THE UNIVERSITY *of* EDINBURGH
informatics

Composition and Inheritance in Declarative Configuration Languages

Paul Anderson <dcspaul@ed.ac.uk>

http://homepages.inf.ed.ac.uk/dcspaul

Introduction

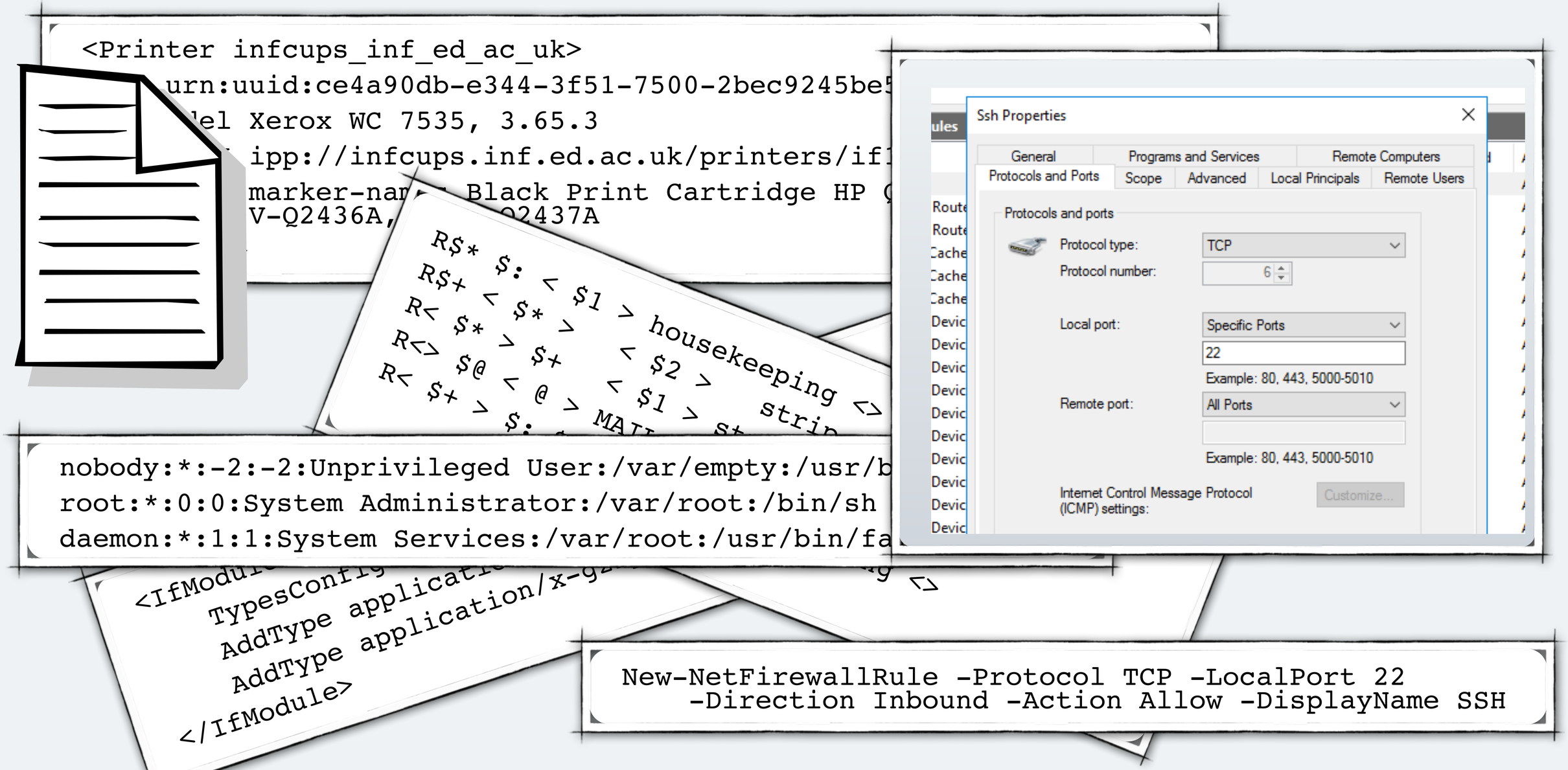
Motivation

- Powershell/DSC & Azure
- A more formal (language) approach to configuration

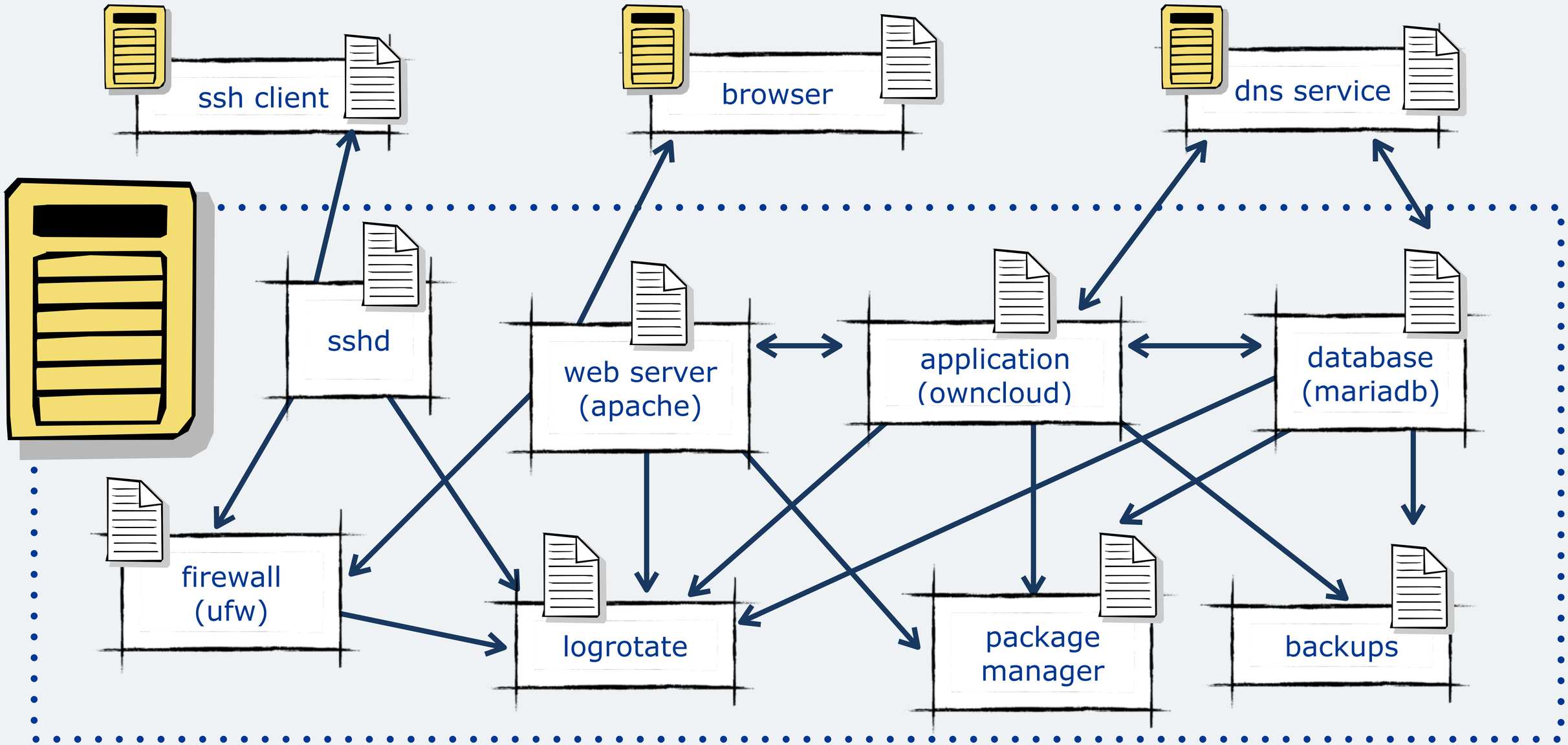
Overview

- Slightly broader than the title might suggest …
- Configuration background & a bit of history
- A more detailed example of something I’ve been thinking about recently

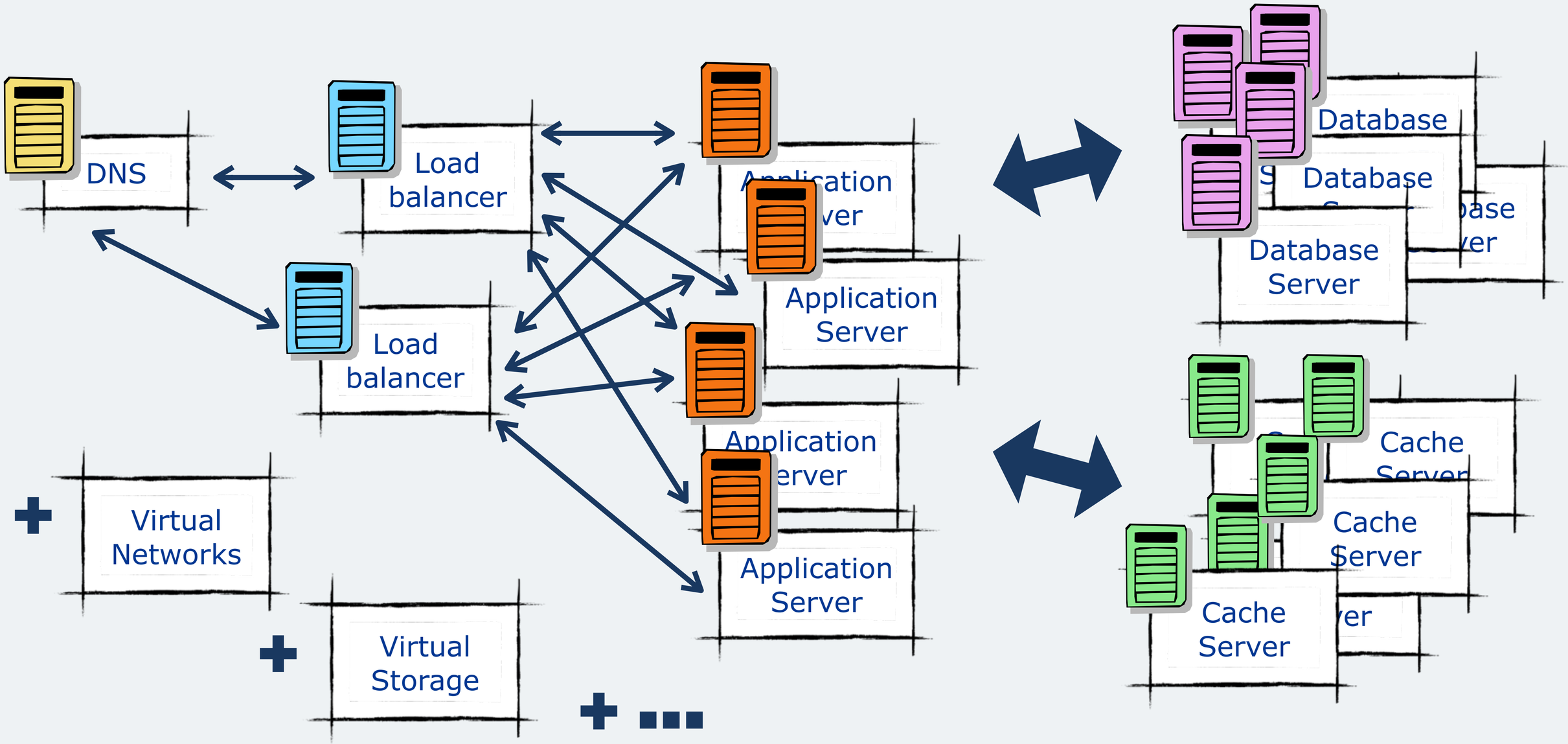
Configuring a “subsystem”



Configuring a host



Configuring a cloud application



❝ One director told me on Friday they spent 2.5 months setting up a cluster of a few thousand machines …. largely due to configuration issues.

Dependencies between them, which to tweak, and what values were relevant for the specific cluster were some issues cited. ❞

Configuration tools

A more uniform way of managing subsystem configurations

- A programmable API ? …
- And/or a common format for specifying configuration parameters ? …

Representing/enforcing relationships between subsystems (and hosts)

- The firewall permits access to the port being used by the web server application
- The client should use the same port number as the server

Sharing of common configurations (with variations)

- I want 20 web servers with “the same” configuration
- But they have to have different addresses …

Higher-level modelling and policy/reasoning

- I want a 3-tier web service the same as X
- Do not configure any machines running finance services to be accessible from the public internet

Two approaches

Imperative (Bash script)

```
if [ [ 0 ne $(getent passed elmo > /dev/null)$? ]
then
  user add elms –gid sysadmin -n
fi
GID=`getent passwd elmo |awk -F: {print $4}'`
GROUP=`getent group $GID |awk -F: {print $1}'`
if [ $GROUP != $GID ] && [ $GROUP != sysadmin ]
then
  user mod –gid $GROUP $USER
fi
```

Declarative (Puppet)

```
user { 'elmo':
  ensure => present,
  gid => 'sysadmin',
}
```

An imperative approach

Evolved from “scripting” the original manual procedures

- It is non-trivial to prove that the sequence of actions produces a final state which meets the requirements
  - the “requirements” may not even be explicit
- The workflow assumes a fixed (set of) starting state(s)
- The ordering implied by the workflow may be over-constrained

But …

- This is still popular because system administrators can use familiar procedures and imperative scripting languages

```
figure = hips
add legs under hips
add torso on top of hips
add head to top of torso
add arms to torso
add hair to head
add hands to arms
add bag to hand
```

A declarative approach (desired state)

Specifies the desired state - not the workflow

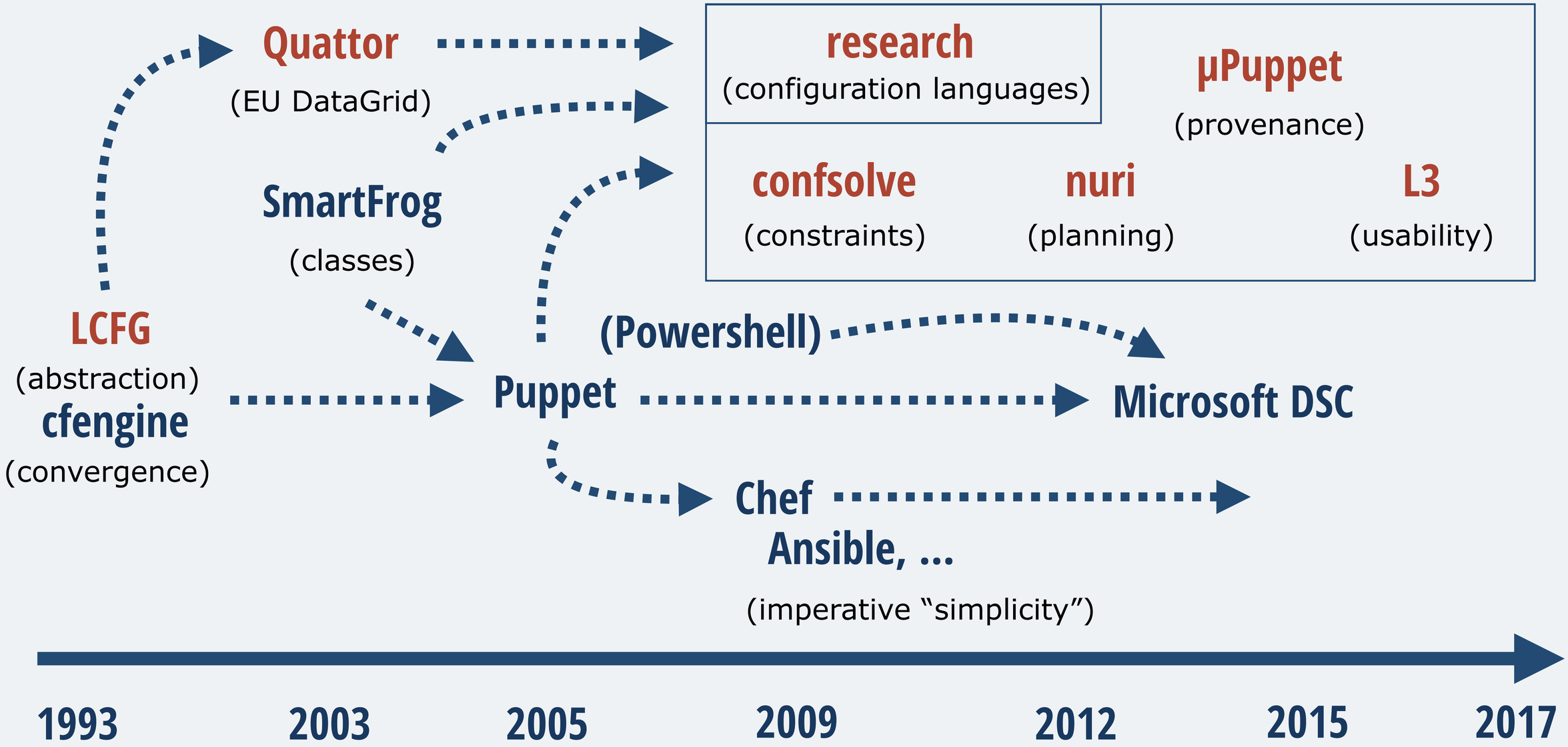
- The specification is independent of the deployment
- It is independent of the starting state
- There is an explicit specification of the “desired” state against which we can compare the “actual” state

But …

- We do need to compute a workflow between the actual and desired states to implement the deployment
- I don’t plan to talk about deployment issues, but ..
  - most production tools do not handle this well
  - we have successfully used automated planning

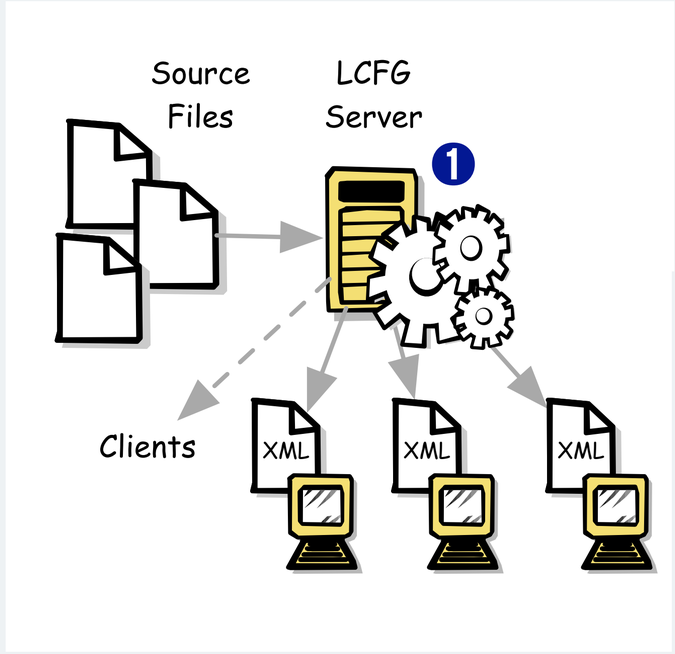
```
figure: {
 head: {
  face: "male"
  hair: {
   style: "short"
   colour: “brown"
  }
  hat: none
 }
 clothing: {
  ...
 }
}
```

Some history (of “desired state” / “declarative” configuration tools)

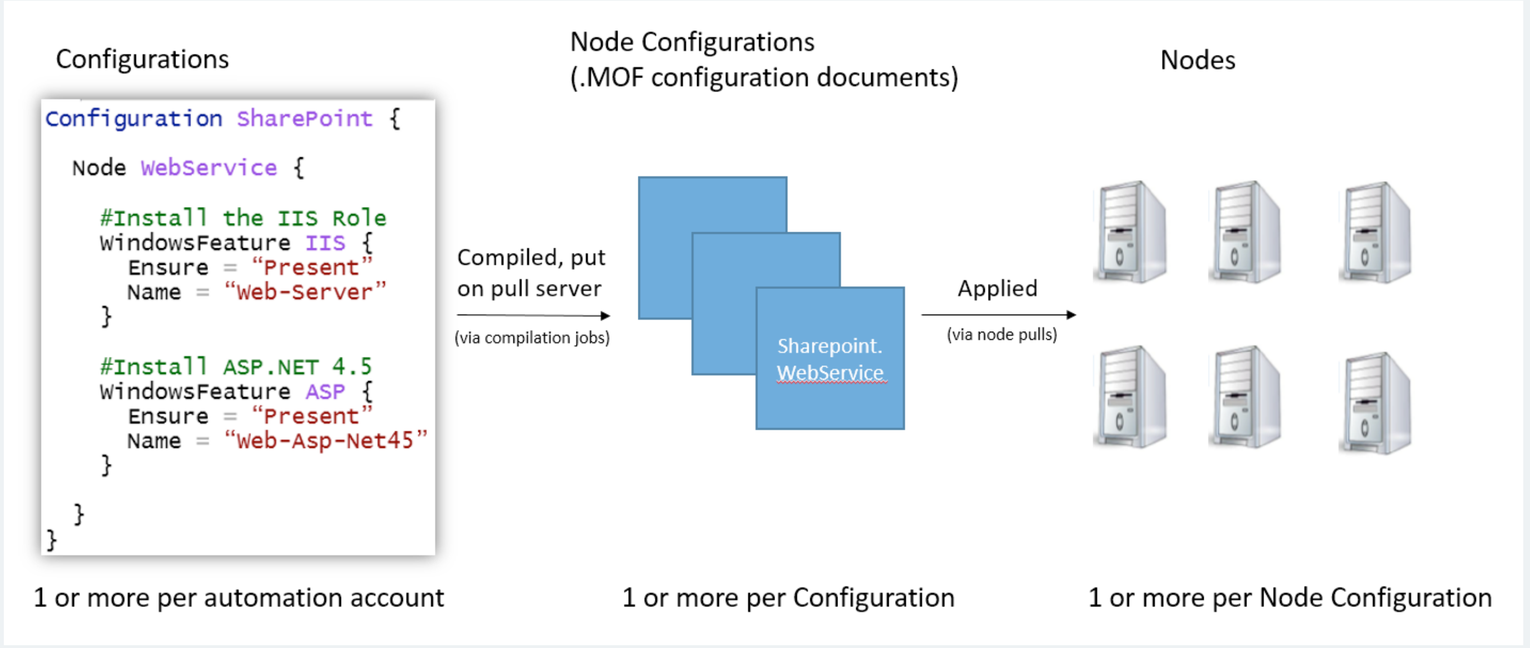


Architecture ... has not changed significantly ...

LCFG (1996)



Azure Automation DSC (2015)



https://docs.microsoft.com/en-us/azure/automation/automation-dsc-overview

But there are different approaches to the source specification language

- Imperative construction of the desired state (using, for example Powershell)
- A more declarative description of the requirements (for example, SmartFrog)

An example problem - composition & inheritance

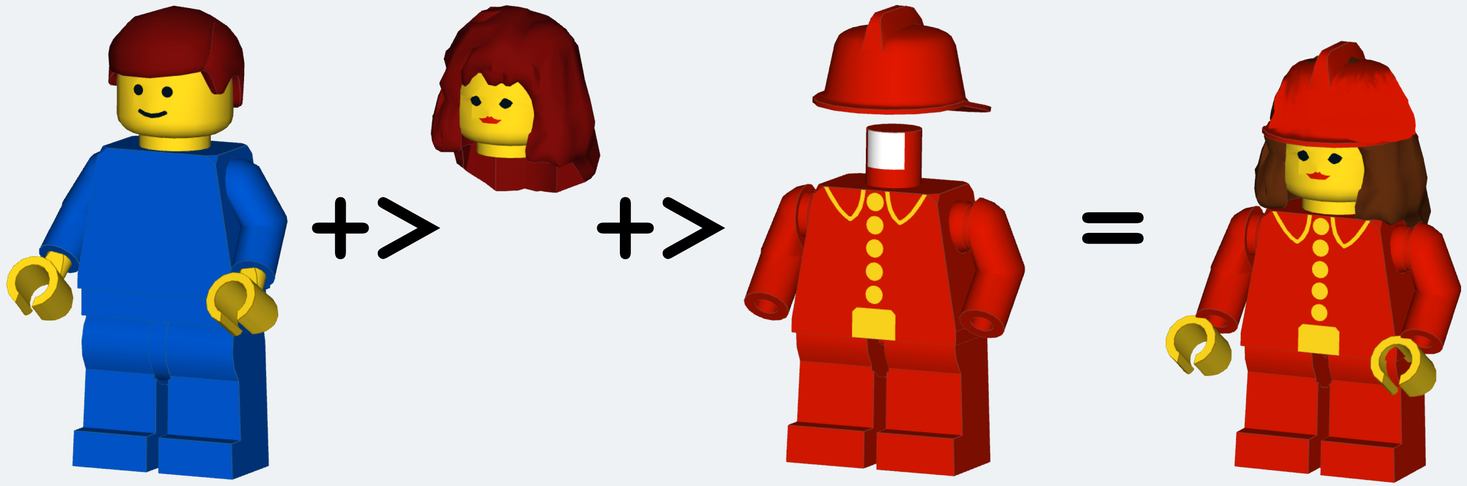
Motivation …

- Declarative configuration languages are not programming languages
  - they describe configurations, not computations
  - they are used by system administrators, not programmers
- But this involves more than (for example) JSON
  - because we want to support operations such as composition …
- We want **different people** to manage different aspects of the configuration
  - these may overlap
  - but it should be possible to author and edit them independently
  - and the resulting composition should meet everyone’s requirements

https://www.flickr.com/photos/foilman/15844421582

(Instance) inheritance +> (specialisation)

This is a typical operation ...

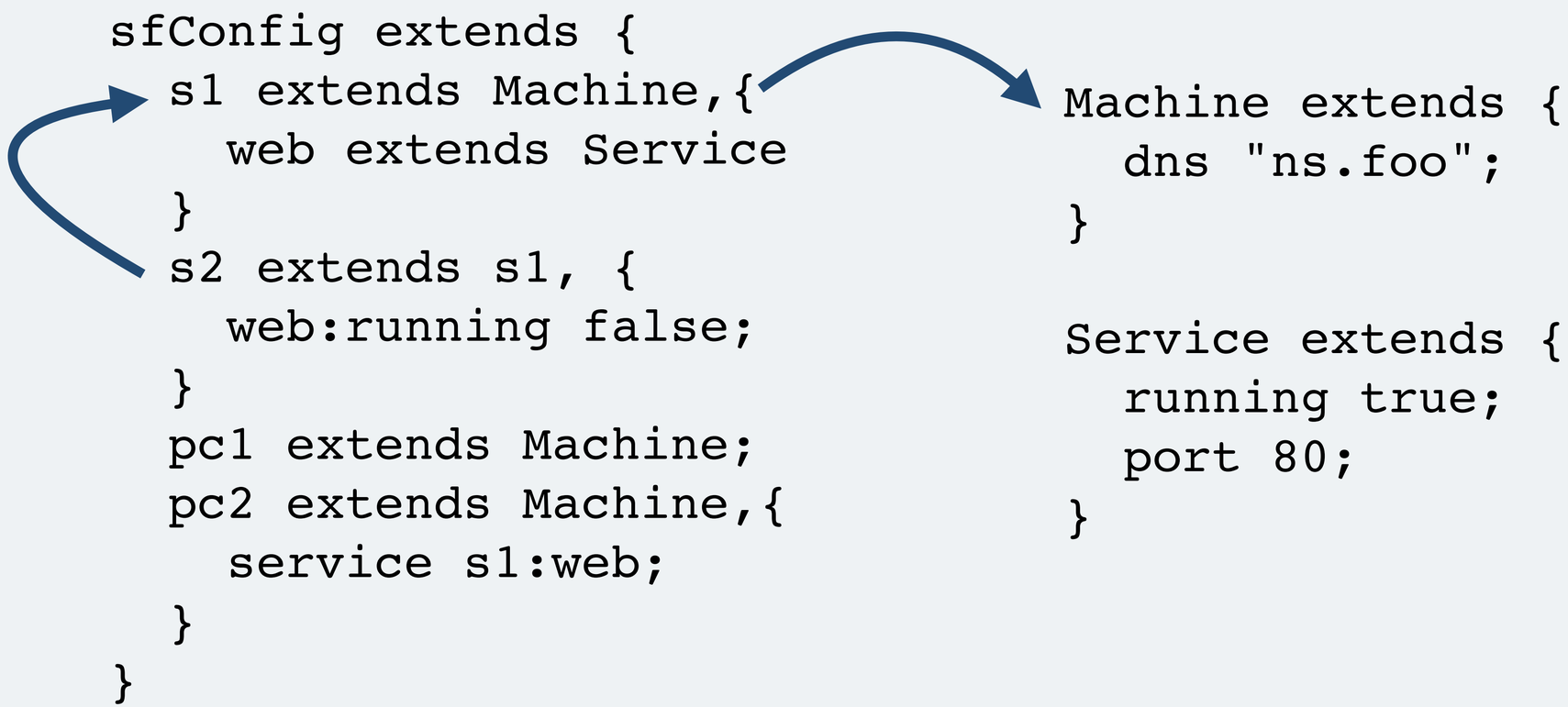


Or ...

"I want a Redhat Linux machine running Apache and Wordpress"

The ordering here is not related to any temporal deployment ordering!

An example in SmartFrog



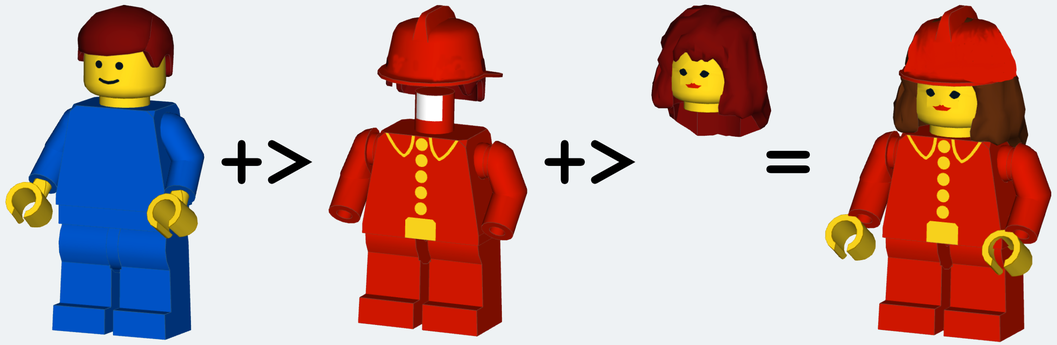
```
sfConfig extends {
  s1 extends Machine,{
    web extends Service
  }
  s2 extends s1, {
    web:running false;
  }
  pc1 extends Machine;
  pc2 extends Machine,{
    service s1:web;
  }
}
```

```
Machine extends {
  dns "ns.foo";
}

Service extends {
  running true;
  port 80;
}
```

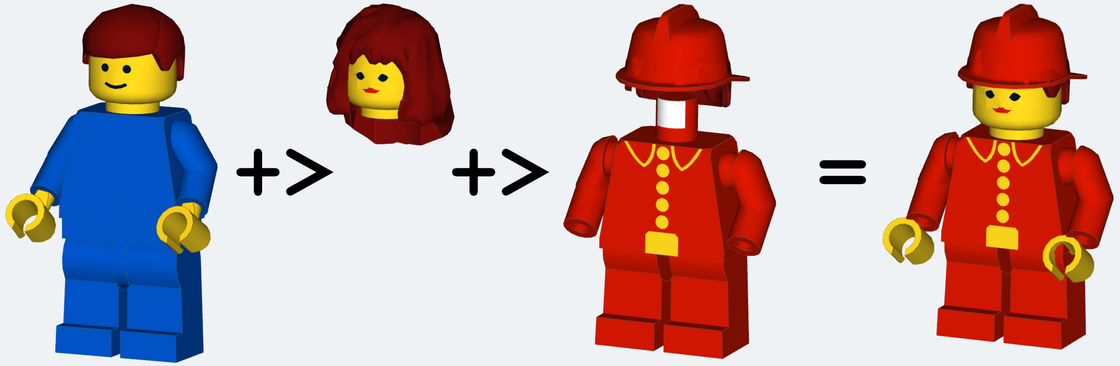
This is instance inheritance, not class/type inheritance

Conflicts

“Hair will not extend beyond the bottom of the earlobe”
(International Association of Women in Fire and Emergency Services)



A female firefighter ?



Or a firefighting female ?

Apache - “how the sections are merged”

The configuration sections are applied in a very particular order. Since this can have important effects on how configuration directives are interpreted, it is important to understand how this works. The order of merging is:

1. <Directory> (except regular expressions) and .htaccess done simultaneously
   (with .htaccess, if allowed, overriding <Directory>)
2. <DirectoryMatch> (and <Directory "~">)
3. <Files> and <FilesMatch> done simultaneously
4. <Location> and <LocationMatch> done simultaneously
5. <If>

Apart from <Directory>, each group is processed in the order that they appear in the configuration files. <Directory> (group 1 above) is processed in the order shortest directory component to longest. So for example, <Directory "/var/web/dir"> will be processed before <Directory "/var/web/dir/subdir">. If multiple <Directory> sections apply to the same directory they are processed in the configuration file order. Configurations included via the Include directive will be treated as if they were inside the including file at the location of the Include directive.

Sections inside <VirtualHost> sections are applied after the corresponding sections outside the virtual host definition. This allows virtual hosts to override the main server configuration.

When the request is served by mod_proxy, the <Proxy> container takes the place of the <Directory> container in the processing order.

https://httpd.apache.org/docs/2.4/sections.html

Active directory - “GPO inheritance”

A user or a computer in an OU can have multiple GPOs applied to it. For example, Local Group Policy, GPOs linked to the site, GPOs linked to the domain and GPOs linked to the OU. Also, multiple GPOs can be linked to any of these containers. The following is the order in which the Group Policy settings take effect.

- Local Group Policy settings are applied first
- GPOs linked at the site level are applied next followed by the GPOs linked at the domain level and OU level. Since GPOs linked to the OU are processed last, they have the highest precedence
- In case of nested OUs, GPOs linked to the parent OUs are applied first followed by the GPOs linked to the child OU
- If multiple GPOs are linked to a container, then the GPO with the lowest link order will have the highest precedence
- To view the list of GPOs applied to a container, double-click the container and select the Group Policy inheritance tab in the right pane. A list of GPOs with link order, location and status will be displayed

The final configuration of policy settings applied to a user or computer is a combination of all the policy settings defined in each GPO. In case of any conflicts, the policy settings configured for the GPO with a higher precedence override the GPO with lower precedence. However, this behavior can be altered using the block inheritance option.

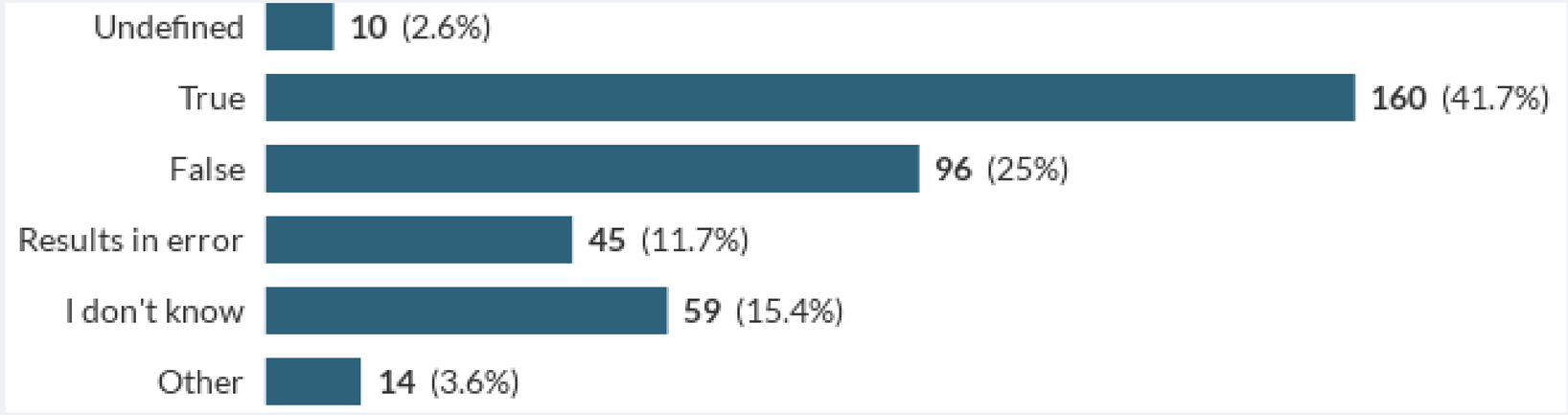
To block inheritance and apply only the policy settings configured in GPOs linked to a particular OU, right-click the OU and select Block Inheritance. This will block all the policy settings from GPOs linked at the domain level, site level and parent OUs.

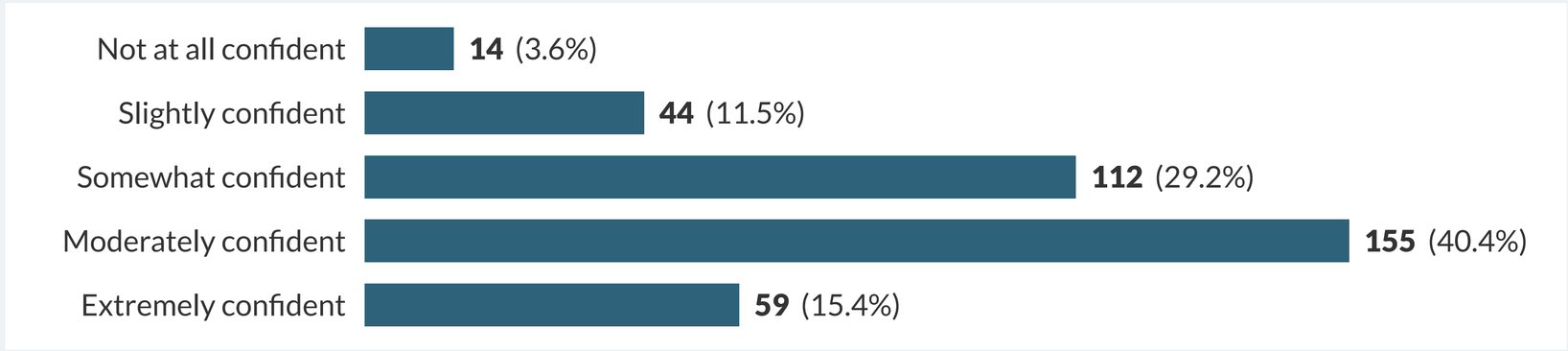
http://www.windows-active-directory.com/windows-gpo-inheritance.html

User interpretation of inheritance

What is the value of MusicBox.showbalance ?

```
Box {
  show-balance = true,
}
Player {
  tracks = 573,
  genres = 11,
  show-balance = false
}
MusicBox
  inherit Box
  inherit Player {
    genres = 9
}
```





Some comments on inheritance

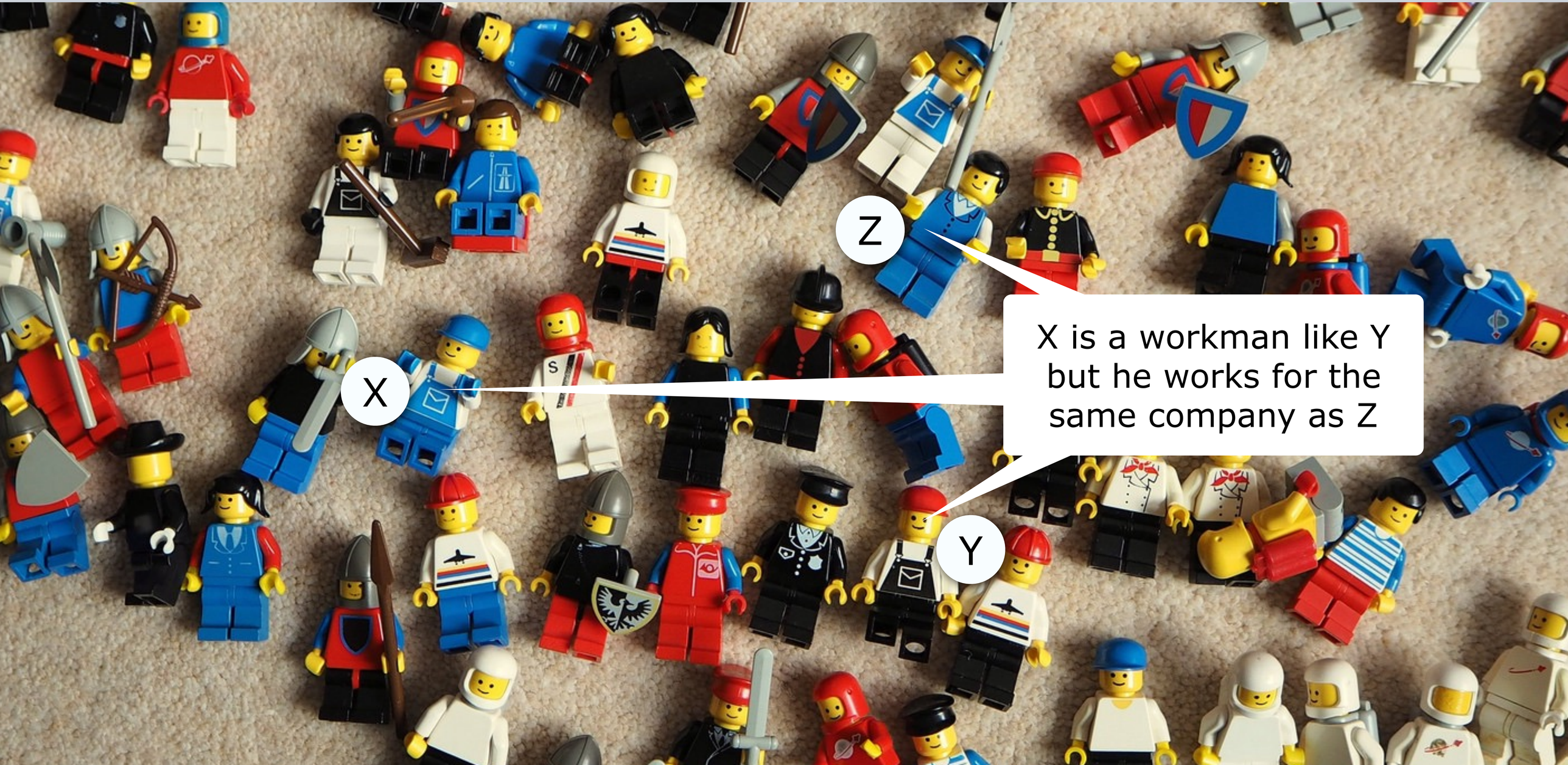
❝ Inheritance is useful when used in high level programming languages. Inheritance in a CfgMgmt System often leads to complexity and should therefore be avoided. ❞

❝ Multiple inheritance was a mistake. ❞

❝ it is strange that _instances_ can _inherit_. I would totally grok 'extends'. 'inherit' is more for classes. ❞

❝ The inheritance question was not difficult at all, probably because I have been educated and trained in object-oriented design and programming. ❞

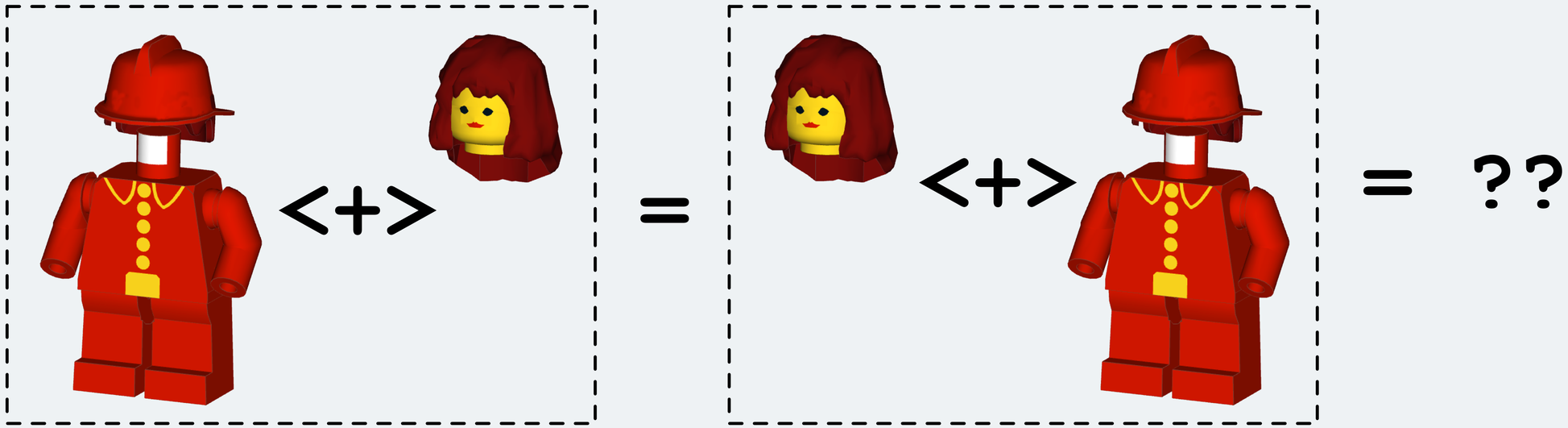
Composition



Commutative composition <+>

The “user” is forced to make a decision about inheritance order

- But they don’t usually have the information to do this
- And neither order may be correct if there are multiple conflicts!



The user needs a commutative composition operation

- And the ***authors*** of the components need to specify how they should be composed

Resolving conflicts

What do we mean when we specify a value for a resource ?

- *❝ The value really must be 42. ❞*
- *❝ I don't really care what the value is, but I can't leave it empty, so I'll put 0. ❞*
- *❝ 36 would be a good value, but I don't care if someone else would rather have something different. ❞*
- *❝ I think it should be 46, but if Jane thinks it should be different, then believe her. ❞*
- *❝ The value must be between 100-200, but I can't specify a range, so I'll say 150 ❞*

Conventional configuration languages are not rich enough to capture this

Constraints & Confsolve

ConfSolve was developed as part of a previous Microsoft-sponsored Phd

- A configuration language which supports very expressive constraints …
- "*I want 4 machines configured as database peers, which can be any machines except themselves*":

```
constraint
 forall (this in 1..4) (
  DatabaseServer_peer[this] != this
 );
```

These compose well (commutative)

- They support all of the previous examples & much more
- But, it requires a lot of thought to specify values which are not under- or over-constrained
- Understanding the consequences of these very general constraints is too difficult in most practical cases & the results can be unpredictable

L3

An experimental configuration language

- A small language with a clear, declarative semantics
- Configurable semantics for experimenting with usability
- Features specifically designed to support operations such as composition
- A balance between usability & expressiveness
- Not a programming language!
- Output in JSON-like format which can easily be converted for deployment by other tools

A work-in-progress ...

Tags & constraints in L3

In “L3”, we can tag resource values …

```
a: { colour: "red" #aliceSays }
b: { colour: "blue" #bobSays }
```

And we can specify precedence between the tags

```
c: ( $a <+> $b ) #aliceSays >> #bobSays
d: ( $a <+> $b ) #aliceSays << #bobSays
```

Two special tags …

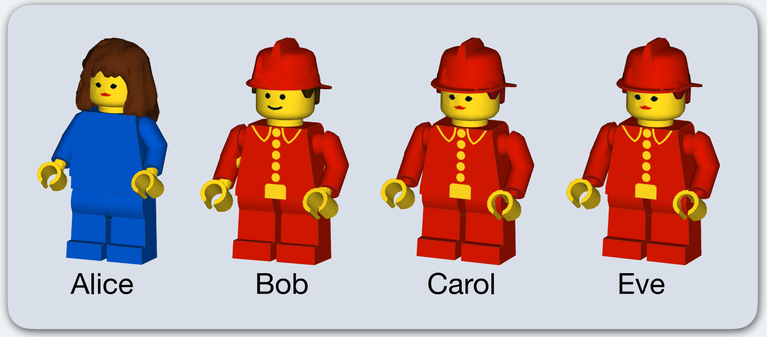
```
v #final
  v >> x ∀ x ≠ v

v #default
  v << x ∀ x ≠ v
  : ¬(x #default)
```

This supports requirements such as …

- *“I think it should be 46, but if Jane thinks it should be different, then believe her”.*
- *“Parameters specified at a departmental level should override those set at a corporate level”.*

Composition example

```
figure: {
 head: {
  face: "male"
  hair: {
   style: "short  }
 }
 clothing: {
  top: "bluetop"
  bottom: "bluebottom"
 }
} #default
```



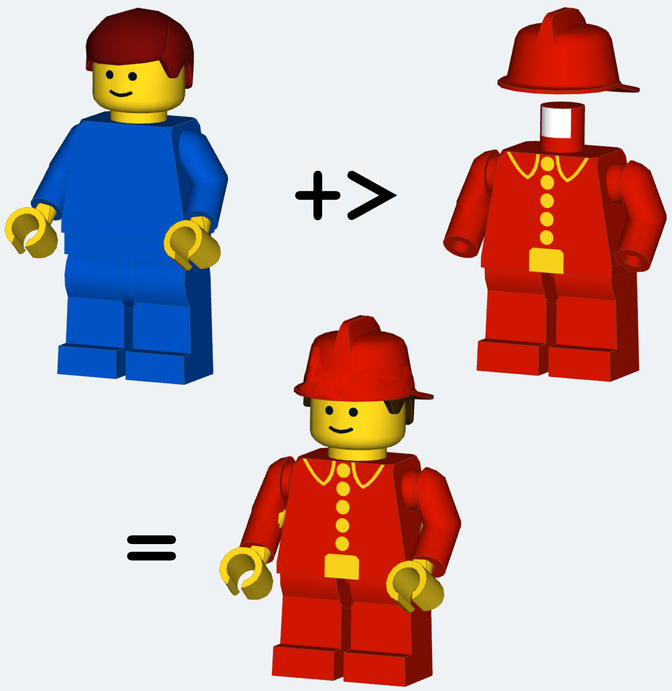

```
female: $figure <+> {
 head: {
  face: "female" #final
  hair: {
   style: "long"
  }
 }
}
```

```
fireperson: $figure <+> {
 head: {
  hair: {
   style: "short"
  }
 }
 hat: {
  style: "fireHat"
  colour: "red"
 }
 clothing: {
  top: "firetop"
  bottom: "redbottom"
 }
} #final
```

```
alice:   $female
bob:     $fireperson
carol:   $female <+> $fireperson
eve:     $fireperson <+> $female
```

Specialisation (instance inheritance)

```
fireperson:
$figure +> {
    head: {
      hat: "firehat"
    }
  }
  clothing: {
    top: "firetop"
    bottom: "redbottom"
  }
}
```



This can now be defined in terms of composition ...

```
(X +> Y) ≡ (X #tag1 <+> Y #tag2) #tag1 << #tag2
```

Usability & semantics

Some questions ...

- Are the tags and constraints confusing?
- How should they propagate?
  - is the example on the right confusing?
  - what are the alternatives?
- Are they sufficient for some “real” problems ?
- What kind of real problems can they not address ?
- How can we tell whether new approaches are genuinely bad/difficult, or just require users to adjust to an unfamiliar paradigm?

```
a: { port: 42 #alice, … }
b: { port: 37 #bob, … }
c: $a <+> $b // (42 or 37)

server: {
  …
  myport: $c.port // (42)
  …
} #alice >> #bob

client: {
  …
  myport1: $c.port // (37)
  myport2: $server.port // (err)
  …
} #bob >> #alice
```

Some concluding thoughts ...

Less complexity - not more!

- the current complexity is driving users back to “simpler” approaches
- tools and languages with fewer, more carefully considered features
- less expressivity - configuration is not programming

Clean separation of concerns

- the admin should not have to “program”
- the dns expert should not need to understand the mail configuration

Declarative specifications of higher-level intent

- not imperative construction
- declarative states with automated change planning

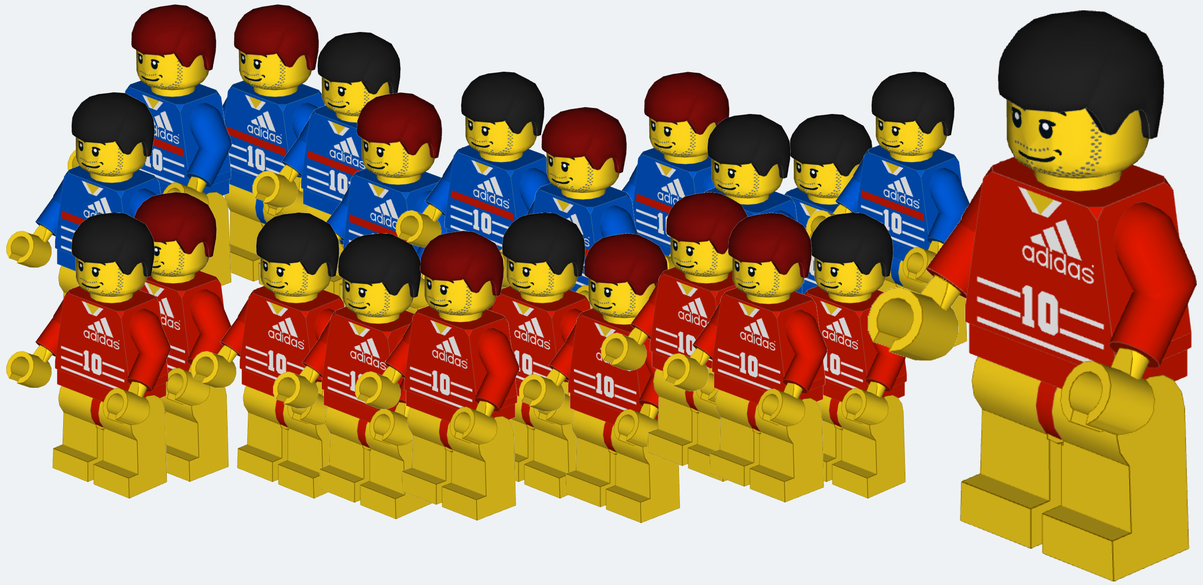
More formalisation

- to improve reliability and security
- to inform the design of clearer tools & languages

```
dave: $figure +> {
  name: "dave",
  head.hair.colour: “black"
} #final

bob: ...

team: {
  team_colour: "blue" #default
  player = $figure +> {
    clothing: {
      top: "shirt"
      colour: $team_colour
    }
  }
  leftWinger: $player <+> $dave
  centreForward: $player <+> $bob
  ...
}

awayTeam: $team +> { team_colour: "red" }
```


Paul Anderson

<dcspaul@ed.ac.uk>



http://homepages.inf.ed.ac.uk/dcspaul

(*publications, talks etc …*)