ‘node to code’ they will not want to go back to a restricted ‘jig-saw’ puzzle approach.
Of all the multi-paradigm languages listed on [https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages](https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages) only Oz, Higher Order Perl and Scala [via Akka] support data flow, imperative and object oriented programming and none support data flow, imperative object oriented and visual programming.

We briefly compared DesignScript with other multi-paradigm languages.

There are important issues concerning the evaluation of usability of these systems. Please see: Robert Aish and Sean Hanna (2017) "Comparative evaluation of parametric design systems for teaching design computation", paper submitted to the forthcoming special issue on Parametric Design System, Design Studies.
Of all the multi-paradigm languages listed on https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

only Oz, Higher Order Perl and Scala [via Akka] support data flow, imperative and object oriented programming

and none support data flow, imperative object oriented and visual programming.

There are important issues concerning the evaluation of usability of these systems. Please see:

More generally, what is the take-up within professional architectural practices of the domain specific programming tools? Of the ‘high-end’ practices, we have reports that while most architects are still using standard design and modelling applications [which don’t require any programming expertise], a small but increasing number are using visual data flow programming, supported by smaller proportion using scripting and general purpose programming tools.....
Often a team of architects within a practice will have a few members with visual data flow programming experience who in turn will be supported by more experienced programmers developing special functions to handle complex requirements. These functions will be used by the data flow programmers as specialist or custom nodes within the visual programming environment. This suggests that tools which offer a range of programming techniques [harnessing different levels of skills] can become effective collaboration platforms for teams of users with different skills.
We might observe that the number of users with different levels of programming expertise is inversely proportional to the expertise required...
In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.
In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

We conclude with some ‘take home’ messages.....
In this application domain, architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real-world projects. To combine exploration and complexity, users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis in ‘Epigrams on Programming’.

This suggests a positive motivation for an end-user to learn programming.
In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.” Green and Blackwell.

One the other hand there are challenges
In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.” Green and Blackwell.

A domain specific application should not avoid abstractions, but avoid abstractions becoming a barrier. The same abstraction may sometimes be a barrier; at other times the key idea that ‘changes the way you think’.

A successful domain specific language has to weave this incredibly difficult path between supporting abstractions but not forcing their use... We have to wait for the users to recognise that there might be a ‘better way’ and then have that ‘better way’ waiting in the wings.
In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration. Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.”

Green and Blackwell.

A domain specific application should not to avoid abstractions, but avoid abstractions becoming a barrier. The same abstraction may sometimes be a barrier; at other times the key idea that ‘changes the way you think’.

Finally

A domain specific computing system will only be successful if it is more than domain specific and introduces the user to more general purpose computing ideas and their application.
Visit www.designscript.io

Questions