How to Build Emergent Software Systems

Tutorial

Barry Porter & Roberto Rodrigues Filho
School of Computing and Communications
Lancaster University

b.f.porter@lancaster.ac.uk
r.rodriguesfilho@lancaster.ac.uk

Tutorial Overview

- Self-Description
- Adaptation
- Component Search
- Compositional Reasoning
- Perception
- Learning

Morning

Afternoon

NAMING THINGS

(and code organisation)

[but not cache invalidation]

Reminder on components

```
interface Sorting {
    Data[] sortArray(Data array[])
    }
```

```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        return 0
        }

    }
```

```
component provides Sorting {

    Data[] Sorting:sortArray(Data array[])
        {
        // ...
        }

    }
```

Reminder on components

There is no “name” here

```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        return 0
        }

    }
```

```
interface Sorting {
    Data[] sortArray(Data array[])
    }
```

```
component provides Sorting {

    Data[] Sorting:sortArray(Data array[])
        {
        // ...
        }

    }
```

Types and names

- Dana does not care about type names at runtime; all type names are relevant at compile time only

- At runtime, the type system only cares about the *structure* (syntax) of a type when checking for compatibility
  - This has the benefit that we can load a lot of different components, over time, from different places, without needing to care about whether or not they happen to have used some of the same type names to mean different things

Types and names

- Wiring other things

```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        return 0
        }

    }
```

```
interface Sorting {
    Data[] sortArray(Data array[])
    }
```

```
interface ReverseList {
    Data[] reverse(Data lst[])
    }
```

dana.rewire(a :> “Sorting”, b :< “Sorting”)

dana.rewire(a :> “Sorting”, c :< “ReverseList”)

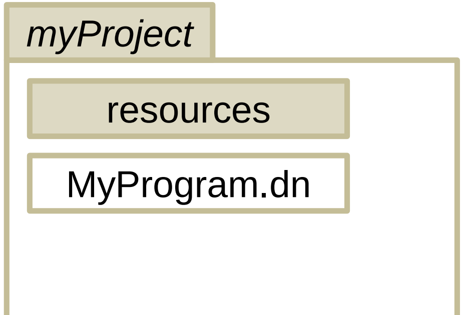
Types and names

- All naming of things is therefore left as a piece of semantic information for the composer (which you create) to decide how to interpret

- This avoids naming collisions during unforeseen adaptation, and also gives a wide range of potential to create very different composition systems

Default composition

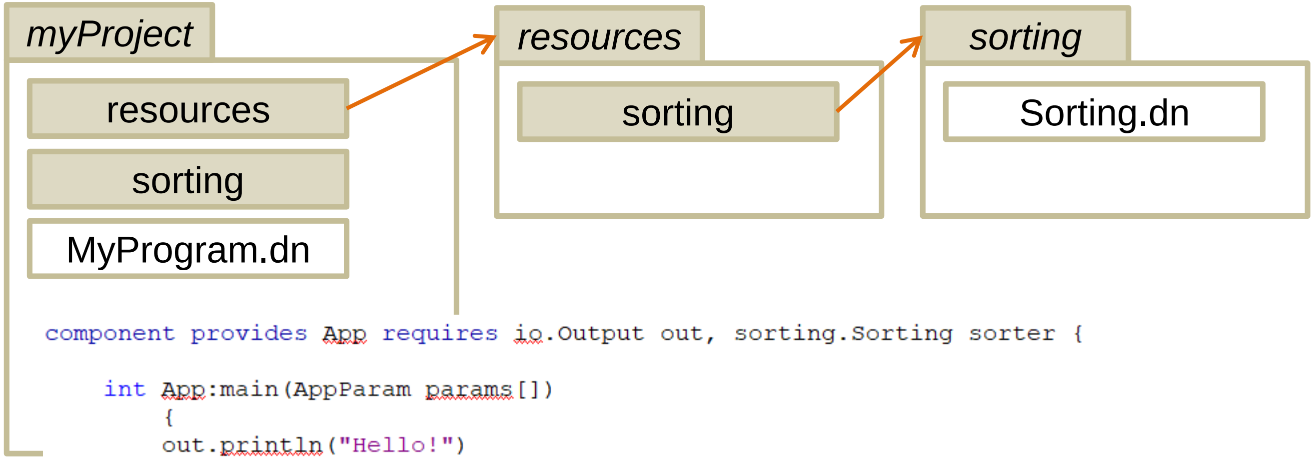
- …having said this, it's useful to define *default* composition behaviour because it makes common-case coding easier




```
component provides App requires io.Output out {

	int App:main(AppParam params[])
		{
		out.println("Hello!")

		return 0
		}

	}
```

Default composition

- …having said this, it's useful to define *default* composition behaviour because it makes common-case coding easier

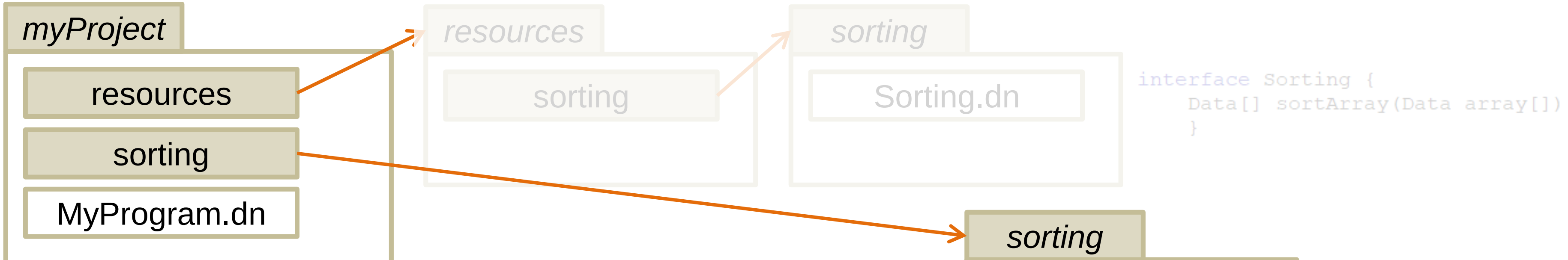



```
interface Sorting {
    Data[] sortArray(Data array[])
    }
```

```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        return 0
        }

    }
```

Default composition

- …having said this, it’s useful to define *default* composition behaviour because it makes common-case coding easier

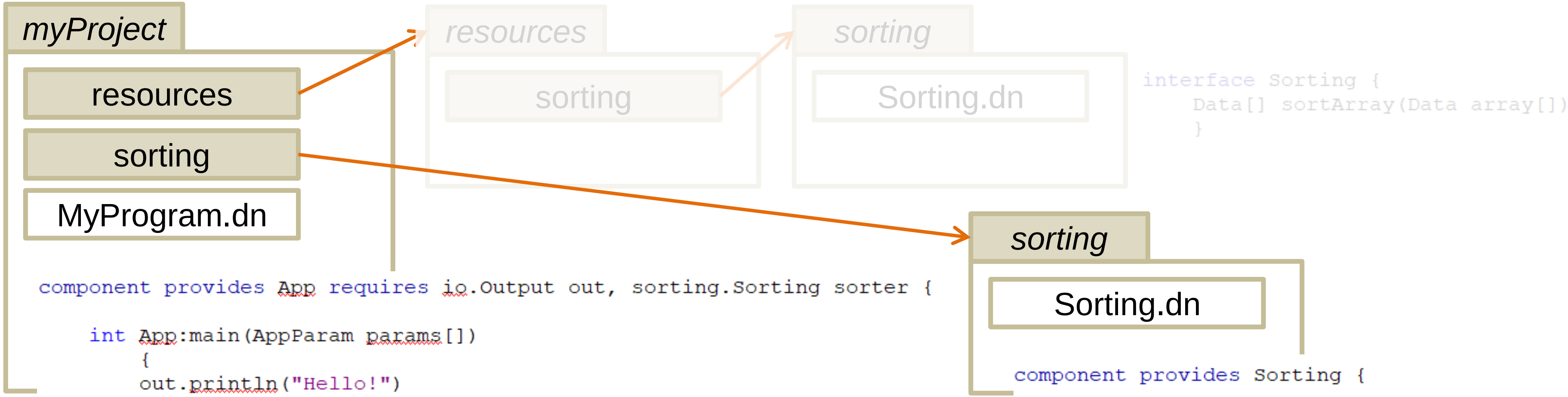



```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        return 0
        }

    }
```


```
component provides Sorting {

    Data[] Sorting:sortArray(Data array[])
        {
        // ...
        }

    }
```

Default composition

- …having said this, it’s useful to define *default* composition behaviour because it makes common-case coding easier



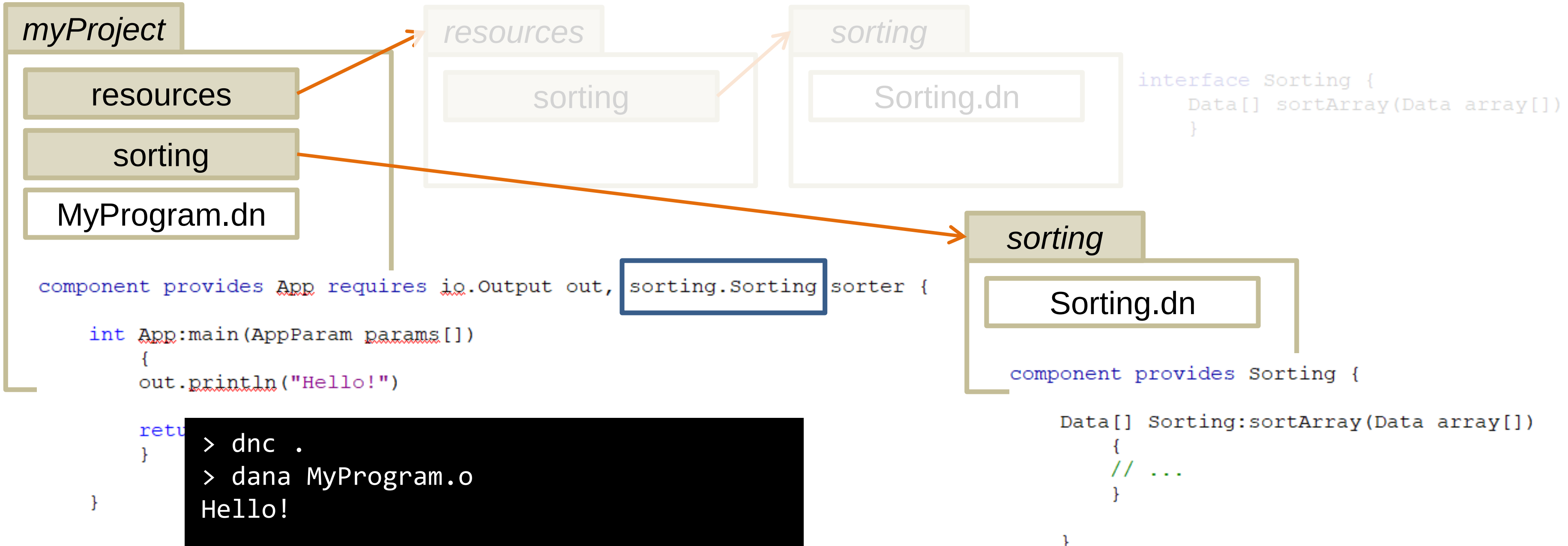

```
component provides App requires io.Output out, sorting.Sorting sorter {

    int App:main(AppParam params[])
        {
        out.println("Hello!")

        retu
        }

    }
```

```
> dnc .
> dana MyProgram.o
Hello!
```

```
component provides Sorting {

    Data[] Sorting:sortArray(Data array[])
        {
        // ...
        }

    }
```

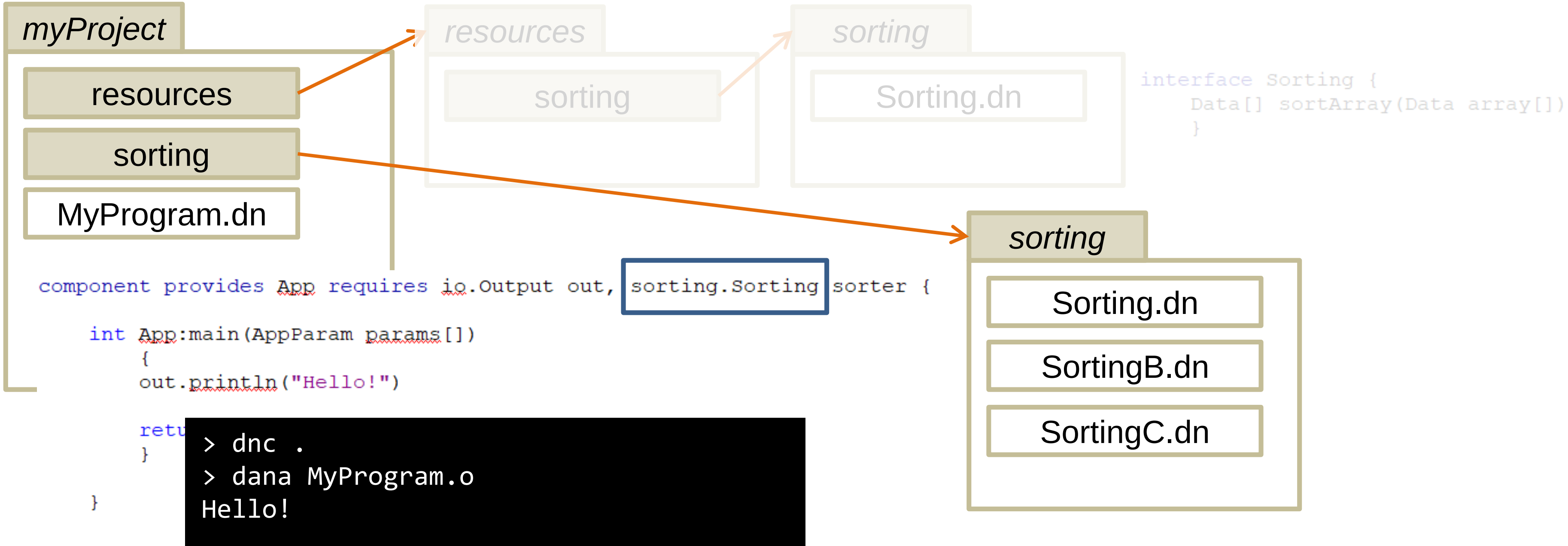
Default composition

- …having said this, it's useful to define *default* composition behaviour because it makes common-case coding easier



Default composition

- This same strategy supports finding *sets* of possible implementations in the same location



Default composition

- This search is available through an API, implemented by an open-source component in our standard library

Interface **Search**

access this type via: `composition.Search` (provides, requires or uses)

Utilities to locate components.

Functions

```
bool isDanaComponent(char path[])
String[] getComponents(char intfPackage[])
String[] getComponentsIn(char intfPackage[], String searchPaths[])
char[] getDefaultComponent(char intfPackage[])
```

bool isDanaComponent(char path[])

Check if a file is a compiled Dana component.

String[] getComponents(char intfPackage[])

Get the list of components that provide the given interface, using standard search paths.

intfPackage The interface type to search for, complete with its package path in dot format. A semantic flavour wildcard can optionally be appended to intfPackage following a colon, as in Parser:*, to locate all implementations of all semantic flavours, excluding plain implementations of the interface. A specific semantic flavour can be appended such as Parser:multipart to find all implementations of that specific flavour.

String[] getComponentsIn(char intfPackage[], String searchPaths[])

Get the list of components that provide the given interface, using the given directories as search paths.

intfPackage The interface type to search for, complete with its package path in dot format. A semantic flavour wildcard can optionally be appended to intfPackage following a colon, as in Parser:*, to locate all implementations of all semantic flavours, excluding plain implementations of the interface. A specific semantic flavour can be appended such as Parser:multipart to find all implementations of that specific flavour.

Default composition

- Dana's standard library is structured in exactly the same way, so is part of the search for implementations

is PC > Windows (C:) > ProgramFiles > Dana > components >

Search components

Name	Date modified	Type	Size
composition	09/01/2019 12:00	File folder	
data	09/01/2019 12:00	File folder	
doc	09/01/2019 12:00	File folder	
encoding	09/01/2019 12:00	File folder	
examples	09/01/2019 12:00	File folder	
io	09/01/2019 12:00	File folder	
locale	09/01/2019 12:00	File folder	
media	09/01/2019 12:00	File folder	
ml	09/01/2019 12:00	File folder	
net	09/01/2019 12:00	File folder	
os	09/01/2019 12:00	File folder	
pal	09/01/2019 12:00	File folder	
resources	09/01/2019 12:00	File folder	
time	09/01/2019 12:00	File folder	
ui	09/01/2019 12:00	File folder	
util	09/01/2019 12:00	File folder	
ws	09/01/2019 12:00	File folder	
compinfo.dn	14/02/2017 14:54	DN File	3 KB
compinfo.o	09/01/2019 12:00	O File	39 KB
docgen.dn	16/11/2018 16:00	DN File	21 KB
docgen.o	09/01/2019 12:00	O File	229 KB
esher.dn	09/08/2018 08:22	DN File	13 KB
esher.o	09/01/2019 12:00	O File	169 KB

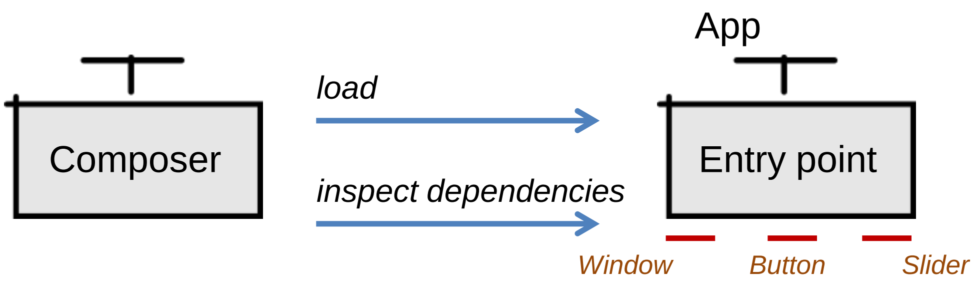
***resources* directory** →

BUILDING A COMPOSER

Building a composer

- The role of a composer is to decide how a system is composed from its parts, before and after deployment
  - Some of these might use our standard approach to naming and search, some might not
  - Some might use machine learning, some might not
  - By making this aspect programmable, you decide how to exploit the capabilities of the programming model for what you need

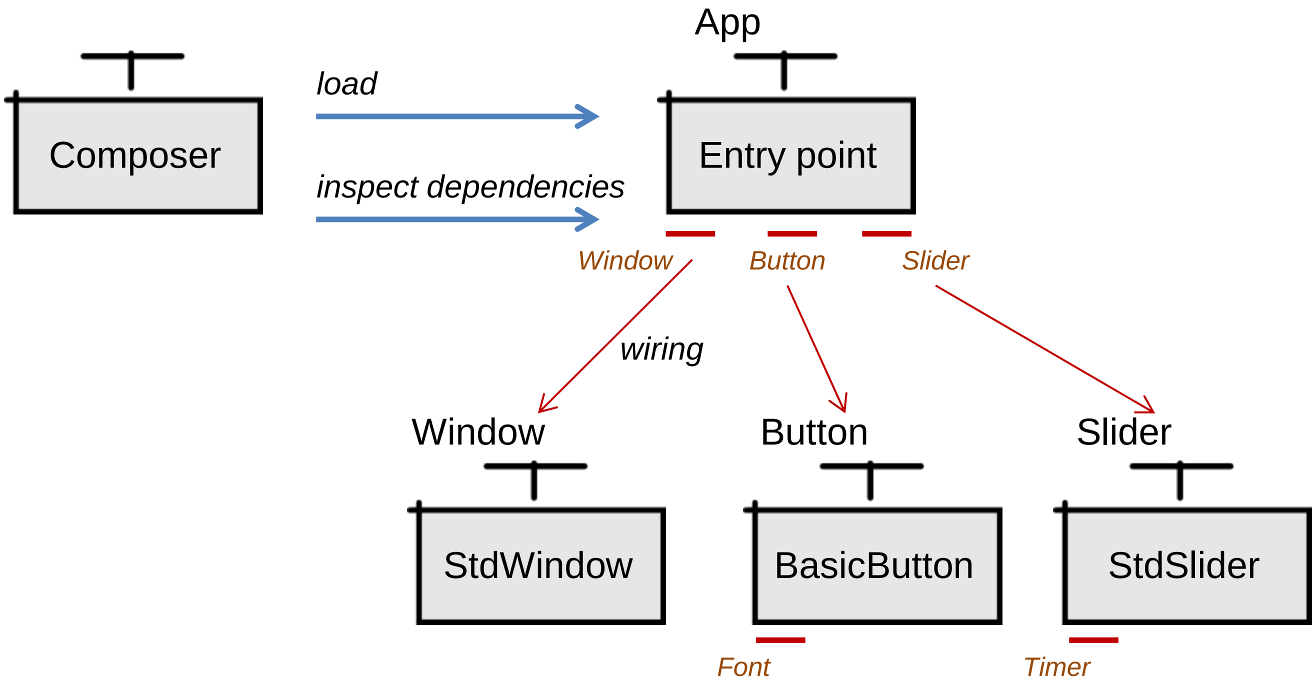
Example




```
interface Button extends ClickableObject{

    Button(char text[])

    void setText(char text[])

    char[] getText()
    }
```

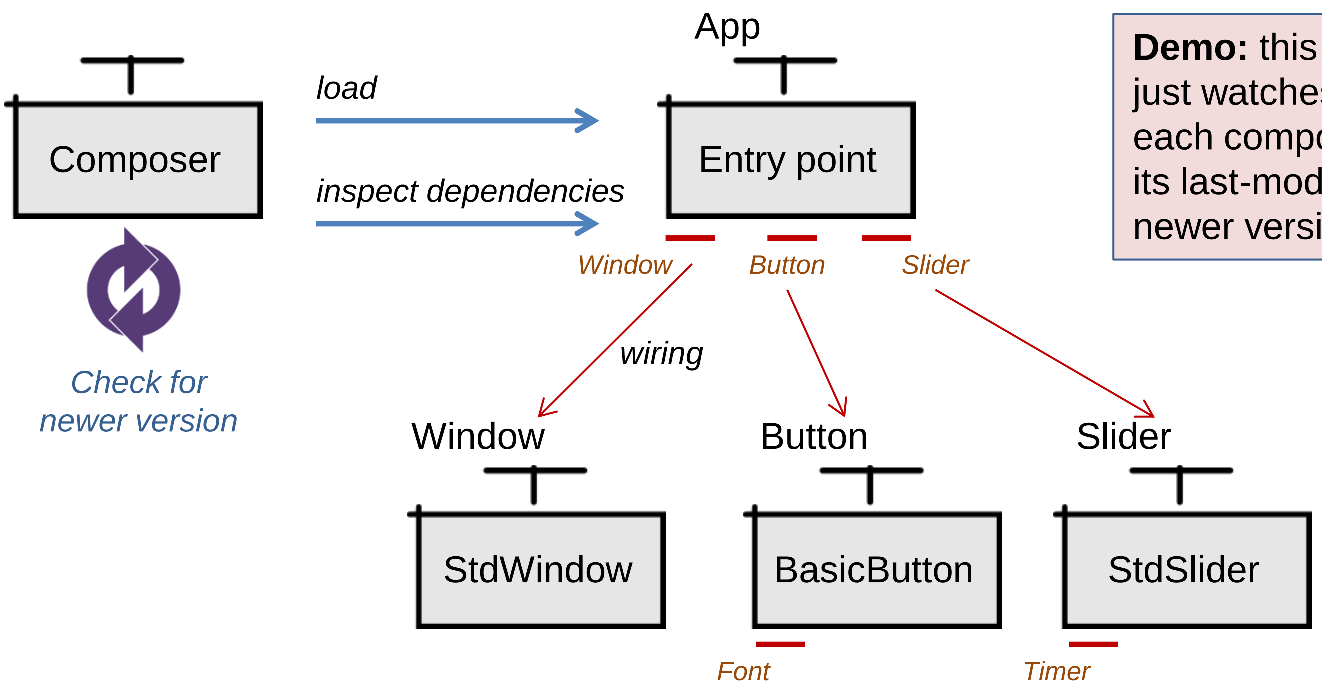
```
component provides App requires ui.Window, ui.Button, Slider {

    Window window
    Button b1
    Button b2
    Slider slider

    eventsink AppEvents(EventData ed)
        {
        if (ed.type == Button.[click] && ed.source === b1)
            {
            slider.setActive(true)
            }
            else if (ed.type == Button.[click] && ed.source === b2)
            {
            slider.setActive(false)
            }
            else if (ed.source === window && ed.type == Window.[close])
            {
            window.close()
            }
        }

    int App:main(AppParam params[])
        {
        window = new Window("MyWindow")
        window.setSize(250, 180)
        window.setVisible(true)

        b1 = new Button("Start")
        b2 = new Button("Stop")

        b1.setPosition(10, 30)
        b2.setPosition(100, 30)

        slider = new Slider()
        slider.setPosition(10, 100)
        slider.setSize(230, 30)

        window.addObject(b1)
        window.addObject(b2)

        window.addObject(slider)
```

Example



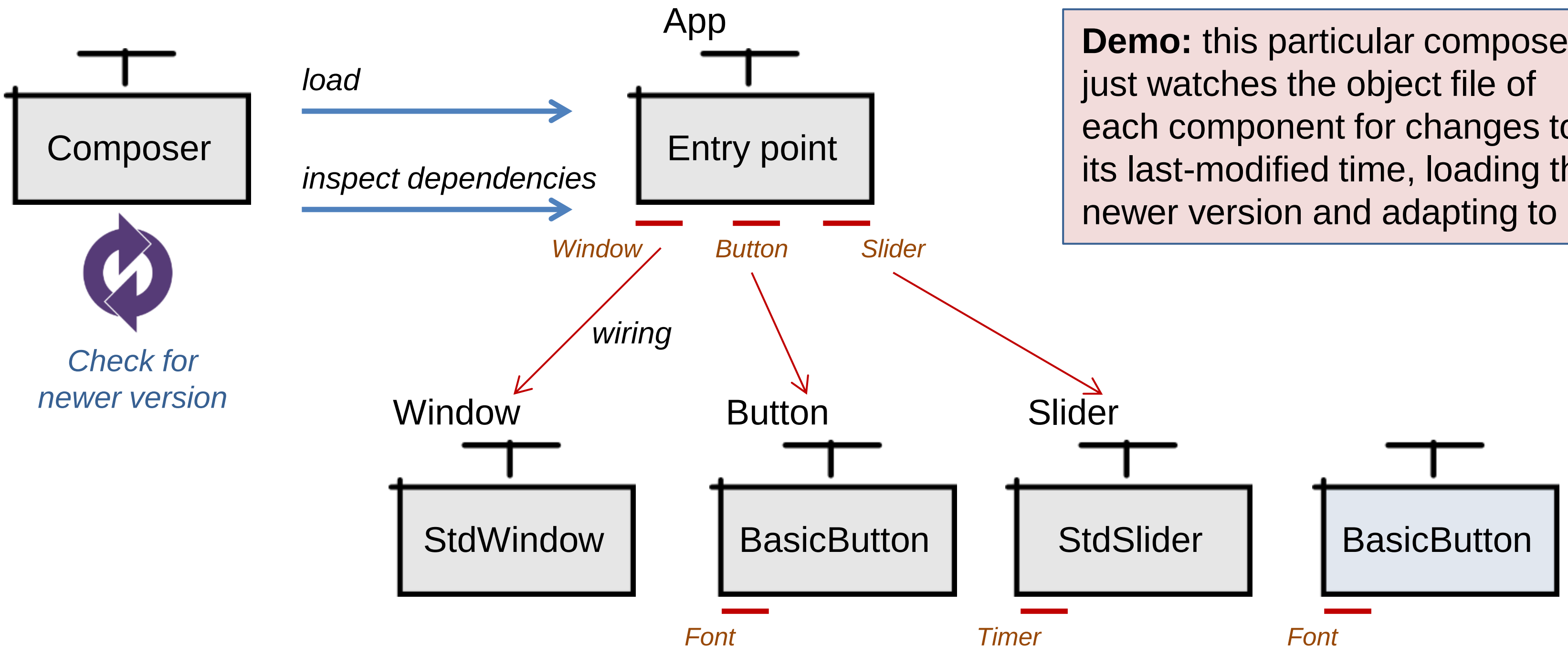
App
Composer
load
inspect dependencies
Entry point
Window
Button
Slider
wiring
Window
Button
Slider
StdWindow
BasicButton
StdSlider
Font
Timer

Example



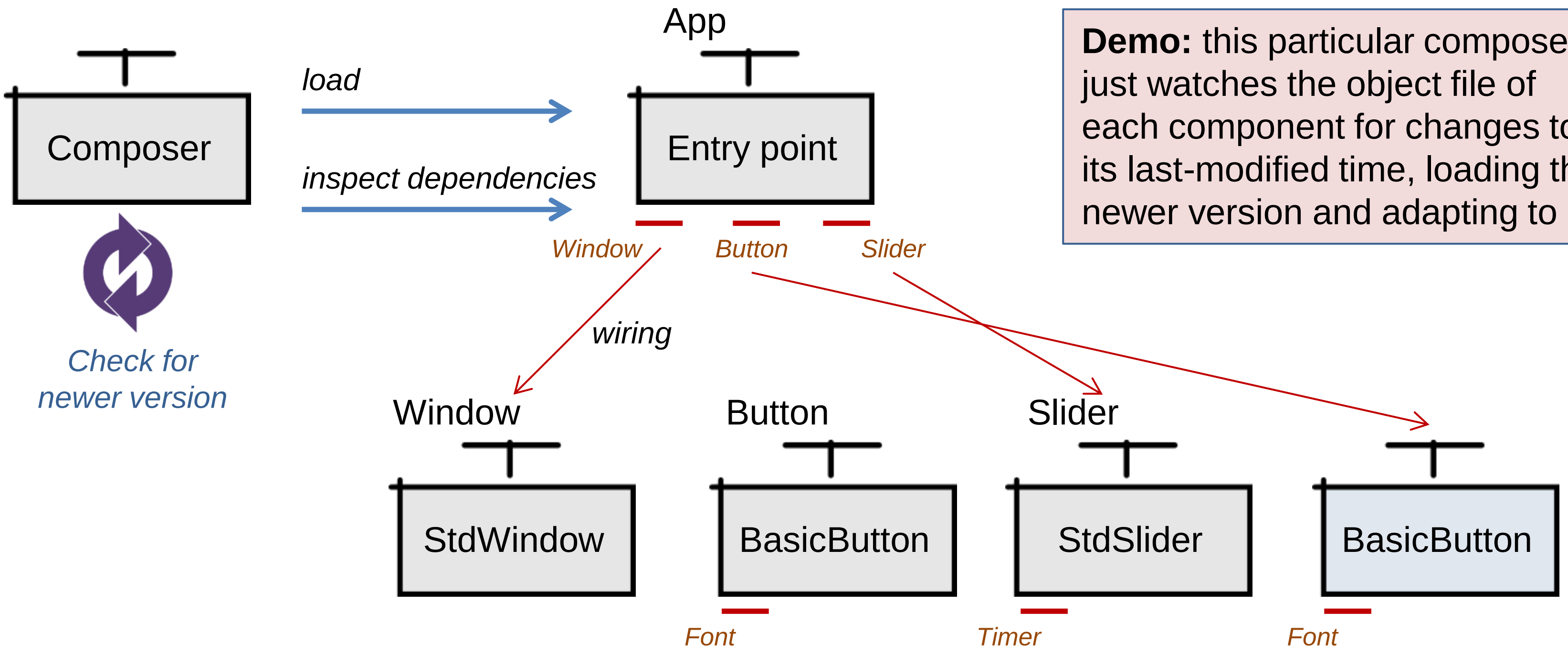
Demo: this particular composer just watches the object file of each component for changes to its last-modified time, loading the newer version and adapting to it

Example



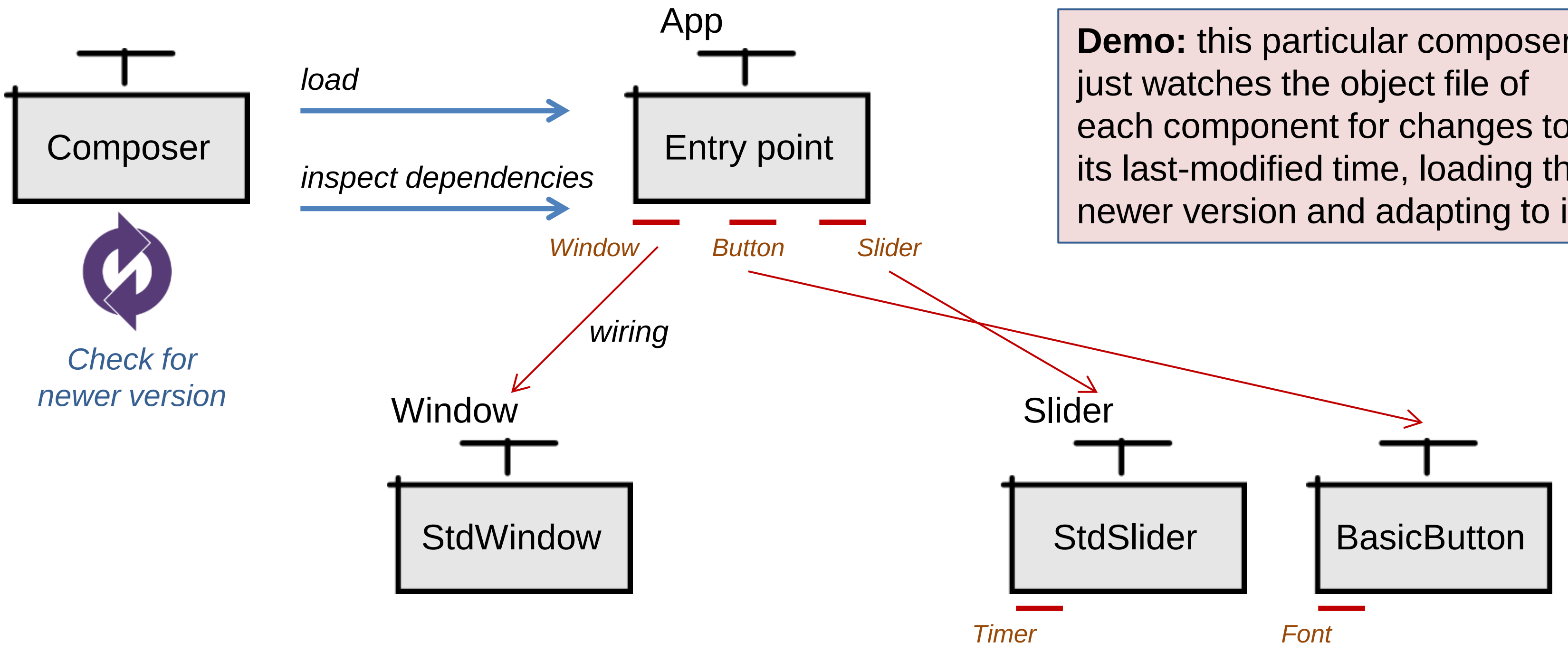
Demo: this particular composer just watches the object file of each component for changes to its last-modified time, loading the newer version and adapting to it

Example



Demo: this particular composer just watches the object file of each component for changes to its last-modified time, loading the newer version and adapting to it

Example



Demo: this particular composer just watches the object file of each component for changes to its last-modified time, loading the newer version and adapting to it

Summary

- Automated component search, and programmatic composition, is the next level of automation
- You can now build a *completely generic* composer, for any system, which locates components and decides how to compose them
- But we’re not done with automation yet…

Practical Assignment

- Using component search to build an example generic composer, applicable to any system

- For work sheet and code (plus lecture slides) go to: https://github.com/barryfp/saso2019tutorial