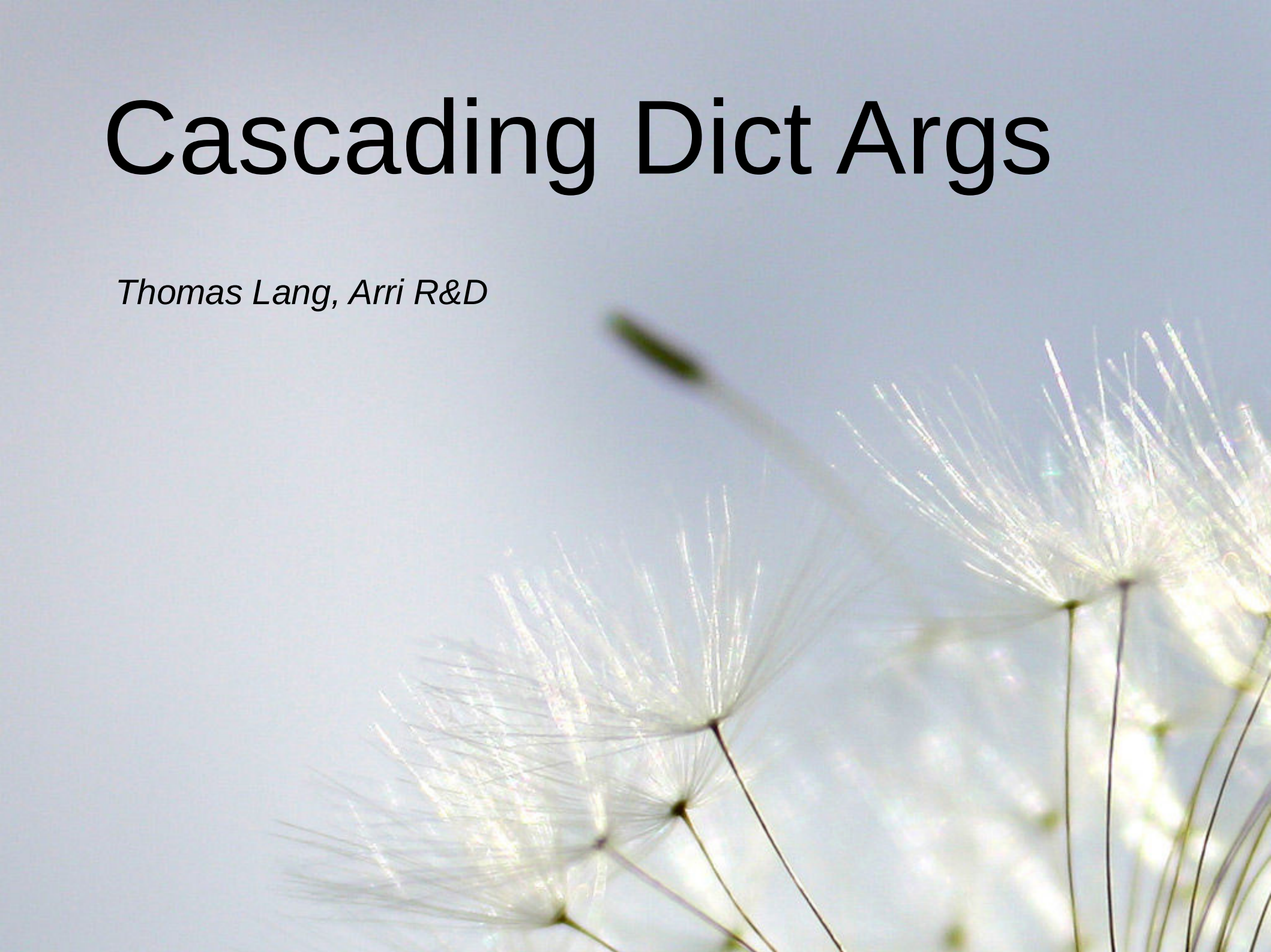
Cascading Dict Args

Thomas Lang, Arri R&D



The Objective Of This Talk

My world view is changing.

Well this is no great news, because it is constantly changing ... at a moderate rate. But recently, fueled by newer techniques that have become available in newer version of Tcl, like lambdas or coroutines, this change has gained momentum.

During the past two years or so, some technique that I'm calling “Cascading Dict Args”, gradually took over a major part of my attention – discovered slowly, rather than actively designed.

In this talk, I want to give a working introduction to dict args by walking through the evolution of a small feature that might fit into my current major project. As we go along, I will also try to show how lambdas and coroutines and stuff go hand in hand with dict args.

I will start with a simple two line loop - the kind of ad hoc throwaway code that you're constantly experiencing if you're used to work interactively in the Tcl console.

Driven by needs that I've really been confronted with while improving my application, I will then gradually add to the complexity of that code. I will even push that technique some further in these examples than I'm actually doing it in my real code – just to explicate some points.

But when we reach the end …

****** SPOILER ALARM ******

... I hope you will be surprised that the code we've ended up with will, in a way, again look nearly as simple as our initial example … but it should have become a good deal more flexible along the way.

'Dict Args' … A First Look

I've started to use dict args out of haste and laziness. I needed a simple technique to parse optional procedure parameters, and I needed it quickly. I found that Tcl dicts offered nearly everything I needed to do all my parameter processing within just two or three lines:

1. Define a dictionary containing all your parameters along with their default values.
2. Merge this dictionary with your procedure's **`args`**
3. Maybe, 'pluck' parameters from the processed dict to move them into local variables.

```
proc myProcedure args {
    set   _  [' width 100 height 100 debug no]
    set   o  [dict merge $_ $args]
    pluck o  width height

    set area [expr {$width * $height}]
    if {[o debug]} {puts “area size is $area”}

    anotherProc {*}$o area $area
}
```

There's no large library support needed to make this possible, but we can simplify dict access which tends to be a bit clumsy (→ `[o debug]`), and I like list creation to have a quoted appearance (→ `'`).

```
proc '     {    args} {return $args                                  }
proc $     {var args} {upvar  $var it; dict get $it {*}$args}
proc o     {    args} {tailcall $ o {*}$args                        }
proc pluck {var args} {upvar  $var it; foreach _ $args {
      uplevel 1 [' set $_ [$ it $_]; dict unset it $_ }    }
```

Moreeover...

```
proc myProc args {
    set   _  [' width 100 height 100 debug no]
    set   o  [dict merge $_ $args]
    pluck o  width height
                                  #  no validation!
    set area
    if {[o debug]} {puts “area size is $area”}

    anotherProc {*}$o area $area
}
```

4

5

These three yellow initial lines have started, slowly but steadily, to pop up everywhere in my code.

4 Haste had let me defer the validation of my parameters indefinitely. Of course this meant that every misspelled parameter would be lost and ineffective – but I still think that's halfway acceptable in the project I'm working on.

But this also opened up a range of new possibilities which I found myself using without giving much thought until I recognized: … oops, something had changed.

5 I started to pass my processed args freely down to other procedures. Just look at what's cascaded down to **anotherProc** in the above demo code: **width** and **height** have already been removed, I'm considering these as private to **myProc**; if **anotherProc** has parameters of the same name, they are shadowed, and **anotherProc** will have to use its defaults. **debug** is a shared parameter and passed down unmodified. **area** is appended, possibly shadowing an **area** provided from the outside. By moving **{*}$o** to the end of the call, we might decide to give an outside **area** precedence.

More parameters might be provided from the outside that will go directly down to **anotherProc** without us needing to know. This way, our inner procs can provide directly callable features to the outer proc and, resembling a parent class / base class relationship, this starts to form a larger whole.

But let's move on to a more concrete example...

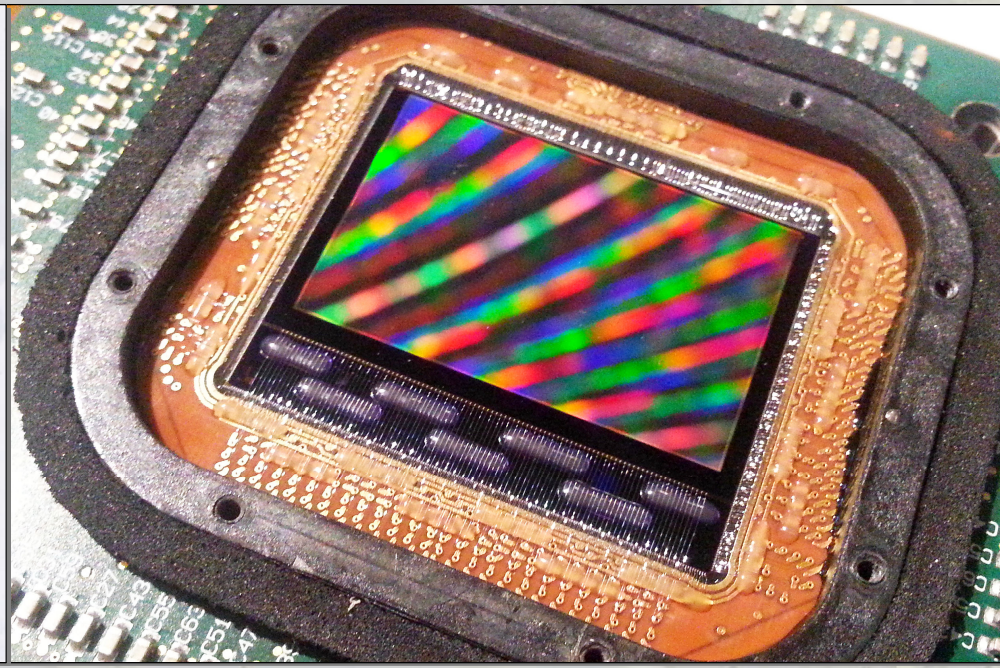
ARRI

- German, Munich-based global supplier of motion picture film equipment
- Founded 1917 by **Ar**nold & **Ri**chter
- Cameras:
    - 1937 – “Arriflex 35” uses the first reflex mirror shutter worldwide
    - 1947 – First Hollywood movie (featuring Humphrey Bogart)
    - Has ARRI earned a cult following in the movie business roughly reminding of apple fanboys – or am I mistaken?
    - 2005 – “Arriflex D-20” is Arri's first digital 35 mm camera
    - 2010 – “Alexa” and “Amira”, using the “ALEV3” B & C 35mm sensors
    - 2015 – The “Alexa65” reintroduces 70mm movie making digitally; >4K
    - Today, movies shot with the Alexa make up a large part of the nominees for major movie awards worldwide
- Lighting: “SkyPanel” – fully tunable LED soft light
- Digital Intermediate:
    - “ARRISCAN” – 35mm film scanner
    - “ARRILASER” – Oscar awarded 35mm film recorder using red, green and blue solid state lasers
- Rental House – The Alexa65 has originally been designed for its exclusive use
- Medical: “ARRISCOPE” – 3D surgical microscope based on the ALEV3
- The recent discontinuation of in-house film processing proves: digital electronics has finally taken over – as it's been predicted for decades

Sensor Tracking

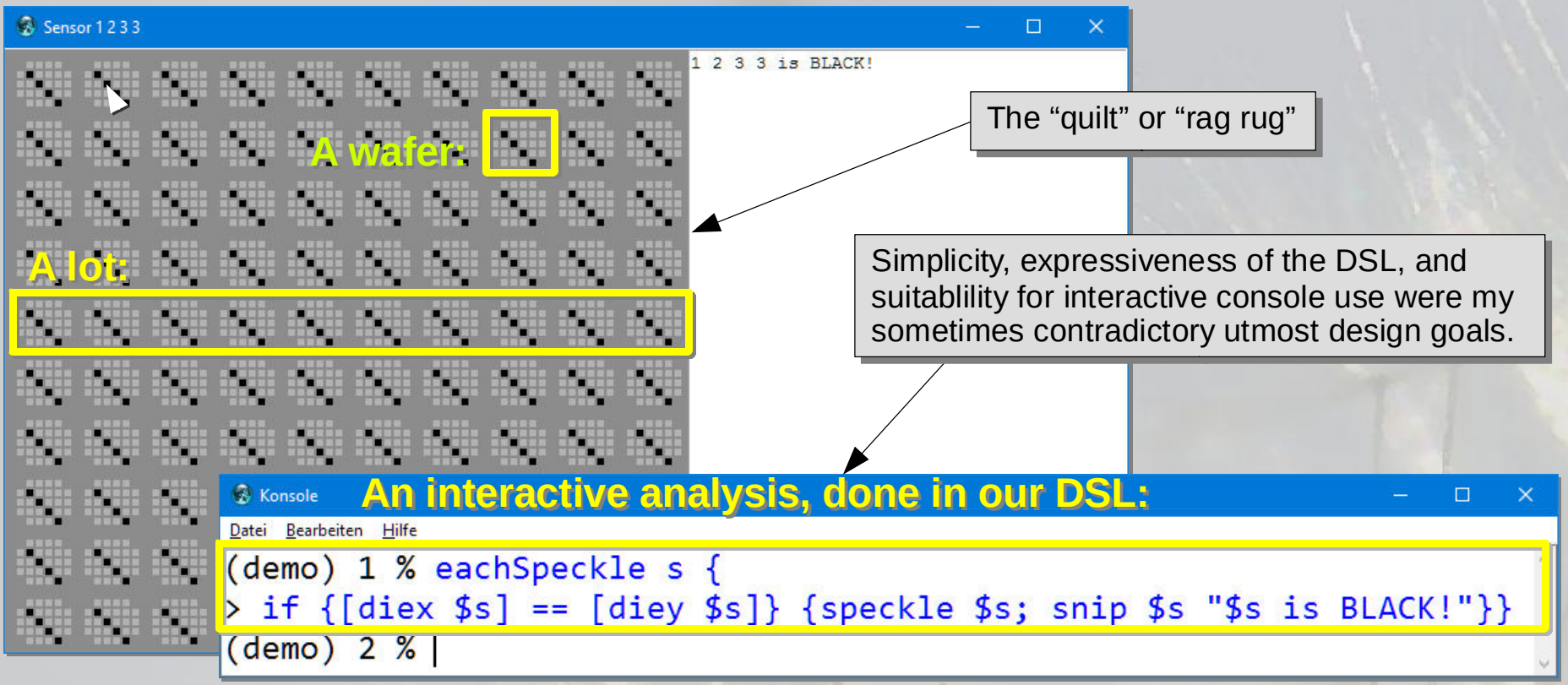
I am part of the sensor team, and we needed for our digital camera projects some means to…

- … collect and access data on our sensors, provided by all parties involved, in order to assemble overviews and statistics showing
  - which sensors hava already been bonded, at best being used in a camera,
  - which sensors are still available,
  - the processing steps each sensor has already passed,
  - which problems have been found on each sensor that may prevent us from applying costly processing,
- … and to do, individually for each Sensor, the kind of in-depth analysis that enables us to rate it. We can do that by using a set of images that's been drawn from each sensor when it was, wafer still uncut, contacted by the needles of the wafer prober.



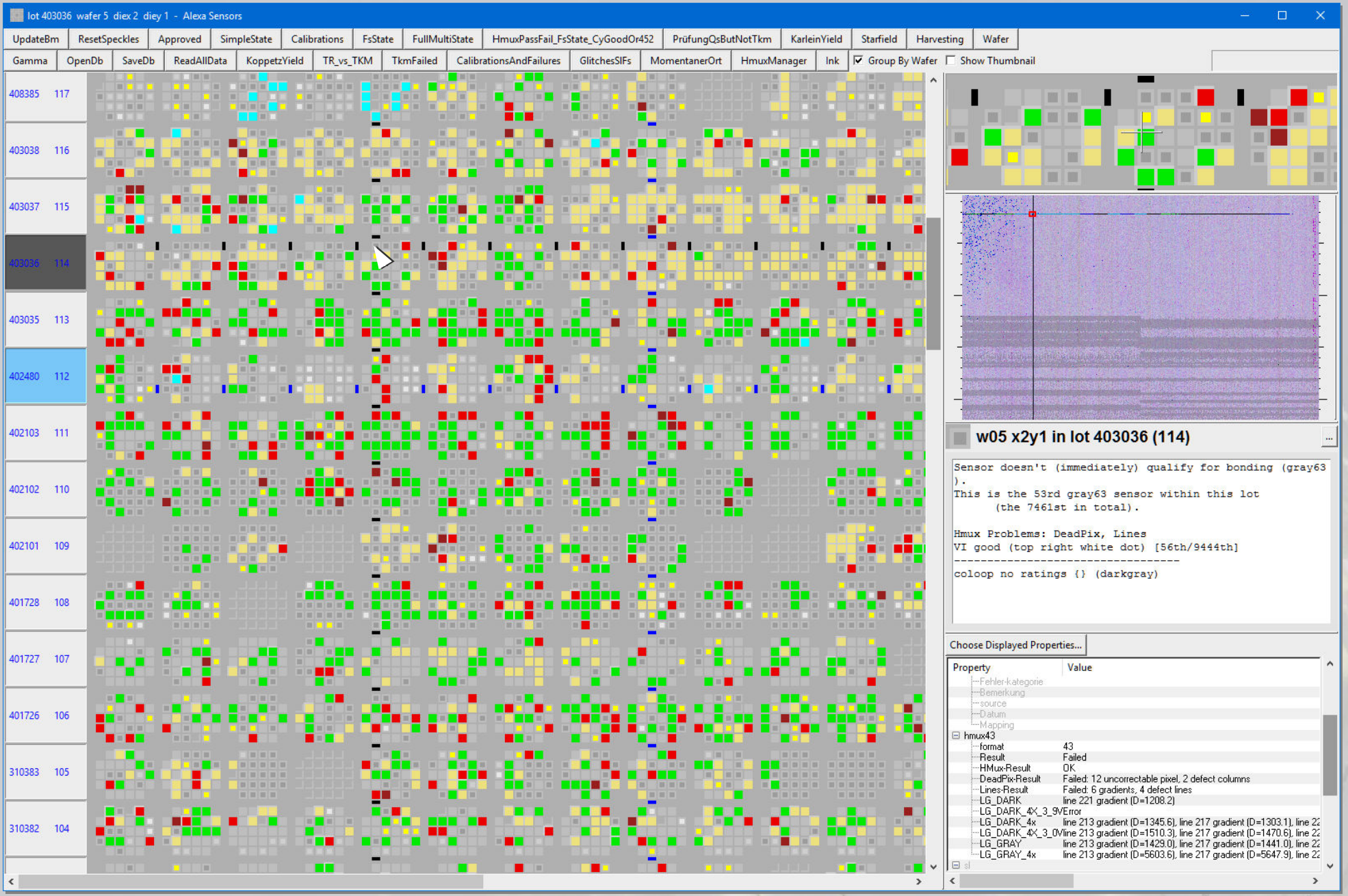
All you need to know about the application are its very beginnings, where I created:

- a grid representing the sensors on the wafers of our sensor lots
- and some commands forming a Domain Specific ~~Language~~ Vocabulary that made it easy to use the console interactively for querying the databases to collect some data, and then color the grid to give some rough overview. Whenever the mouse hovers over the grid, more precise information will be shown in the adjacent text window.



Of course there are other features that will allow to get more detailed information about single sensors – but this overview should be sufficient to understand the examples I'll use.

And even if the current version of this window appears a little bit more complex today …



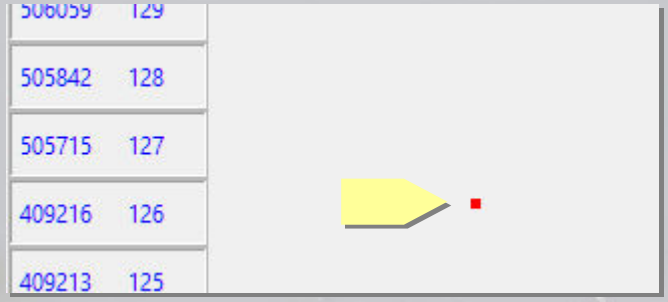
… it should still be possible to recognize the fundamental parts that I've just explained.

Some first single command examples

As a warm up, we're going to manually clear the quilt – which is just a Tk image referenced in the `$::Speckles` variable), and then paint a single speckle:

1a
1b

```
$Speckles blank
speckle {409216 4 2 2} red
```



But `resetSpeckles` will give us a better way to prepare a background for us:

1c

```
resetSpeckles lots [range 126 128]
```

The quilt is much bigger than we need it for our demo purposes, and actions like this can be quite time consuming, so I'm limiting the iteration to a small number of lots.

Originally, my custom iteration did always go over all speckles in the quilt. I'm using a recent change here that made it accept additional parameters (guess how?) changing how it's iterating. You will only see the `lots` iteration parameter in today's example.

`resetSpeckles` passes these parameters transparently down to the iteration – and, as I've hinted before, it neither knows nor checks the additional data it's carrying around.

Our Basic Example Code

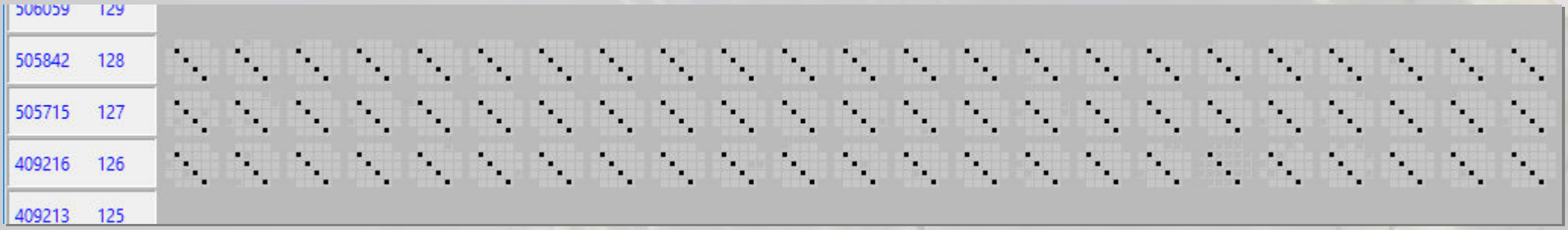
Above this background, we can execute our own algorithm to display specific information.

Our demo algorithm emphasizes sensors for which the x wafer coordinate happens to equal the y sensor coordinate. This doesn't provide us with a really useful feature, but it makes a good and simple demo.

Here's the simplest version of our demo code that you'll see today. We will extend that code gradually, and every time, the new insertions will be marked with a yellow background.

2

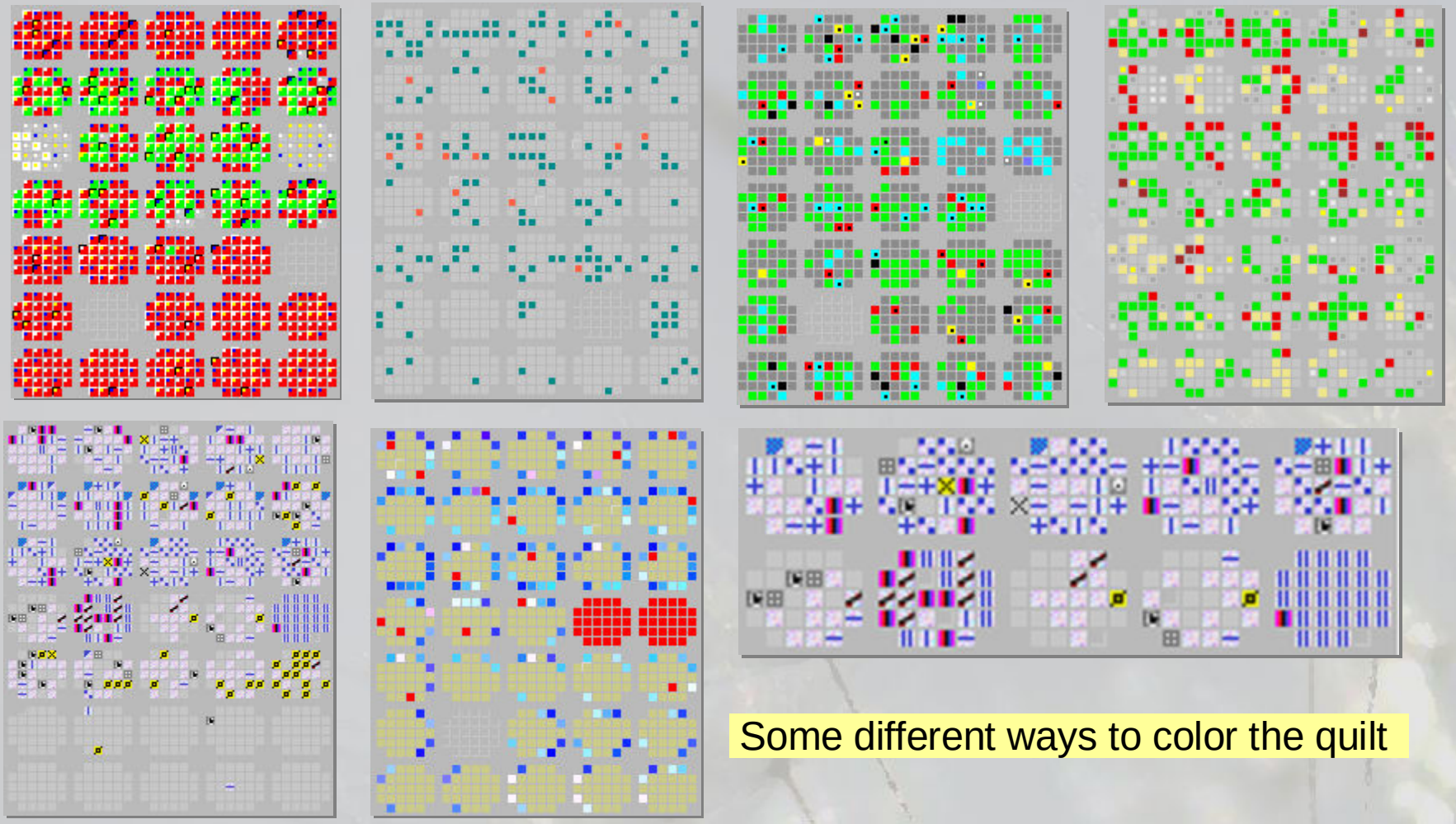
```
eachSpeckle s lots [range 126 128] {
    if {[diex $s] == [diey $s]} {
        speckle $s black 2
        snip    $s "$s is black"
}   }
```



For the rest of this talk, I'm going to paint that same speckles over and over again, with hardly much change in the visible results – but I will gradually increase the complexity of my code – and I hope you'll agree I'm doing that in a reasonable way.

A Real Coloring Of The Quilt Will Always Be More Complex ...

The quilt is intended to show as much information about our sensors as possible. Therefore, the relevant algorithms are usually much more complex and will need to access my databases. But I wanted to keep my demo short and stay away from the databases.



Some different ways to color the quilt

… But The Overall Architecture Is Always The Same.

How To Provide Visible Feedback While The Algorithm Is Running

The loop we've just written will execute atomically, making our application unresponsive while it's running.

If we're iterating through the full quilt, the user has no clue how long he will have to wait. Sometimes, he might even suspect the application might be caught in an infinite loop.

Of course we could add an **update** somewhere inside the loop - with all the potential drawbacks of re-entering the event loop from somewhere deep down on the stack.

A new **action** command offers the alternative to start its subcommand as a **coroutine**, giving our loop the chance to yield and restart itself using an **after** …

3a

```
action resetSpeckles lots [range 126 128]
```

… and I've created a **suspend** command to interact closely with **action**, caring about all the necessary suspending and restarting for us:

3b

```
action eachSpeckle s lots [range 126 128] {
    if {[diex $s] == [diey $s]} {
        speckle $s black 2
        snip    $s "$s is black"
        after 20 ;# pretend that's a complex algorithm
        suspend $s
}   }
```

Putting Both Parts Together

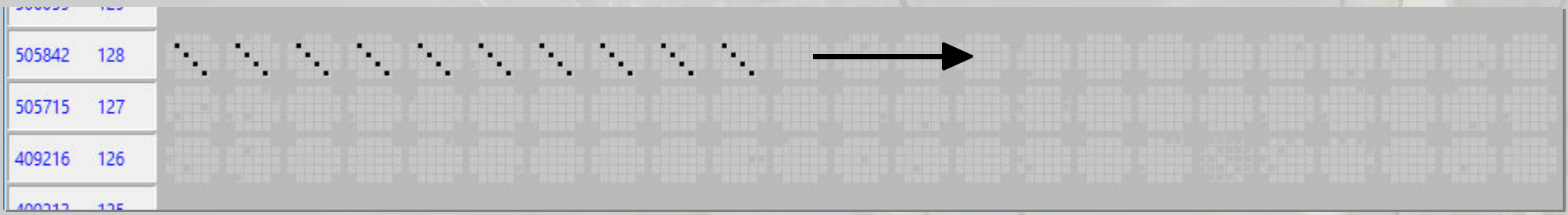
We can only have one single active action at a time. Starting a new one will immediately stop a currently running action.

The reasoning behind this is the same as the reasoning against multithreading.

But we can wrap our two separate actions – i.e. the preparation of the background and the main algorithm – inside a lambda, and then start this lambda as an action.

```
action apply {{} {
    resetSpeckles lots [range 126 128]
    eachSpeckle s lots [range 126 128] {
        if {[diex $s] == [diey $s]} {
            speckle $s black 2
            snip    $s "$s is black"
            after   10
        }
        suspend $s
    }
}}
```

Now we can watch while the quilt gets painted in two passes.

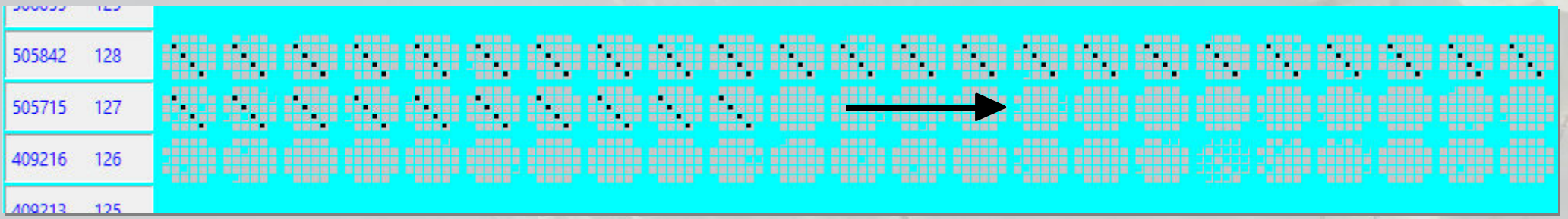


Random Pass-Through Args

We can transform the iteration range that we're using twice into an optional argument for our lambda – and we don't even need to care what's passing through – if we provide an **`args`** parameter thats ready to accept random stuff and hand it down without us introspecting it.

This will only work if all participating procs agree to use a common structure – e.g. a **`dict`**


```
action apply {{args} {
    resetSpeckles {*}$args
    eachSpeckle s {*}$args {
        if {[diex $s] == [diey $s]} {
            speckle $s black 2
            snip    $s "$s is black"
            after 10
        }
        suspend $s
    }
}} lots [range 126 128] bg cyan
```



The **`bg`** parameter, which is interpreted by **`resetSpeckles`**, is routed down there fully transparently, just like **`lots`** goes down to **`eachSpeckle`**. Our short algorithm doesn't know about that, but from an outside perspective both, **`lots`** and **`bg`**, will behave as if they had been original, fully intended and perfectly normal features of our algorithm from the start.

Introducing: (Cascading) Dict Args

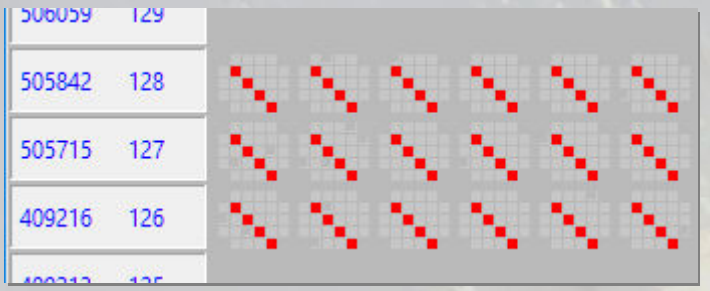
Our code can partake in that mechanism by adding parameters of its own.

In the listing below, there are these distinctive initial code lines again that I've mentioned earlier on the introductory slide.


```
action apply {args {

    set   _ [' color black smaller 2]
    set   o [dict merge $_ $args]
    pluck o color smaller

    resetSpeckles {*}$o
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
            speckle $s $color $smaller
            snip    $s "$s is $color"
        }
        suspend $s
    }
}} lots [range 126 128] color red smaller 0
```



Now I can add parameters specifying color and size of my speckles … and that's just what I did in the demo code above that generated that quilt clipping on its right.

Now if, for example, **`resetSpeckles`** would also use accept a **`color`** argument of its own, then we couldn't provide that from the outside – it is shadowed by ours. And that's ok … an inner call can offer features, but the outer proc gets the final word about its interface.

We Can Move Dict Args Around

Here's that last version of our code again. Now suppose...

7a

```
action apply {args {

    set   _ [' color black smaller 2
                                    ]
    set   o [dict merge $_ $args]
    pluck o color smaller

    resetSpeckles {*}$o
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                speckle $s $color $smaller
                snip    $s "$s is $color"
        }
        suspend $s
    }
}} color red smaller 0 lots [range 126 128
      ]
```

We Can Move Dict Args Around

Suppose we would like to add an optional **bg** parameter of our own … maybe, to paint all the Speckles that we're not going to highlight in **color**. Of course, we might find a better name instead of **bg**, but let us explore that name collision just for the sake of the example.

7a

```
action apply {args {

    set   _ [' color black smaller 2
                                    ]
    set   o [dict merge $_ $args]
    pluck o color smaller

    resetSpeckles {*}$o
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                speckle $s $color $smaller
                snip    $s "$s is $color"
        }
        suspend $s
    }
}} color red smaller 0 lots [range 126 128
      ]
```

We Can Move Dict Args Around (Step 1)

Suppose we would like to add an optional **bg** parameter of our own … maybe, to paint all the Speckles that we're not going to highlight in **color**. Of course, we might find a better name instead of **bg**, but let us explore that name collision just for the sake of the example.

If we don't **pluck** that parameter, it won't be there as a variable – and we will have to access it using the o command. In this case, it will also go down to **resetSpeckles**.
If this was intended, everything is fine. A Dict Arg can be shared like this – see the **lots** arg.

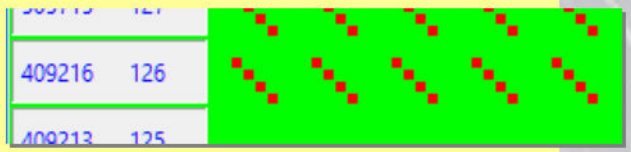
7b

```
action apply {args {

    set   _ [' color black smaller 2   \
                bg    blue                ]
    set   o [dict merge $_ $args]
    pluck o color smaller

    resetSpeckles {*}$o
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                speckle $s $color $smaller
                snip    $s "$s is $color"
        } else {speckle $s [o  bg]}
        suspend $s
    }
}} color red smaller 0 lots [range 126 128
      ] bg green1
```

(1) don't pluck it



We Can Move Dict Args Around (Step 2)

Suppose we would like to add an optional **bg** parameter of our own … maybe, to paint all the Speckles that we're not going to highlight in `color`. Of course, we might find a better name instead of **bg**, but let us explore that name collision just for the sake of the example.

If we don't `pluck` that parameter, it won't be there as a variable – and we will have to access it using the o command. In this case, it will also go down to `resetSpeckles`.
If this was intended, everything is fine. A Dict Arg can be shared like this – see the `lots` arg.

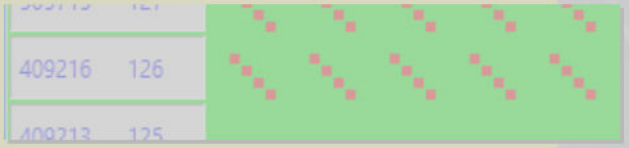
7c

```
action apply {args {

    set   _ [' color black smaller 2   \
              bg    blue                ]
    set   o [dict merge $_ $args]
    pluck o color smaller bg

    resetSpeckles {*}$o
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
              speckle $s $color $smaller
              snip    $s "$s is $color"
        } else {speckle $s    $bg }
        suspend $s          ;# [o  bg]
    }
}} color red smaller 0 lots [range 126 128
     ] bg green1
```

(1) don't pluck it



(2) pluck it

If we don't know how **bg** is used down the cascade, it is better to pluck it (purple insertions, red removals). No **bg** will cascade down … this effectively hides it, and the default will be used.

We Can Move Dict Args Around (Step 3)

Suppose we would like to add an optional **bg** parameter of our own … maybe, to paint all the Speckles that we're not going to highlight in **color**. Of course, we might find a better name instead of **bg**, but let us explore that name collision just for the sake of the example.

If we don't **pluck** that parameter, it won't be there as a variable – and we will have to access it using the o command. In this case, it will also go down to **resetSpeckles**.
If this was intended, everything is fine. A Dict Arg can be shared like this – see the **lots** arg.

7d

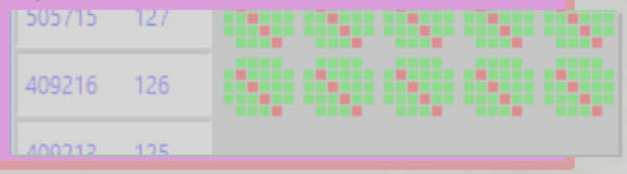
```
action apply {args {

    set   _ [' color black smaller 2    \
                bg    blue  mainBg gray ]
    set   o [dict merge $_ $args]
    pluck o color smaller bg mainBg

    resetSpeckles {*}$o bg $mainBg
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                speckle $s $color $smaller
                snip    $s "$s is $color"
        } else {speckle $s    $bg }
        suspend $s
    }
}} color red smaller 0 lots [range 126 128
      ] bg green1 mainBg white
```

(1) don't pluck it

(2) pluck it



(3) pass down a replacement

If we don't know how **bg** is used down the cascade, its better to pluck it (purple insertions, red removals). No **bg** will cascade down … this effectively hides it, and the default will be used.

If we continue to insist on our own **bg**, but also want to supply another one to **resetSpeckles**, we might push in a replacement like it's shown in the green insertions. But note … the way we're doing that, we're also overriding resetSpeckles' default. Think carefully before doing it that way!

We Can Move Dict Args Around

Suppose we would like to add an optional **bg** parameter of our own … maybe, to paint all the Speckles that we're not going to highlight in **color**. Of course, we might find a better name instead of **bg**, but let us explore that name collision just for the sake of the example.

If we don't **pluck** that parameter, it won't be there as a variable – but I've created the **o** command to provide easy access to it. In this case, it will also go down to **resetSpeckles**. If this was intended, everything is fine. A Dict Arg can be shared like this – see the **lots** arg.

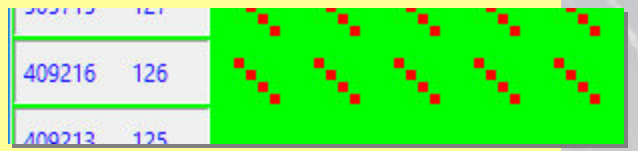
(7d)

```
action apply {args {

    set   _ [' color black smaller 2    \
                bg    blue  mainBg gray ]
    set   o [dict merge $_ $args]
    pluck o color smaller bg mainBg

    resetSpeckles {*}$o bg $mainBg
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                speckle $s $color $smaller
                snip    $s "$s is $color"
        } else {speckle $s [o $bg]}
        suspend $s          ;# [o  bg] ↔ $bg
    }
}} color red smaller 0 lots [range 126 128
      ] bg green1 mainBg white
```

(1) don't pluck it



(2) pluck it

(3) pass down a replacement

If we don't know how **bg** is used down the cascade, its better to pluck it (purple insertions, red removals). No **bg** will cascade down … this effectively hides it, and the default will be used.

If we continue to insist on our own **bg**, but also want to supply another one to **resetSpeckles**, we might push in a replacement like it's shown in the green insertions. But note … the way we're doing that, we're also overriding resetSpeckles' default. Think carefully before doing it that way!

How To Make Parts Of Our Algorithm Pluggable

We're not limited to the classical value based type of parameters. We can, just as easily, go even further and make parts of our algorithm pluggable.

In the example below, **speckle** is no longer a command for us but rather: a dict arg containing a partial command. The default value for this parameter is, of course, "**speckle**" to ensure my algorithm will in the default case behave like it did before (this explains the strange repetition below).

I've used that to separately measure the times consumed by the actual drawing and by the algorithm that decides what to draw – I'll demonstrate that on the following slide.

8a

```
proc demoSpeckles8 args {

    set   _ [' color  black smaller 2 bg blue \
              mainBg cyan  speckle speckle   ]
    set   o [dict merge $_ $args]
    pluck o color smaller bg mainBg

    resetSpeckles {*}$o bg $resetBg
    eachSpeckle s {*}$o {
        if {[diex $s] == [diey $s]} {
                {*}[o speckle] $s $color $smaller
                snip           $s "$s is $color"
        } else {{*}[o speckle] $s  $bg}
        suspend $s
    }
}
```

BTW … I've switched from lambda to **proc** now, in order to keep the following demos shorter.

Here are some examples of what you might do with a pluggable **speckle** command:

- Do no speckling at all to measure the time spent in your algorithm and, indirectly, also the time that's used to do the actual painting. When an **action** finishes it will print the time it took in the console – so we don't need to worry about the actual measuring here.

8b
```
action demoSpeckles8 lots [range 126 128] speckle {apply {args {}}}
```

- Count how often **speckle** is called. We will need to check that count in the console afterwards, because **action** returns before the action finishes. (Maybe we need an ActionFinished event.)

8c
```
set count 0; action demoSpeckles8 lots [range 126 128] speckle {
                apply {{args} {incr ::count; speckle {*}$args}}}
```

- Do not paint the speckles, but collect a list of deferred speckles that may be executed later.

8d
```
set deferred ""; action demoSpeckles8 lots [range 126 128] speckle {
                        apply {{args} {lappend ::deferred {*}$args}}}
```

8e
```
action apply {{} {foreach _ $::deferred {
               speckle {*}$_; suspend [lindex $_ 0]}}}
```

Please note: This would also work without using **apply**, but **action** currently logs all its args into the console – and the long list would make the console slow!

- We might experiment with alternative **speckle** implementations using canvas instead of image
- … or even delegate the actual drawing into another thread.

More Pluggable Stuff

Here's the most heavily dissected version I could come up with.

We can configure the type of loop were using (because we might, for example want to iterate only over some sensors provided in a list instead of contiguous parts of the quilt (… which is quite far-fetched in our quilt coloring example, but reasonable when we're implementing other features). This calls for some additional indirection which I don't want to explain in-depth (the red code), because we're soon going to see a better way to achieve the same result.

9a...

```
proc chooseLoop {createLoop var args} {{*}$createLoop $var  {*}$args}

proc overEachSpeckle  {     var args} {' eachSpeckle  $var  {*}$args}
proc overList         {list var args} {' foreach          $var $list}
proc overSensorText   {text var args} {' eachSensorInText $var $text}

proc demoSpeckles9 {args} {
    set   _  ['  coreAlgorithm speckleIfXEqualsY \
                 prepare       resetSpeckles     \
                 loop          overEachSpeckle   ]
    set   o  [dict merge $_ $args]
    pluck o  coreAlgorithm prepare

    {*}$prepare                 {*}$o
    {*}[chooseLoop [o loop] s {*}$o] {
        {*}$coreAlgorithm       {*}$o for $s
        suspend $s
}   }
```

I'm also making the preparation of our background exchangeable (which leads to an interesting set of new combinations), and I'm pulling out our complete core algorithm – which might not be the idea because it leaves us with an overly-generic empty shell, but today it will enable some easy examples.

The more we're approaching genericity, the more we're typically gonna see the expansion operator `{*}`

Just in case you sadly missed the implementation of our core algorithm … here it is:

...9a

```
proc speckleIfXEqualsY {args} {
    set   _ [' for $::SelectedSensor smaller 2 \
                color black speckle speckle     ]
    set   o [dict merge $_ $args]
    pluck o color smaller speckle for

    if {[diex $for] == [diey $for]} {
        $* speckle $for $color $smaller
        snip       $for "$for is $color"
        after 10
}   }
```

You should still recognize the last four lines here from the previous example.

The only thing that happened is that this part of the code has become a procedure of its own, and it therefore also got to parse its own arguments.

It brought over arguments like **color** or **smaller** from our earlier versions which aren't used any more in the code that's now calling it. Additionally, the sensor id which was the loop variable before needs to be extracted from the command line. We decided to use the **$::SelectedSensor** as the default for that, which seems to be a good choice especially this is to be used at the command line.

9b

```
speckleIfXEqualsY  ;# try that after selecting different sensors
```

We are still plucking the args we're using … but to which effect? We received a copy of our caller's args … so if we would, as in our earlier example, use **bg** and wanted to replace it with **mainBg** before it's going down to resetSpeckles … we couldn't. **resetSpeckles** is called by our caller, we cannot change our callers args, and anyway … **resetSpeckles** will be called before we are called.

In the long run, extracting this procedure might have been a bad idea. Problems like this seem to set the limits of the dict args technique. I've never been bitten by that, and I'm expecting that to happen anytime.

More Pluggable Stuff

Different Looping styles, different preparations...:

This example loops over only the three sensors of a short list.
In the result, only these three speckles will be painted on the quilt because the loop argument is cascaded down to resetSpeckles which will use it in the same way.

9c

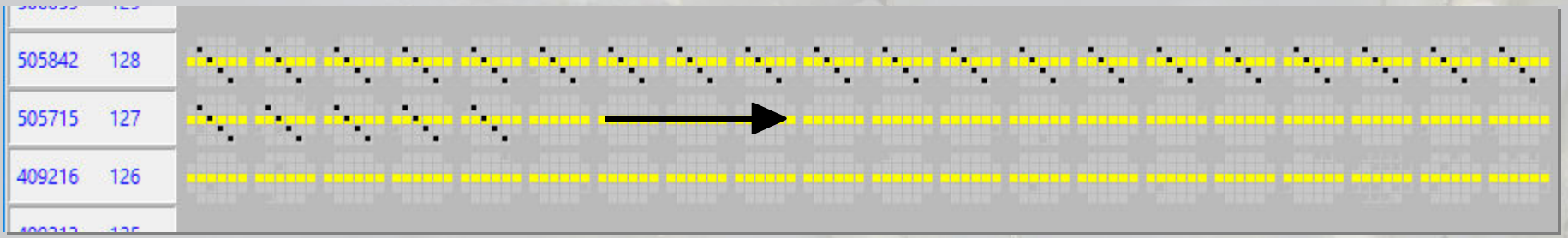
```
action demoSpeckles9 lots 126 loop [' overList {
    {409216 3 3 3}  {409216 3 3 4} {409216 3 4 4}
}] color white smaller 0
```

- Here's a complex example in which I'm doing three passes by using **demoSpeckles9** twice:
  - The outer instance (= last pass) of **demoSpeckles9** uses a second instance of **demoSpeckles9** instead of its default preparation.
  - The second instance uses the default preparation (→ **resetSpeckles**), but replaces the core algaorithm to paint the yellow speckles below.

9d

```
action demoSpeckles9 lots {126 127 128} prepare {demoSpeckles9 coreAlgorithm {
     apply {{args} {if {[diey [_ $ args for]] == 3} {speckle $_ yellow}}}  }}
```

  - This image has been taken right in the middle of the algorithm's third pass.



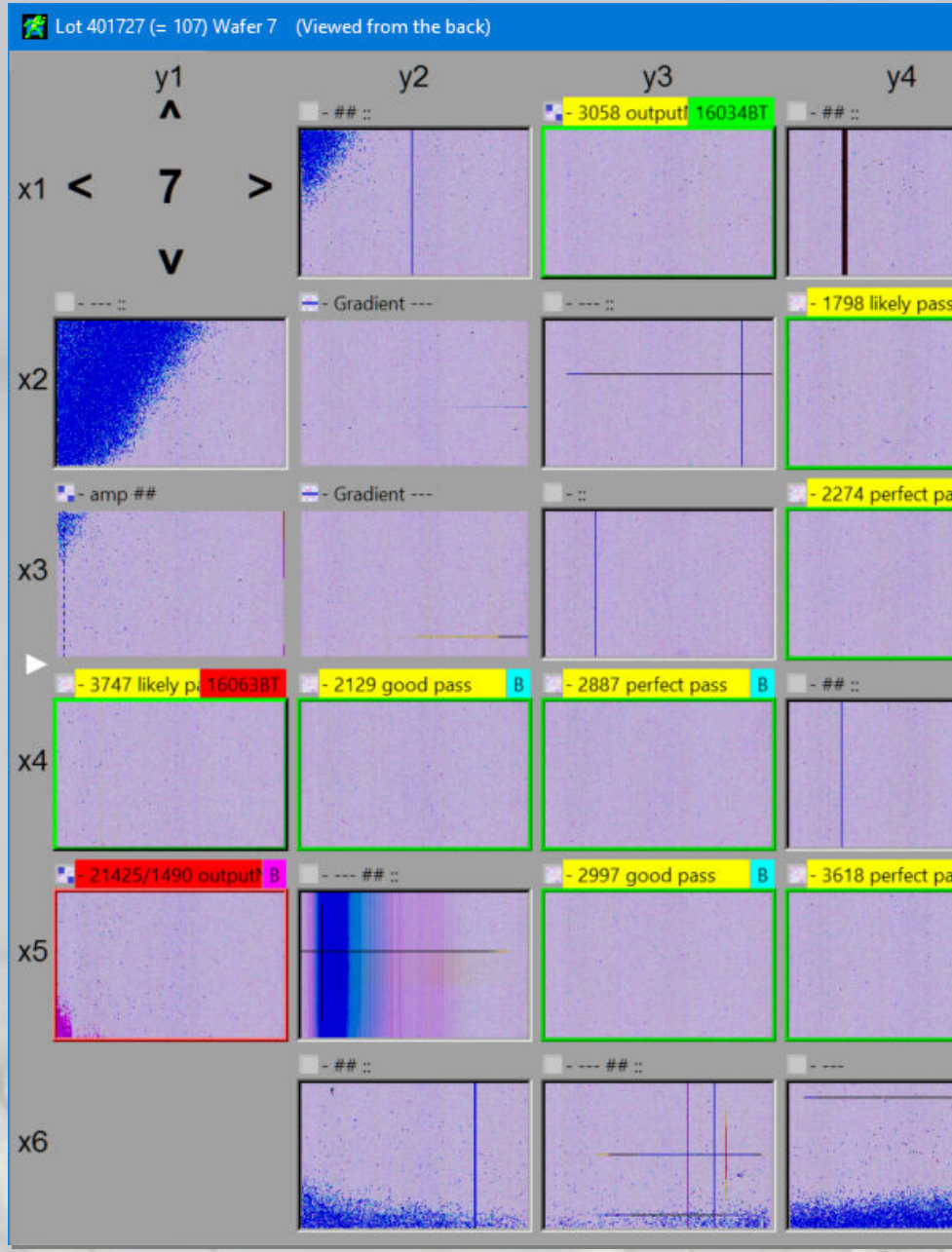
A More Realistic Plugging Example

In my wafer overviews, I'm using all available means to encode information. The idea was that information should be visible if you look for it, but also easy to neglect if you look for something else.

In the sensor reliefs, I'm showing sensors as sunken if they are still waiting to be rated manually (If a sensor's problem is readily visible I usually don't rate it), while bonded sensors appear as raised (ridge if both should be true).

By some recent changes, this has become mostly redundant – so this algorithm is a candidate for being made pluggable as '**`relief`**', which will give the user a possibility to encode different information in the sensor relief.

Moreover, by accepting Tk's original relief words in addition to implementatons of the required interface, we might also mimic Tk's **`-relief`** option.



Cascades Of Differently Relevant Default Values

The wafer overview is created by the **showWafer** procedure which in turn calls **showSensor** for every sensor on the wafer. Together, they got roughly 50 parameters. (BTW – yes, I'm simply using procs. No classes, no megawidgets.)

showSensor defines its defaults which may or may not be replaced from within **showWafer**. For example: the thumbnail size whicg should be smaller if a full wafer is shown. But … what if some user has a high resolution display and needs to increase the thumbnail size? We can prepare for that by merging in additional sets of defaults:

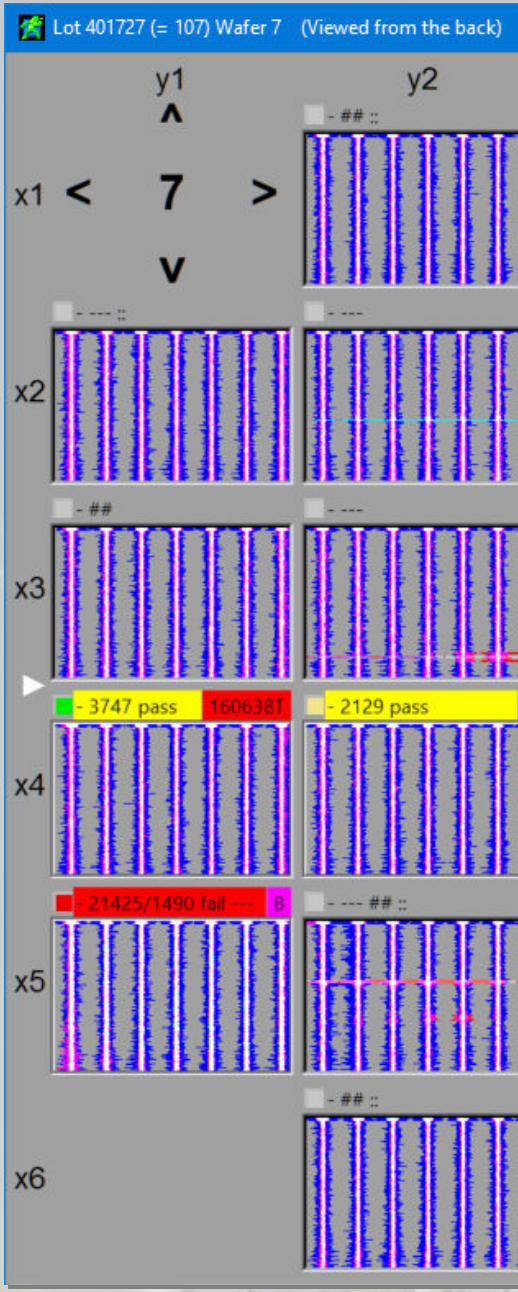
```
set o [dict merge $_ $::DefaultValues(ShowWafer) $args]
```

This variable can be edited in the console or with a GUI, and its values will take precedence over the hard coded defaults, but not over the args.

And it doesn't end here. Each time the overview is created, it logs its command line in the console. Thismakes it easy to copy, paste and tweek the call. The image on the right was created by adding '**aspect gradients**'.

If I have to do that over and over again after GUI clicks, I'd like to specify it once and for all. We can help here without disturbing other calls by merging in another set of defaults in the code bound to the GUI event:

```
bind .quilt <3> {showWafer for [getSensorAt %x %y] and maybe \
     more adjustments {*}$::DefaultValues(ShowWaferViaQuilt) }
```



After I've been using dict args quite some time, something creepy happened:

Dict Args Ate Up My Positional Args !!!

I began to use dict args to fill the needs of an evolving application. Nearly all procedures started out using only positional arguments. As the appliction grew, I added more and more positional arguments and made them optional. But eventually, I've had enough of needlessly having to fill in six values just to be able to specify the seventh.

So … this mechanism seemed the easiest and quickest way to replace a long row of optional positional arguments – but it was never intended to replace the formerly mandatory arguments.

Let's take at the wafer overview window as an example: You would, among many optional things, always require to specify at least the wafer you wanted to see, and a window name … wouldn't you?

No, you wouldn't.

It makes perfect sense to use a default window name, and to reuse this window whenever a wafer overview is requested. This way, there's usually always at most one wafer overview window – except when a second (third …) one has been requested explicitly.

And about the wafer … after a `$::SelectedSensor` had been introduced, this is the mandatory choice as the default of a then optional argument. Especially when called interactively via console.

Positional arguments have become a rare thing.

Migration Of Features / Growth Of Tiny Languages

So … did you notice?

We are just witnessing how parts of the code that were intrinsic to procs in our earlier, more classical programming style, are starting to regroup, migrating away.

Some parts of the code have a tendency to migrate upwards, towards the caller who will gain more flexibility by exchangeable features he might override if there's any need – just like we've seen in the `speckle` or `coreAlgorithm` examples.

Others features have just the opposite tendency to seep downwards where they are assimilated by and extending existing features, from where they also add to the callers flexibility – for example when `eachSpeckle` receives additional `fromList` or `fromText` parameters, voiding our earlier, overly complicated attempt to iterate over different kinds of containers. Or maybe, when a `relief` parameter will, in addition to the relief values defined by Tk, accept a partial command that will calculate the required relief.

Either way, this migration is forming “tiny languages”.

I've heard some features of Tcl, like “`end-1`” style list index specifications, being called tiny languages. I think we can rightfully call a `relief` parameter which accepts different kinds of values a “tiny language” in its own right … and maybe we should also use this term for Tk's original **relief**.

This is not a fact, but rather a question of perspective … and in my changing perspective I'm also ready to call the `eachSpeckle` command with all its parameters a tiny language and, yes, our `demoSpeckles` example too. Tiny languages from the same domain (e.g. sensors/quilt) are forming domain specific languages. And dict args (currently, in my eyes) seem to form a common namespace within a such a DSL, where every party may add freely, but responsibly. This is a current key aspect of my world view; objectoriented hierarchies are still there but receding behind this, supporting it.

“TCL” suddenly seems to be the abbreviation of “Tiny Cooperative Languages”.

Co-Looping = Coroutines Sharing A Single Loop

Currently, we still have to wait while our speckles are getting painted in two passes – first, the background will get refreshed, and then before the main information can start to appear … but this takes too long!

Some of the real specklings I've created are using even more passes.

Early versions of my software had even worse behavior because, back then, I used to add new lots at the bottom. This meant that I had to wait to the very end before the information I've been most eagerly waiting for appeared. (Well … at least I had some visual feedback).

Of course, I do want to see information concerning my newest lots early, and without having to wait through multiple passes.

Again, coroutines can provide an answer – and I want to show how I'm using them to create a a chain of procedures that can share one common loop.

Only the first coroutine in this chain (the "master") will do the actual looping, and provide this loop to all its slaves.

Master and slaves need to be instrumented differently, and I will show both these instrumentations.

All the necessary communication is contained in the master or the slave – we don't need any additional framework, but I will eventually show how to factor out a utility procedure.

All in all, we will insert only about 10 lines, and these will form a pretty compact block.

The Co-Loop Master

resetSpeckles is already prepared to act as a slave upon request, so our first experiment is to transform **demoSpeckles** into a master, and to use that feature of **resetSpeckles**.

10

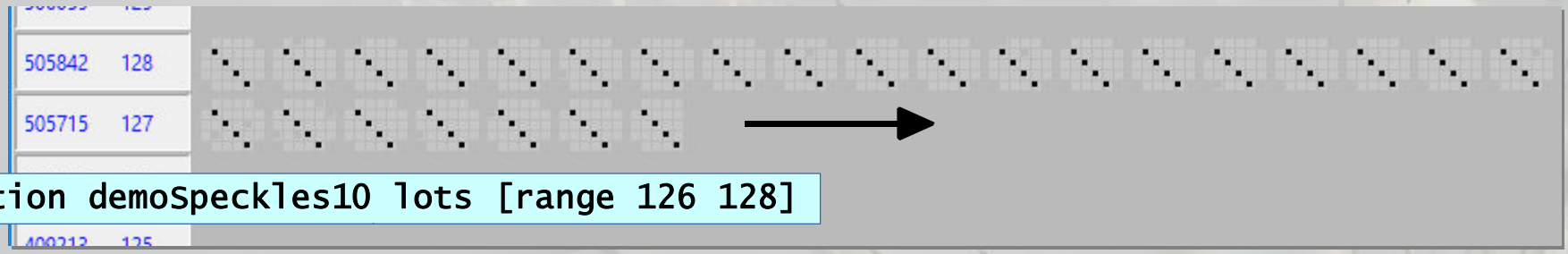
```
proc demoSpeckles10 {args} {
    set   _  [' coreAlgorithm speckleIfXEqualsY prepare resetSpeckles]
    set   o  [dict merge $_ $args]
    pluck o  prepare coreAlgorithm

    set co nop
    if {[llength $prepare]} {
        set        co  coloop#[incr ::ColoopCount]
        coroutine $co  do $prepare {*}$o coloop yes
        tidy -     co  {{} {if {$co ne ""} {$* co}}}
    }

    eachSpeckle s  {*}$o  {
        {*}$co $s  ;# no args needed here - it already got them!!!
        {*}$coreAlgorithm {*}$o for $s
        suspend $s
}   }
```

1 2 3

- 1 When starting the coroutine, the parameter `coloop` is passed down to the slave using dict args to let it know that it is supposed to act as a slave.
- 2 **tidy** ensures that the lifetime of our coroutine is, through the use of var traces, in a technique that's called RAII in C++, controlled by the variable **co** which is our reference to the coroutine.
- 3 In each iteration step, the slave is triggered with the current loop value; no value meaning stop.




```
action demoSpeckles10 lots [range 126 128]
```

The Co-Loop Slave

To prepare our code as a slave, we need to...

1. • accept a coloop parameter hat tells us if we should act as a slave
2. • do an infinite loop instead of looping over our sensors
3. • `yield` to accept the current loop value
4. • `break` the infinite loop when didn't get a loop value

```
proc demoSpeckles11 {args} {
1       set    _  [' coreAlgorithm speckleIfXEqualsY coloop no  \
                    prepare resetSpeckles loop overEachSpeckle ]
11      set    o  [dict merge $_ $args]
        pluck  o  coreAlgorithm coloop prepare

        set co nop
        if {[llength $prepare]} {
            set        co  coloop#[incr ::ColoopCount]
            coroutine $co  do {*}$prepare {*}$o coloop yes
            tidy -     co  {{if {[llength $co]} {$* co}}}
        }
2       {*}[if {$coloop} {' while 1} else {chooseLoop [o loop] s {*}$o}] {
3           if {$coloop} {set s [yield]}
            $* co $s
4           if {![llength $s]} {break}
            suspend  $s

            do $coreAlgorithm {*}$o for $s
}   }
```

co-looping in all its glory

Remember *not* to pluck the `loop` … it will be needed down the chain if we're not co-looping.

Co-Loop Instrumentation As A Library Procedure

We can create a library procedure from that instrumentation code, and using that library procedure will make our speckling algorithm simple again.

Don't try to immediately grasp this library procedure below … it's only a rough draft. Go ahead to the next slide that will show the final version of our example code and some last calls using it.

12

```
proc coloop {coloopflag slave loopvar loop arglist body} {
    set o [dict merge {allowColoop yes} $arglist]
    upvar $loopvar it
    set co nop
    set allowed [$ o allowColoop]
    set slave   [' {*}$slave {*}$arglist coloop $allowed]
    if {[llength $slave]} {
        if {$allowed} {
            set        co  coloop#[incr ::ColoopCount]
            coroutine $co  do {*}$slave                    ;# Start the slave and
            tidy -     co  {{if {[llength $co]} {$* co}}} ;# let it tidy itself
        } else {do {*}$slave}                  ;#  or execute the slave in one go
    }
    {*}[if {!$coloopflag || !$allowed} {
        chooseLoop $loop it {*}$arglist        ;#  Masters do the original loop,
    } else {' while 1}] {                      ;#  Slaves loop indefinitely.
        if {$coloopflag} {set it [yield]}      ;#  Take it from your master
        $* co $it                              ;#  then pass it to your slave
        if {![llength $it]} break              ;#  and if our loop isn't done
        uplevel 1 $body                        ;#  call the loop body.
}   }
```

This implementation additionally offers to choose between the original loop and the co-looping version by offering an additional dict arg.

The Final Version Of Our Demo

13

Heavy use of dict args,

higher order loops -

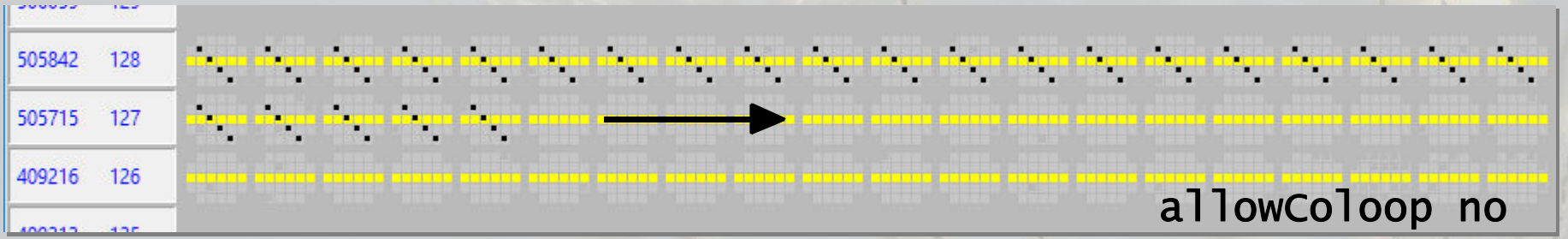
otherwise, we're back to simple again.

```
proc demoSpeckles {args} {
    set    _  [' coreAlgorithm speckleIfXEqualsY coloop no  \
                  prepare resetSpeckles loop overEachSpeckle ]
    set    o  [dict merge $_ $args]
    pluck  o  coreAlgorithm coloop prepare

    coloop $coloop $prepare s [o loop] $o {
        suspend $s
        do $coreAlgorithm {*}$o for $s
}   }
```

With this final version, we can revisit our old three pass demo, while using all our new features. Admittedly, this is not the code I'd like to show Tcl beginners. But it's possible to read it if you take the due time – after all, it is crammed with features. For me, this is gluing of a higher level than I've known it five years ago – yet it's still possible to do it as some quick console throwaway code.

```
action demoSpeckles lots {126 127 128} prepare {demoSpeckles coreAlgorithm {
    apply {{args} {if {[diey [_ $ args for]] == 3} {speckle $_ yellow}}}
}} allowColoop yes
```



Conclusion

I think we've come a long way from that initial two-liner:

```
eachSpeckle s lots [range 126 128] {if {[diex $s] == [diey $s]} {
    speckle $s black 2; snip $s "$s is black"
}}
```

We've added lots of flexibility, but still our code is a simple

```
proc NAME args {
    PARSE_YOUR_ARGS_SOME_STANDARD_WAY
    LOOP ... {
        DO_YOUR_STUFF
}   }
```

... and this amazes me. We didn't need to bend or twist our original straight forward procedures. There are no complex frameworks we need to learn, no adapter classes we need to subclass before we might think about appoaching our main task - only some very simple helper commands.

Everything seems to be falling into place quite easily.

Thank you.

Thom@s-Lang.com