EMERGENT SOFTWARE SYSTEMS

Summer School

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

Funded by The Royal Society Newton Fund

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

THE ROYAL SOCIETY

Course Overview (weeks 1 and 2)

Week 1

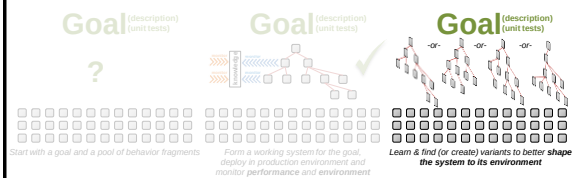
Monday	• Overview • Component models • Dana programming
Tuesday	• Autonomic Computing • Emergent Software
Wednesday	• Assembly, Perception and Learning • Distributed Emergent Software
Thursday	• Reinforcement Learning • Multi-Armed Bandits
Friday	• Project Creation Workshop

Week 2

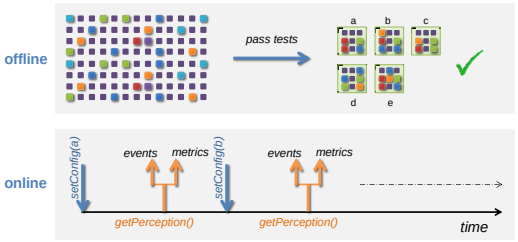
Monday	• Project Work
Tuesday	
Wednesday	
Thursday	• Presentations
Friday	

EMERGENT SOFTWARE

Emergent Software



Emergent Software



Emergent Software

- This implies some technological requirements:
- Modules of code that we can independently load and unload from main memory
- Each module should be self-describing of its **capabilities** and its **dependencies** on other modules
- We should be able to **hot-swap** modules (replace one for another) seamlessly at runtime; this should be **fast** enough to do regularly to allow cheap continuous learning of behaviours

COMPONENT MODELS

Component Models for Software

- A component model separates **program logic** from **program structure** (i.e., how logic is *composed*)
- The idea first became prominent in work by Douglas McIlroy in 1968, “Mass Produced Software Components”
- Since then, various flavours of the general idea have been proposed and implemented

Comparisons to Object Orientation

```
import java.xml.DocLoader;

public class MyClass extends Athena
    {
    MyClass()
          {
          DocLoader q = new DocLoader();
          ...
          }
    }
```

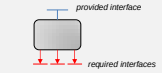
Classes refer directly to **other specific classes** in the world, where class = **implementation**

Once compiled, this class is **statically** bound to link against these specific implementations

We can say here that *structure* is thus **entangled** with *logic*

> Although Java programs appear to be well-encapsulated and "dynamic", in structural terms a compiled program is effectively a monolith – and may as well be a single binary.

Characteristics of a Component Model

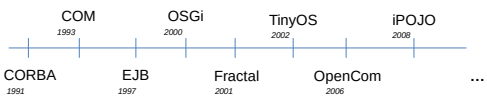


Components advertise *provided* and *required* interfaces, where an interface is a collection of function prototypes

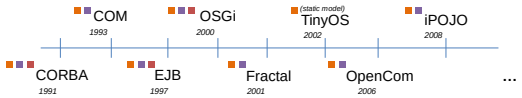
Required interfaces are wired to type-compatible provided interfaces by a **composer**, which is separate from the logic of the system

Components are strongly encapsulated, so that they must only interact via interfaces (there are no side channels such as shared memory)

Component Models Through Time

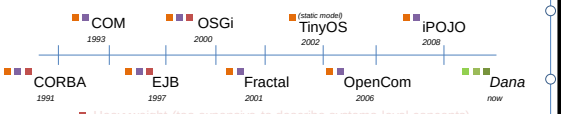


Component Models Through Time



- Heavyweight (too expensive to describe systems-level concepts)
- Frameworks (many things cannot be expressed as components)
- Quiescence (stop-the-world approach to runtime adaptation)

Component Models Through Time



- Heavyweight (too expensive to describe systems-level concepts)
  - Lightweight (can efficiently describe systems-level concepts)
- Frameworks (many things cannot be expressed as components)
  - Language (everything is inherently expressed as a component)
- Quiescence (stop-the-world approach to runtime adaptation)
  - Interposition (runtime adaptation is seamless and fast; ~10µs)

THE DANA LANGUAGE

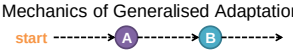
Theory

Core Concepts - Theory

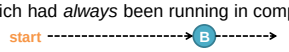
- Dana is a **general-purpose** systems building language which is based on the component paradigm
- We have extended and refined the component paradigm to support all design patterns found in modern systems
- Dana has its own compiler and a custom-built interpreter, designed to support fully generalised runtime adaptation

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation



- Aim of perfect runtime adaptation is to move from a system in one composition of **logic A** to another composition of **logic B**, so that the running system in **B** is indistinguishable from a version which had *always* been running in composition **B**



- This is very hard to achieve in general for stateful systems; Dana satisfies the aim for the *structural* elements of a system

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation
- The design of Dana as a language is built from the ground up around making every element of a system adaptable at runtime, so that adaptation is seamless, fast and cheap
- All other elements of the language (type system, syntax, etc.) follow from this fundamental design goal

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

```
interface Tokeniser {
    ParseToken[] tokenise(char str[], String tokens[])
    }
```

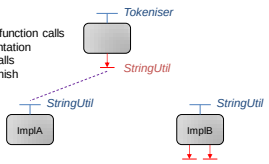

```
interface StringUtil {
    char[] subString(char str[], int start, int len)
    char[] trim(char str[])
    String[] explode(char str[], String tokens[])
    }
```

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish



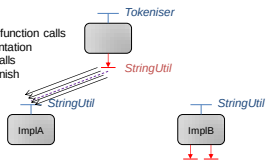
Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish

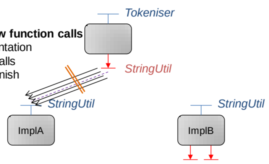


Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation
  - *Components and Interfaces*

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish

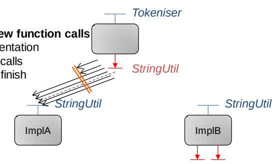


Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation
  - *Components and Interfaces*

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish

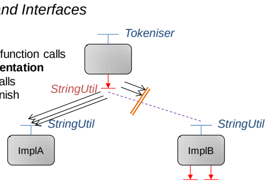


Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation
  - *Components and Interfaces*

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish

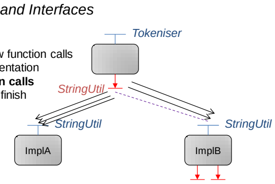


Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation
  - *Components and Interfaces*

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish



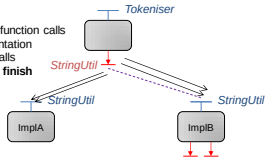
Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish



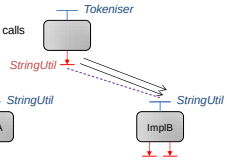
Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

1. Pause wiring; hold new function calls
2. Rewire to new implementation
3. Resume held function calls
4. Wait for active calls to finish



Adaptation using dynamic interposition

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

What about this:

```
File fd = new File("stuff.txt")

byte b[] = fd.read(512)

fd.close()

HashTable ht = new HashTable()

ht.put("alpha", sdn)
ht.put("beta", q)
...
qv = ht.get("alpha")

Window w = new Window("My App")

Button b = new Button("Go!")

w.add(b)
```


```
interface Tokeniser {
    ParseToken[] tokenise(char str[], String tokens[])
    }
```

```
interface StringUtil {
    char[] subString(char str[], int start, int len)
    char[] trim(char str[])
    String[] explode(char str[], String tokens[])
    }
```

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References

```
Panel p = new Panel()

Button backButton = new Button("back")
backButton.setPosition(20, 400)

p.add(backButton)
```


```
interface FileBrowser {
    ParseToken[] tokenise(char str[], String tokens[])
    }
```

```
interface Panel extends GraphicsObject {
    Panel()
    void add(GraphicsObject g)
    }
```

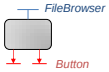
```
interface Button extends GraphicsObject {
    Button(char name[])
    void setColor(Color c)
    }
```

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References

```
Panel p = new Panel()

Button backButton = new Button("back")
backButton.setPosition(20, 400)

p.add(backButton)
```



When an interface needs state, we declare it as **transfer state** within the interface; this state is available to a new adapted implementation.
> This may be a generic/abstract representation of the implementation's actual internal state, which may be translated to/from the abstract transfer state format as part of adaptation.

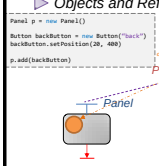
```
interface Panel extends GraphicsObject {
  transfer GraphicsObject objects[]

  Panel()
  void add(GraphicsObject g)
  }
```

```
interface Button extends GraphicsObject {
  transfer char name[]
  transfer Color c

  Button(char name[])
  void setColor(Color c)
  }
```

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

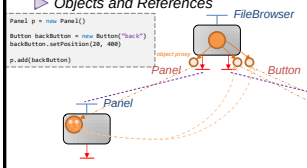
```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

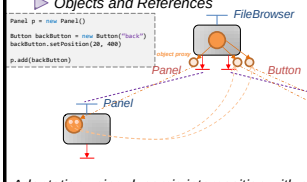
```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



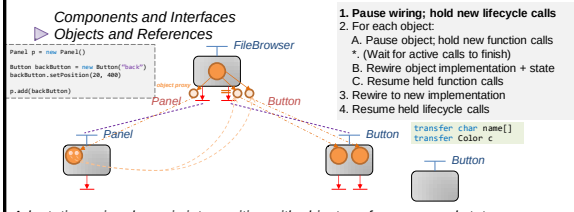
1. Pause wiring; hold new lifecycle calls
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

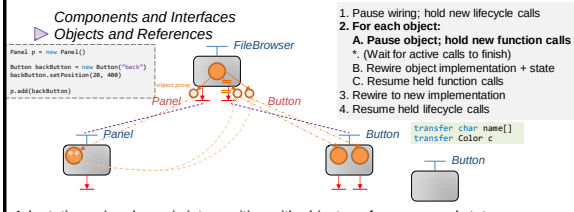


1. **Pause wiring; hold new lifecycle calls**
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

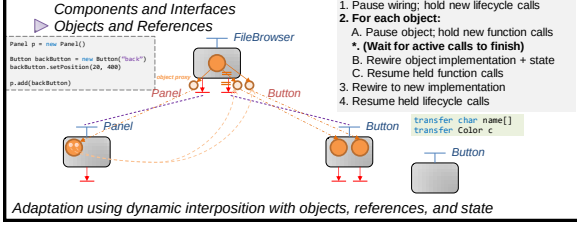


1. Pause wiring; hold new lifecycle calls
2. **For each object:**
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

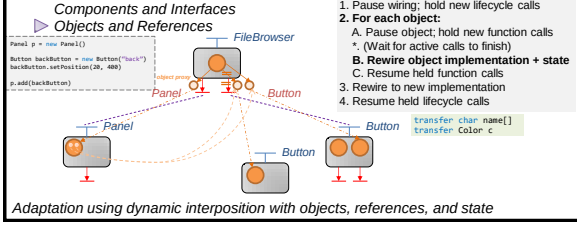


1. Pause wiring; hold new lifecycle calls
2. **For each object:**
   A. Pause object; hold new function calls
   ***. (Wait for active calls to finish)**
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation



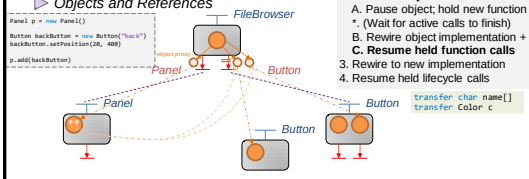
1. Pause wiring; hold new lifecycle calls
2. **For each object:**
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. **For each object:**
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

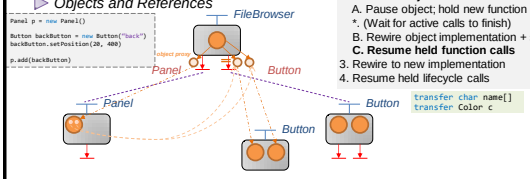
```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. **For each object:**
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. Resume held lifecycle calls

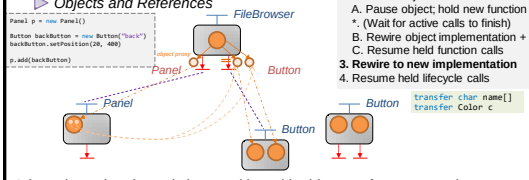
```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. **Rewire to new implementation**
4. Resume held lifecycle calls

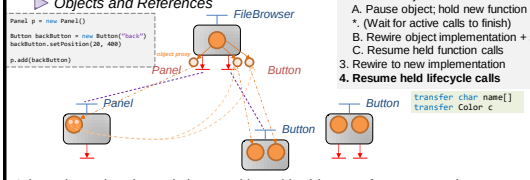
```
transfer char name[]
transfer Color c
```

Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References



1. Pause wiring; hold new lifecycle calls
2. For each object:
   A. Pause object; hold new function calls
   *. (Wait for active calls to finish)
   B. Rewire object implementation + state
   C. Resume held function calls
3. Rewire to new implementation
4. **Resume held lifecycle calls**

```
transfer char name[]
transfer Color c
```

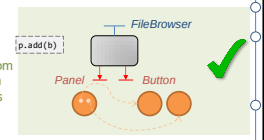
Adaptation using dynamic interposition with objects, references, and state

Core Concepts - Theory

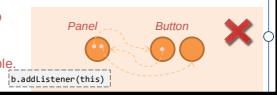
- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References

A 'parent' component can instantiate objects from its dependencies and pass references between them; we can adapt any of the implementations and perfect adaptation is upheld.



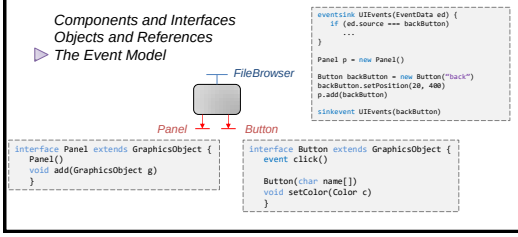
An object **cannot** hand out *self-references* to other objects (the Dana language has no concept of a self-reference), as this creates cases in which perfect adaptation is impossible.



Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References
The Event Model



Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References
The Event Model
Data and References

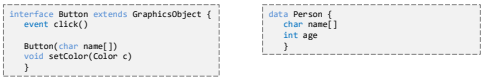
Because all **behaviour** (via objects) is represented though an interface-implementation pair, we want a convenient way to represent **pure data**

We do this via a **data type**, which has a set of member fields (but has no functions). This gives us two type hierarchies: **interfaces** if you want to have functions/behaviour; **data** if you just want member fields with no behaviour.

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References
The Event Model
Data and References



Data instances have a concept of *ownership* and *write permissions*. The instantiator (owner) is permitted to write to fields; other objects holding a reference may only read. This prevents a shared memory side-channel for objects to communicate over, which we could not properly track under our safe runtime adaptation protocol.

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References
The Event Model
▷ *Data and References*

```
interface Button extends GraphicsObject {
   event click()

   Button(char name[])
   void setColor(Color c)
   }
```

```
data Person {
   char name[]
   int age
   }
```

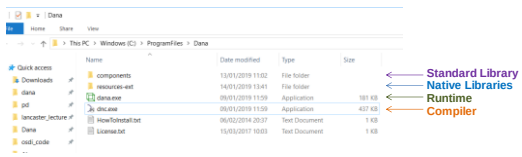
All together these ideas create a kind of hybrid programming model in Dana: a lot of code is "functional" and only uses data to pass between functions; where needed, some code uses objects with internal state / state machines.

THE DANA LANGUAGE

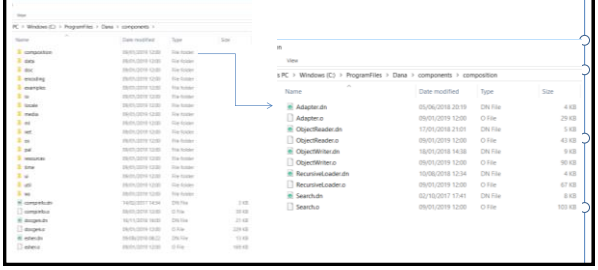
Practice

Core Concepts - Practice

Core Concepts - Practice



Core Concepts - Practice



Core Concepts - Practice

Core Concepts – First Program

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

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

        return 0
        }

}
```

Core Concepts – First Program

```
data AppParam {
    char string[]
    }

interface App {
    int main(AppParam params[])
    }
```

in standard library root package

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

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

        return 0
        }

}
```

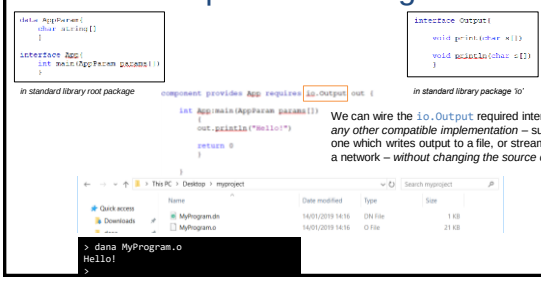
```
interface Output {

    void print(char s[])

    void println(char s[])
    }
```

in standard library package 'io'

Core Concepts – First Program

Core Concepts – First Program

Core Concepts – First Program



A multi-component program

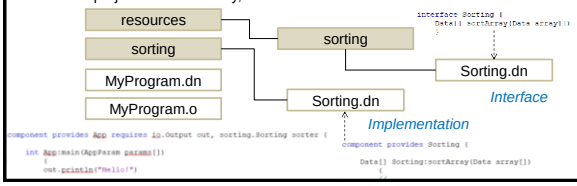
- Programs using multiple components are organised into two symmetric directory trees: **resources** and available implementations

A multi-component program

- Programs using multiple components are organised into two symmetric directory trees: **resources** and available implementations
- Let's imagine we're going to write a new re-usable component to sort an array, in a package *sorting*
- We create a new directory *resources* for the interface definition (and any other types), with a sub-directory *sorting* for our package

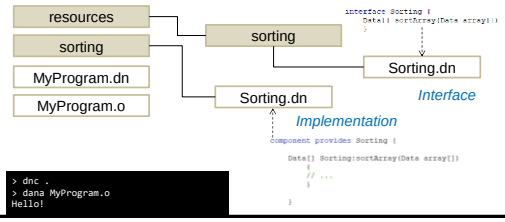
A multi-component program

- We create a new directory *resources* for the interface definition (and any other types), with a sub-directory *sorting* for our package
  - We also create a *default implementation* under a directory *sorting* within the project's root directory, with the same file name as the interface



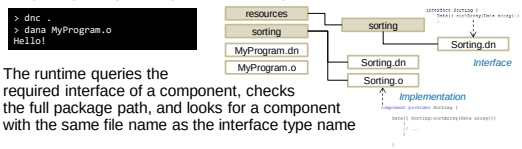
A multi-component program

- We can compile the entire system using the command "`dnc .`" and then run the App component as before



A multi-component program

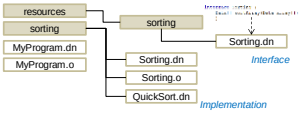
- The Dana runtime uses a *default name linking* approach when composing multiple components into a system



- The runtime queries the required interface of a component, checks the full package path, and looks for a component with the same file name as the interface type name
- If found, the runtime loads that component, wires to its provided interface, and does the same process on the newly loaded component

A multi-component program

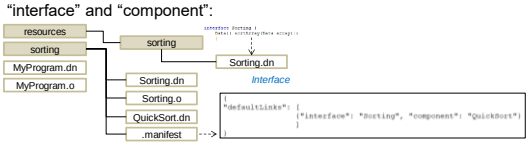
- Let's imagine that we now decide to create a new implementation variant of the `Sorting` interface, which uses a quick sort algorithm
- Because the interface is the same, we just add a new implementation to the directory of components:



- A meta-composer will use this alternative sorting algorithm if / when it likes, for example as a result of learning the runtime conditions in which it performs better

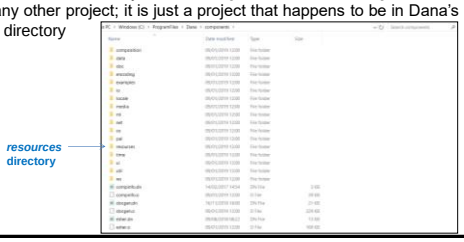
A multi-component program

- We can also change the default implementation of sorting to our new component, so that any composers using default linking rules (like the Dana runtime) will use the new implementation as a first choice
- This is done using `.manifest` files, which contain JSON pairs of "interface" and "component":



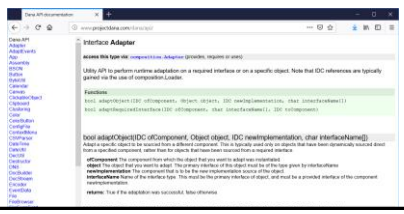
The standard library

- Dana's standard library uses **exactly the same directory structure** as any other project; it is just a project that happens to be in Dana's root directory



The standard library – documentation

- Dana's API documentation is generated from the source code in the standard library's resources directory:



Summary

- Dana is a cutting edge implementation of the component-oriented paradigm, embedded in a programming language that has been built from the ground up to be ultra-adaptive
- Code is highly reusable and can be composed together in highly flexible ways without needing to edit source code
- At runtime, every component in a system can be adapted safely and extremely quickly (in microseconds)
- Dana's standard library is completely open-source, including native libraries linking to OS functionality (written in C)

Practical Assignment

- Introductory Dana programming: we'll look at how to install Dana for yourself and create your first programs
- We also cover how linking works, manifest files, and getting familiar with the standard library
- For work sheet and code (plus lecture slides) go to: https://github.com/barryfp/summer_school