How to Build Emergent Software Systems

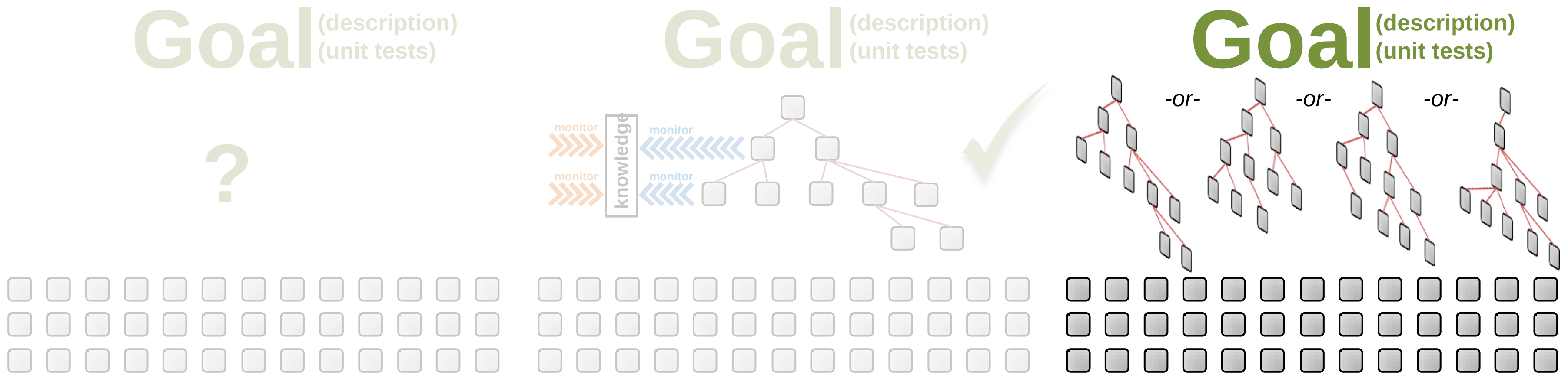
Tutorial

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

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

Emergent Software

Emergent Software

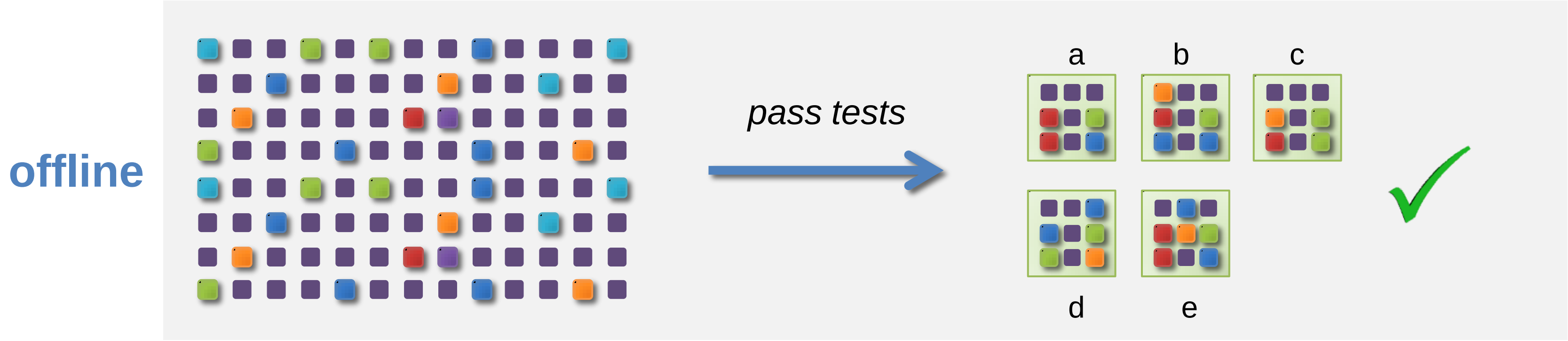


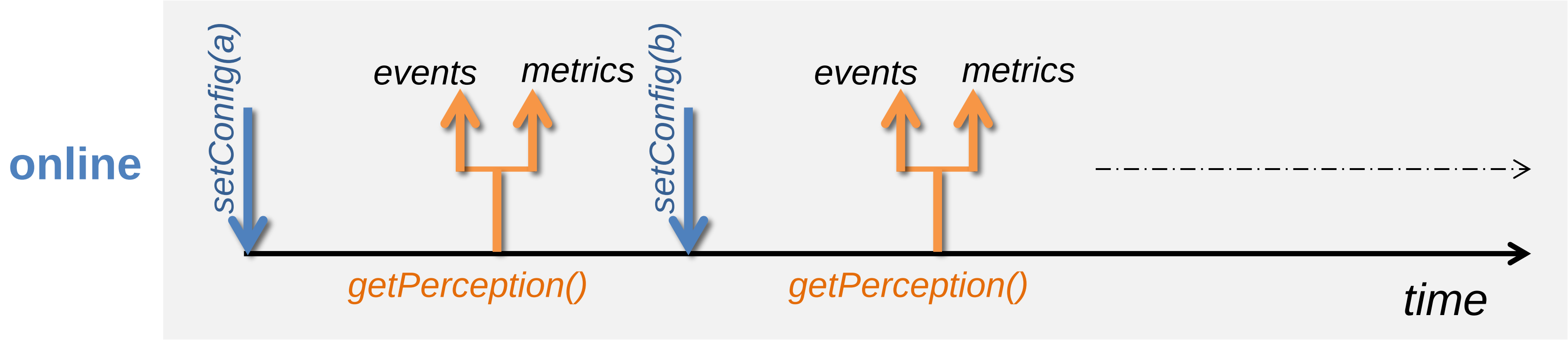
Start with a goal and a pool of behavior fragments

Form a working system for the goal, deploy in production environment and monitor ***performance*** *and* ***environment***

Learn & find (or create) variants to better ***shape the system to its environment***

Emergent Software





Tutorial Overview

- Self-Description
- Adaptation
- Component Search

Morning

- Compositional Reasoning
- Perception
- Learning

Afternoon

This tutorial is interactive ☺

- We have four sessions today

- Each one will start with some concepts and theory, and will end with a hands-on section to practice these ideas

- This is the best way to gain a deeper understanding of these ideas and how they might change the way you think

Self-Description & Runtime Adaptation

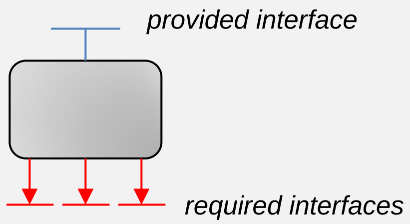
Programming Model

- We use a *component model* to separate **program logic** from **program structure** (i.e., how logic is *composed*)

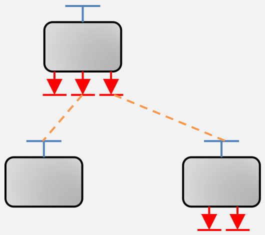
- This idea first became prominent in work by Douglas McIlroy in 1968, “Mass Produced Software Components”

- Since then, various flavours of the general idea have been proposed and implemented

Characteristics of a Component Model



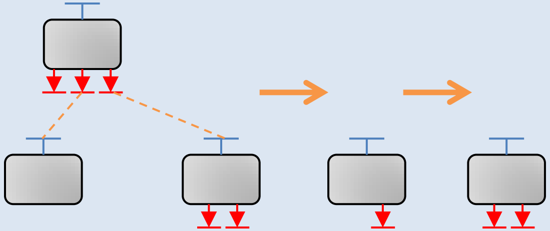
Components advertise *provided* and *required* interfaces, where an interface is a collection of function prototypes



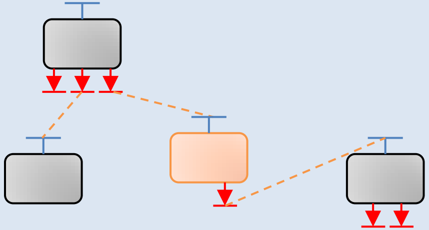
Required interfaces are wired to type-compatible provided interfaces **programmatically** by a **composer**, <u>separate</u> from the system's logic

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

Adaptation in a Component Model



Because we can programmatically control the wirings between components, we can later **change (adapt)** those wirings at runtime



We can also **inject probes** between two components to **monitor** aspects of the system, such as execution time of functions

Although we therefore have an adaptable technology in principle, the theory behind which kinds of things *can* really be adapted is complex

Component Models Through Time

CORBA *1991*

COM *1993*

EJB *1997*

OSGi *2000*

Fractal *2001*

TinyOS *2002*

OpenCom *2006*

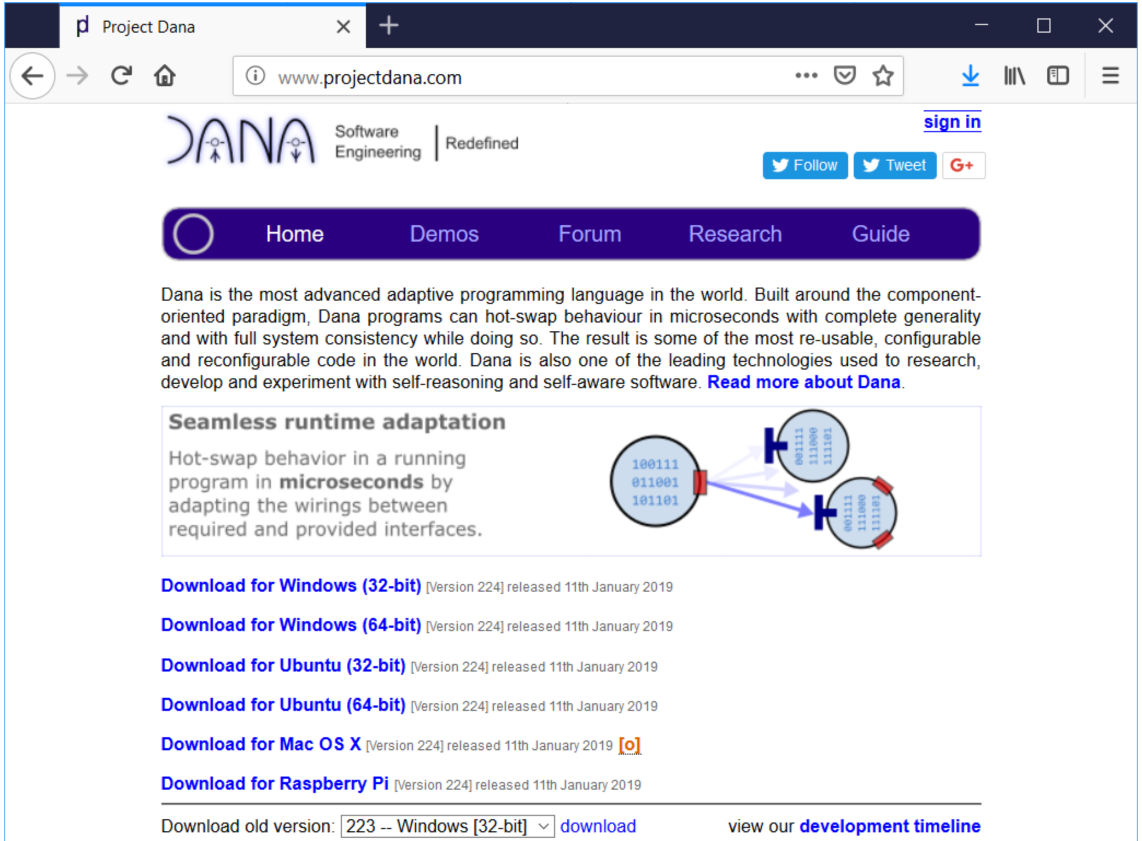
iPOJO *2008*

...

These approaches typically allow you to “play at the edges” of adaptation, by defining certain specific concepts as adaptable; the kinds of things which technically *can* be adapted is limited

From a systems perspective it’s interesting to explore a programming model which can capture *everything* under the same uniform, adaptable model, to offer *generalised* theories

Programming Model



The Dana Programming Model

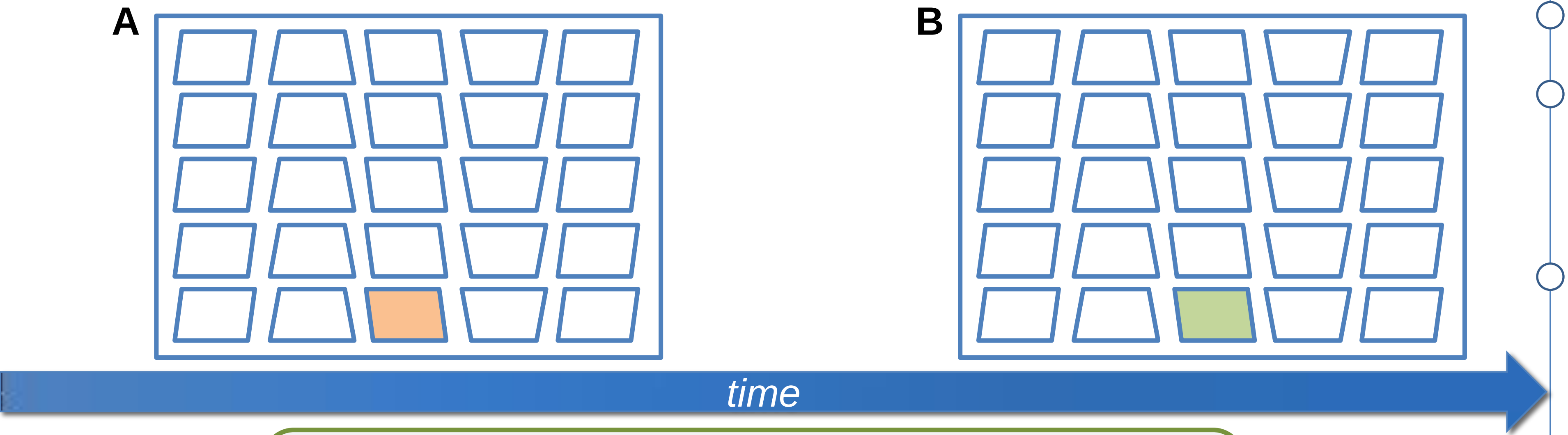
Core Concepts – Theory

- Dana is a **general-purpose** systems building language which is based on the component-oriented paradigm
- We have extended and refined this paradigm to support the complex set of design patterns found in full-stack systems, based on what can and can't be adapted
- Dana has its own compiler and a custom-built interpreter, designed to support fully generalised runtime adaptation

Core Concepts - Theory

For more, see: A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency"

- The Generalised Runtime Adaptation Problem

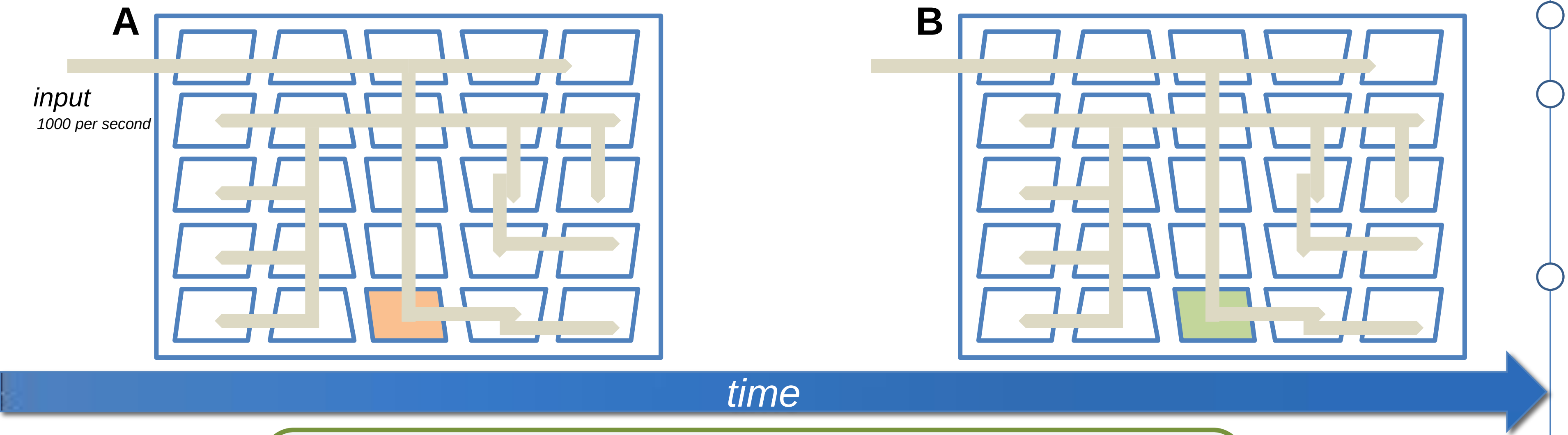


Perfect runtime adaptation is to move from a system in one composition of **logic A,** to another composition of **logic B**, so that the system in **B** is indistinguishable from one which had *always* been in composition **B**

Core Concepts - Theory

For more, see: A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency"

- The Generalised Runtime Adaptation Problem

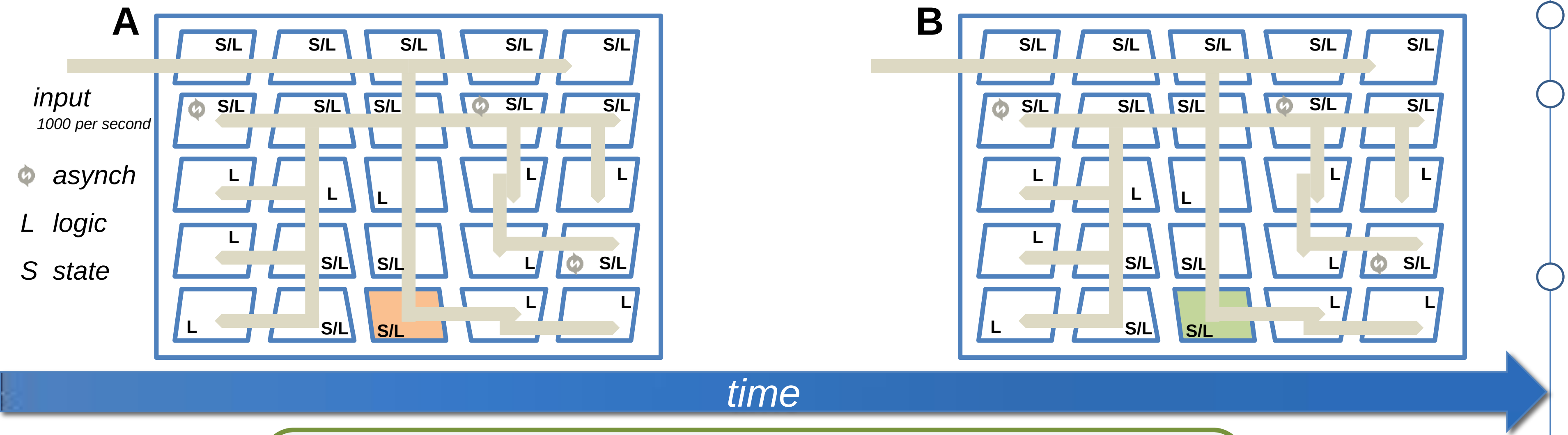


Perfect runtime adaptation is to move from a system in one composition of **logic A,** to another composition of **logic B**, so that the system in **B** is indistinguishable from one which had *always* been in composition **B**

Core Concepts - Theory

For more, see: A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency"

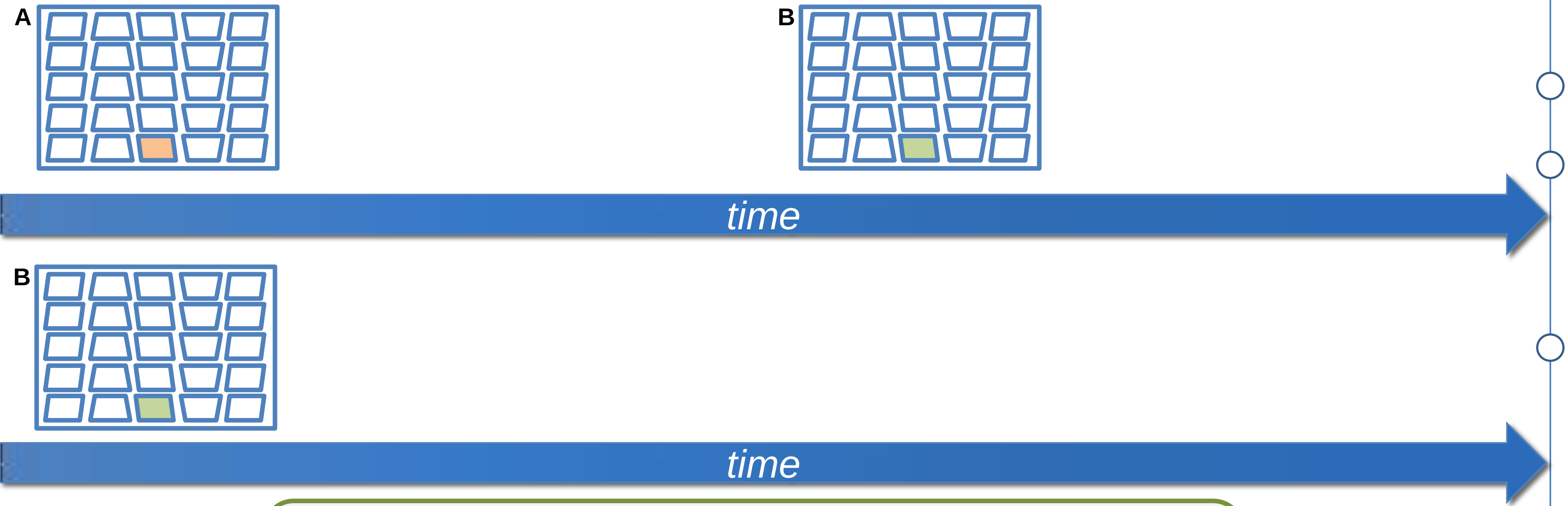
- The Generalised Runtime Adaptation Problem



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

Core Concepts - Theory

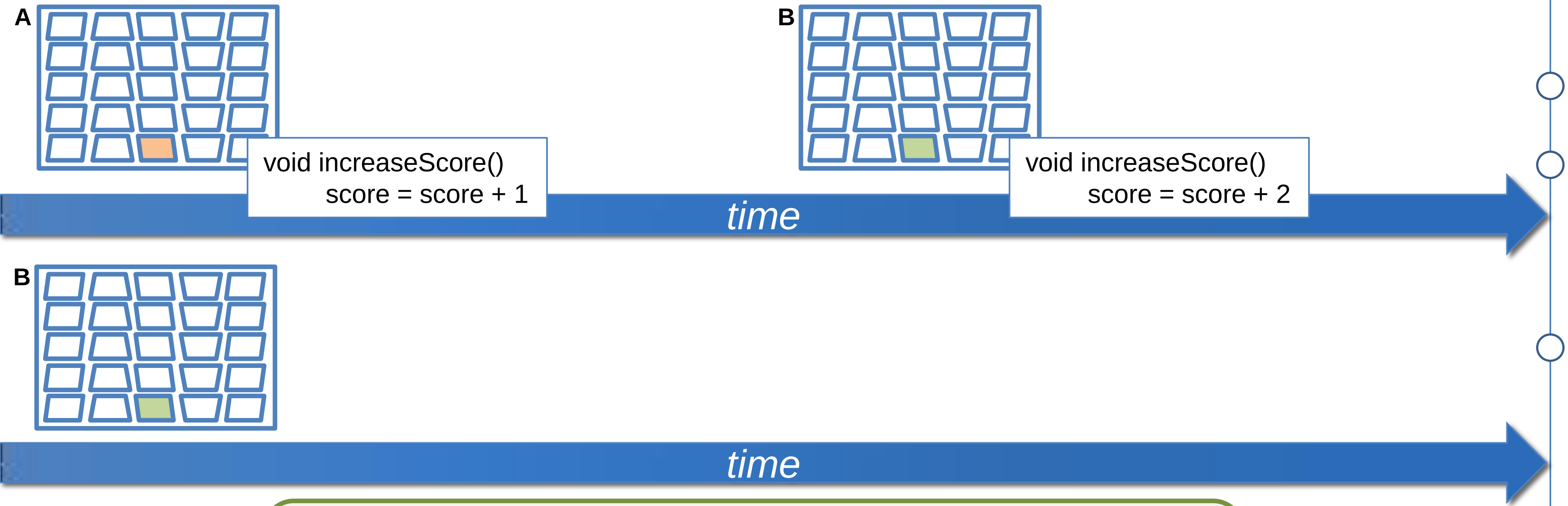
For more, see: A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency"



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

Core Concepts - Theory

For more, see: A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency"



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

Core Concepts - Theory

Guaranteeing this property is impossible in the general case, without extreme constraints on a programming model – constraints which would also prevent the expression of most serious systems

A *time* B

In Dana we uphold the perfect adaptation property at a structural level; we allow complex reference graphs of objects, needed to express full-stack systems concepts, and ensure from a *structural* perspective that adapting between A and B looks like the system had *always* been in B

B *time*

This effectively reduces the assurance of “safe adaptation” to verifying individual components against each other, rather than needing to understand and verify their effects on the broader system

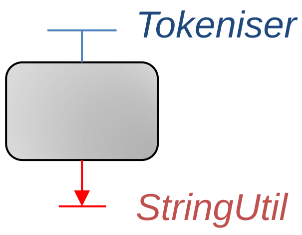
Core Concepts - Theory

Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

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



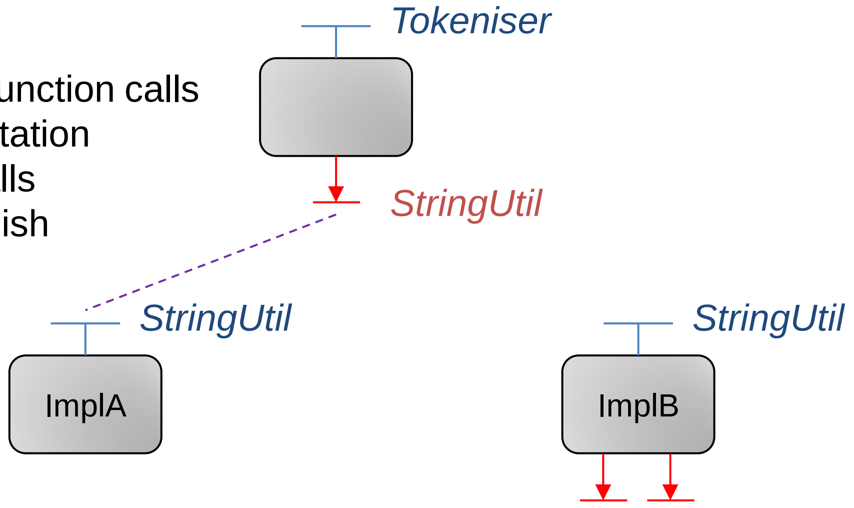

```
interface StringUtil {
   char[] subString(char str[], int start, int len)
   char[] trim(char str[])
   String[] explode(char str[], String tokens[])
   }
```

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



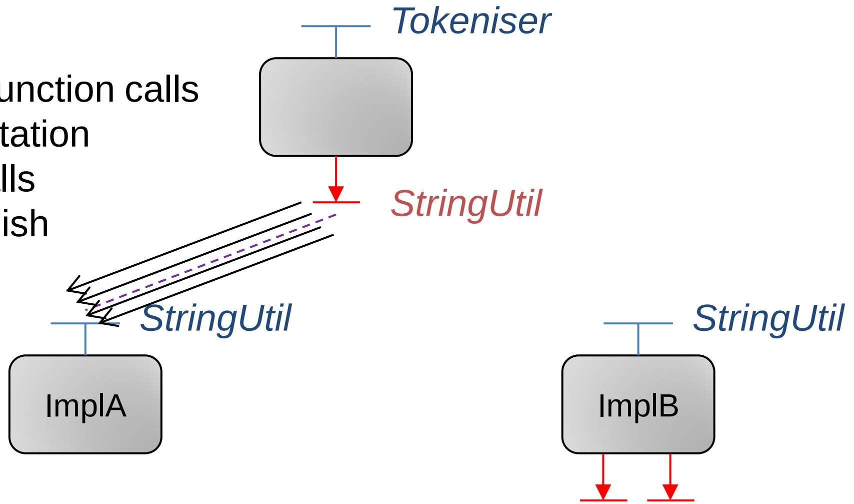
Adaptation using dynamic interposition

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



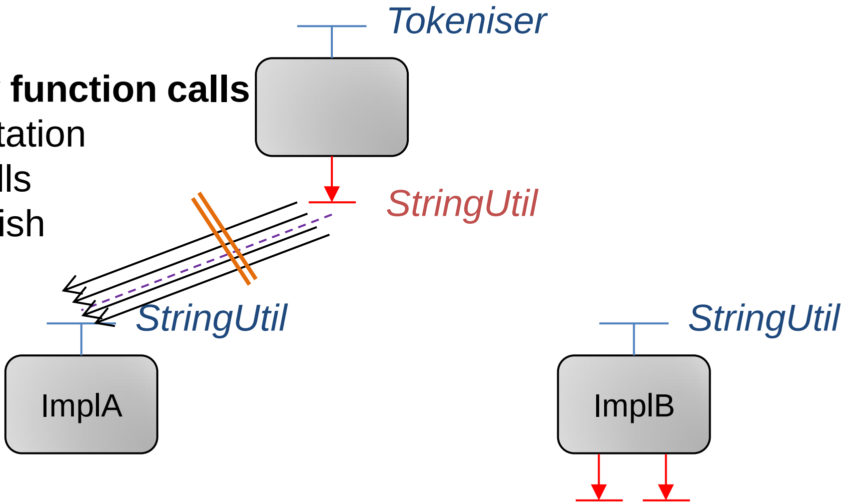
Adaptation using dynamic interposition

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



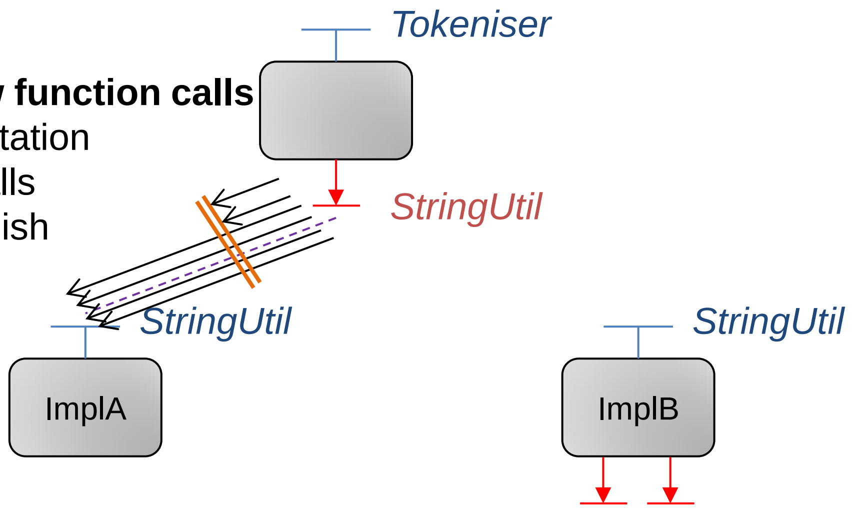
Adaptation using dynamic interposition

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



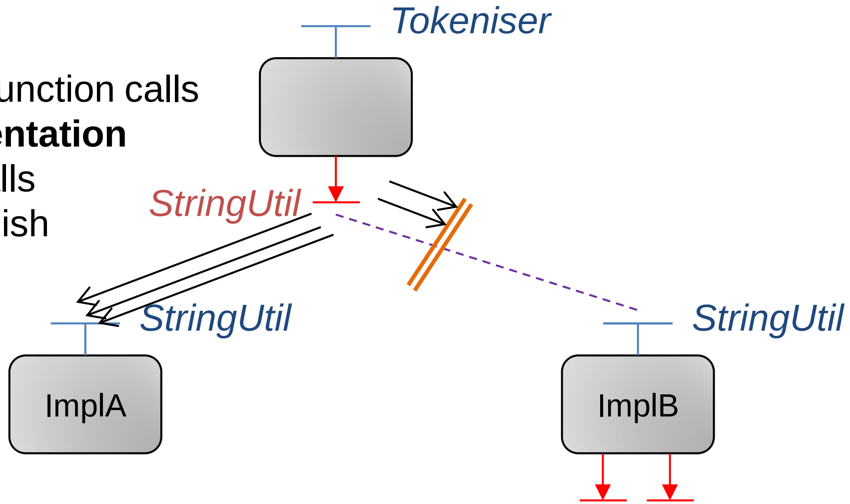
Adaptation using dynamic interposition

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



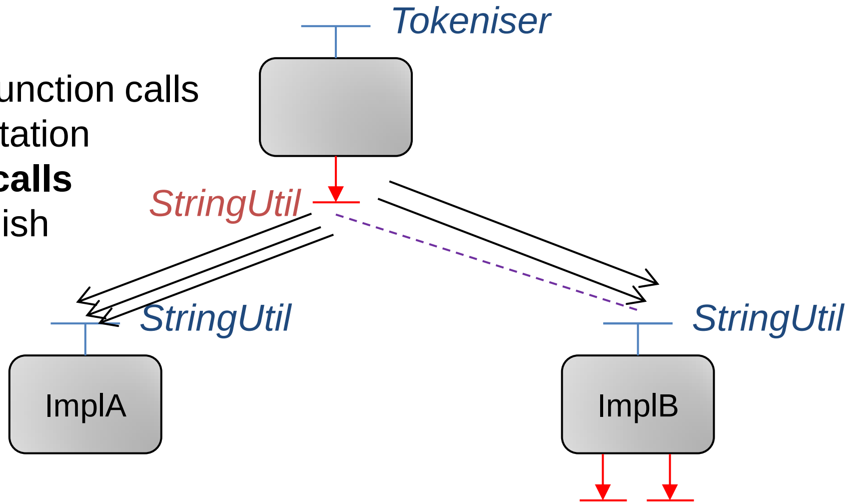
Adaptation using dynamic interposition

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



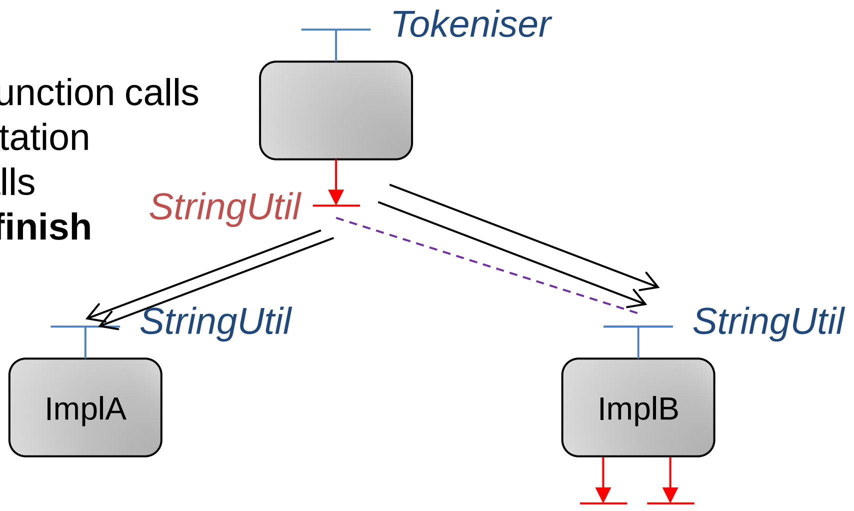
Adaptation using dynamic interposition

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**



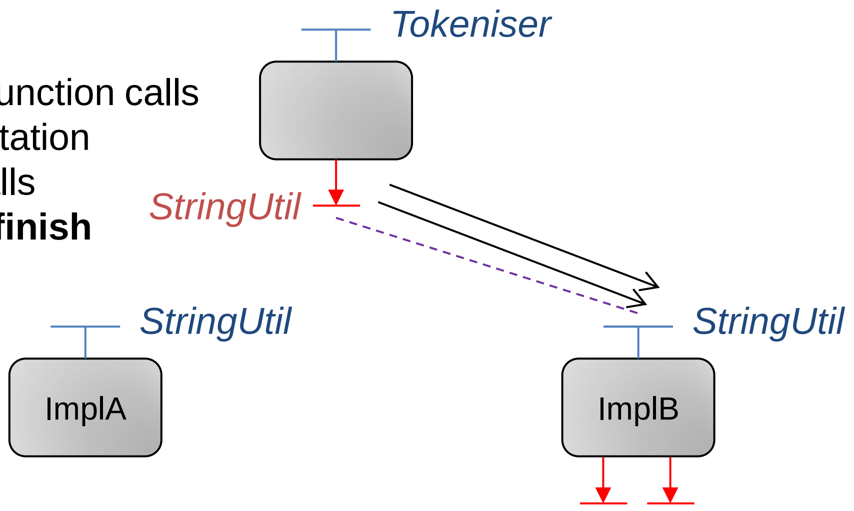
Adaptation using dynamic interposition

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

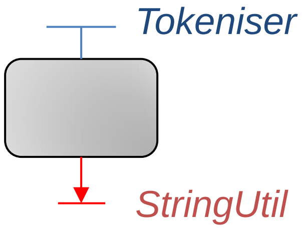
What about this:

```
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

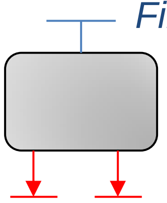
- Structural Mechanics of Generalised Adaptation

Components and Interfaces

▷ *Objects and References*

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

```
Panel p = new Panel()

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

p.add(backButton)
```




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

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

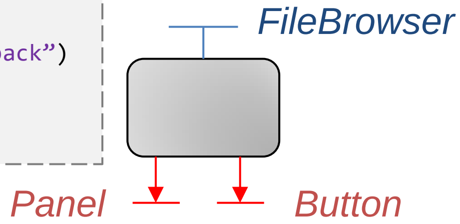
Core Concepts - Theory

- 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)
   }
```

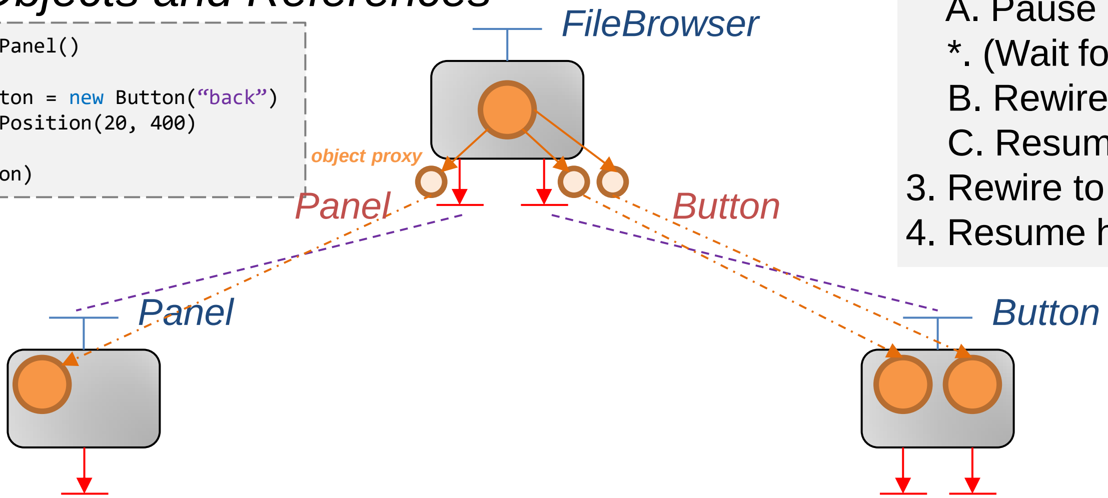
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

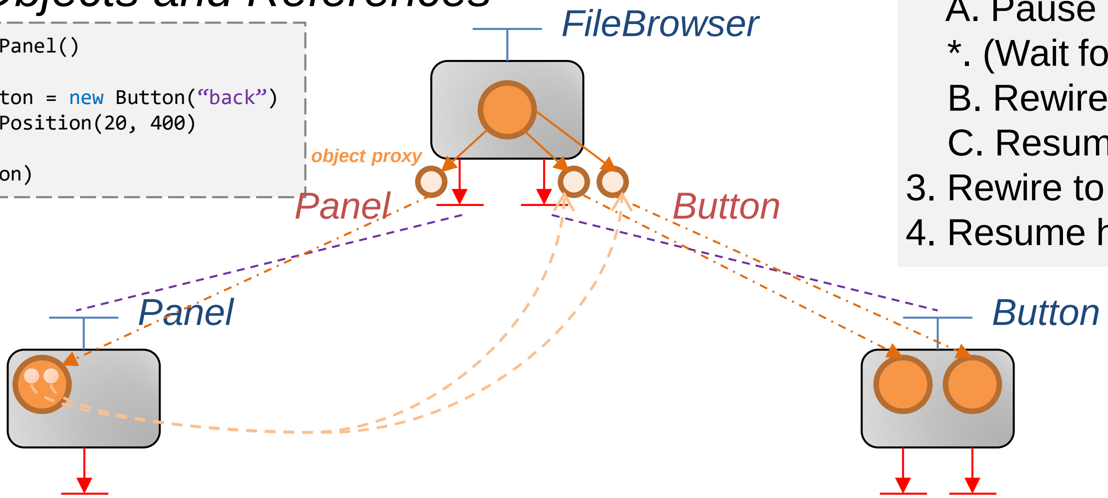
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

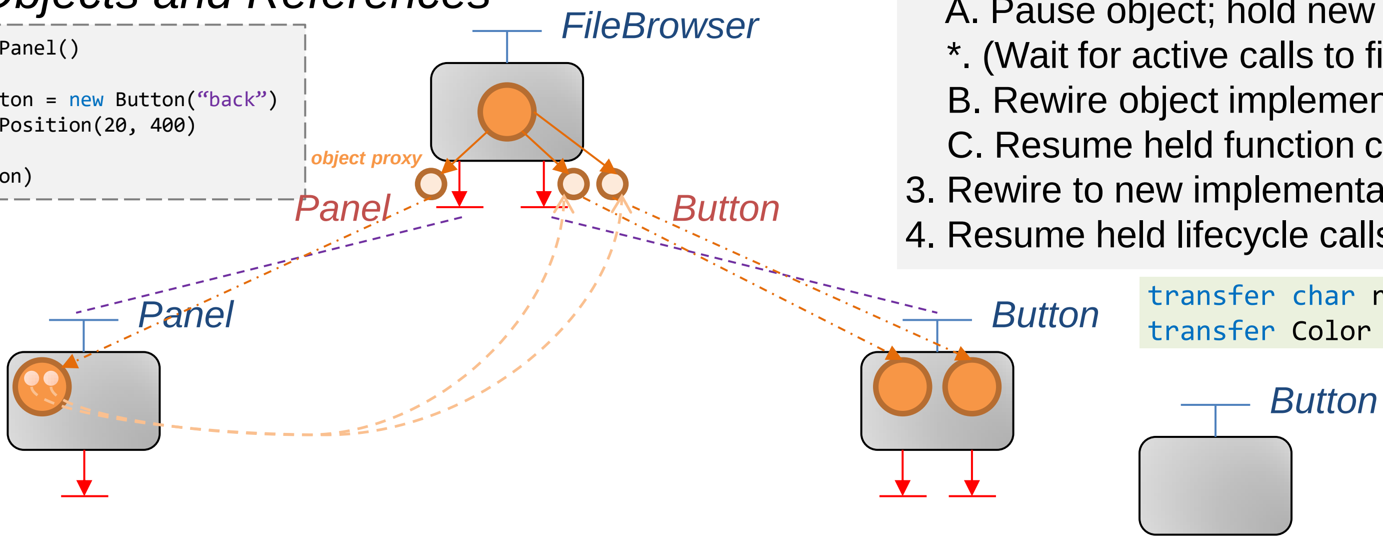
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

▷ *Objects and References*

```
Panel p = new Panel()

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

p.add(backButton)
```



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

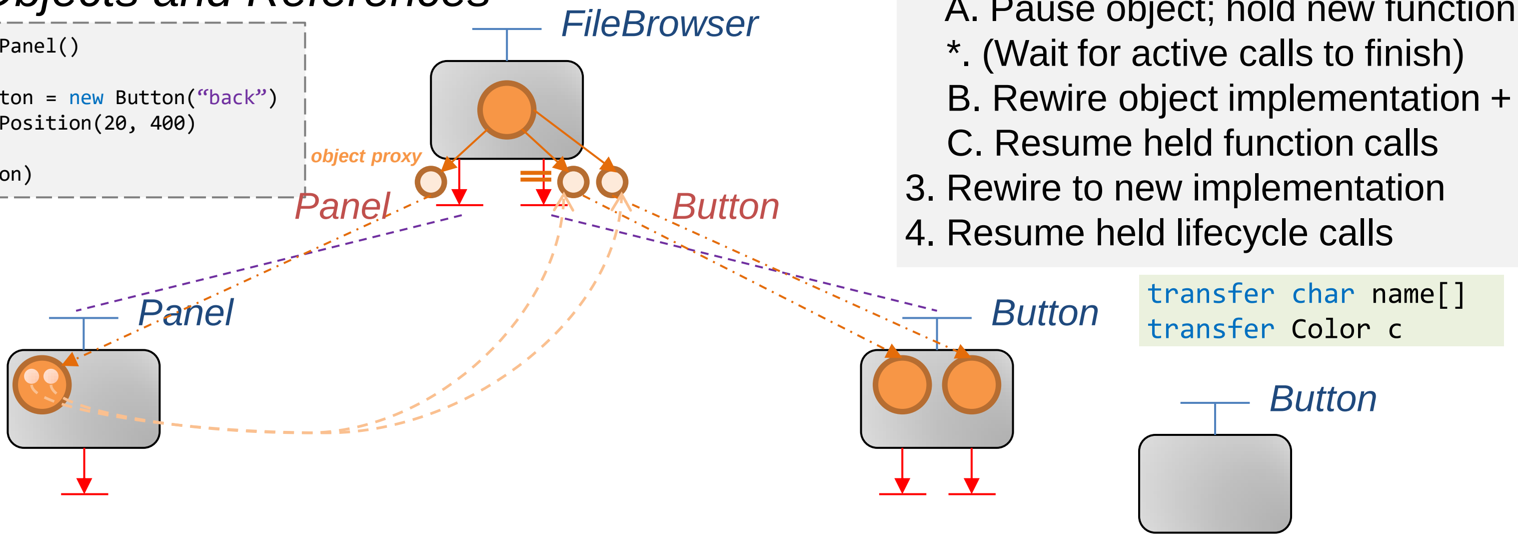
- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```

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

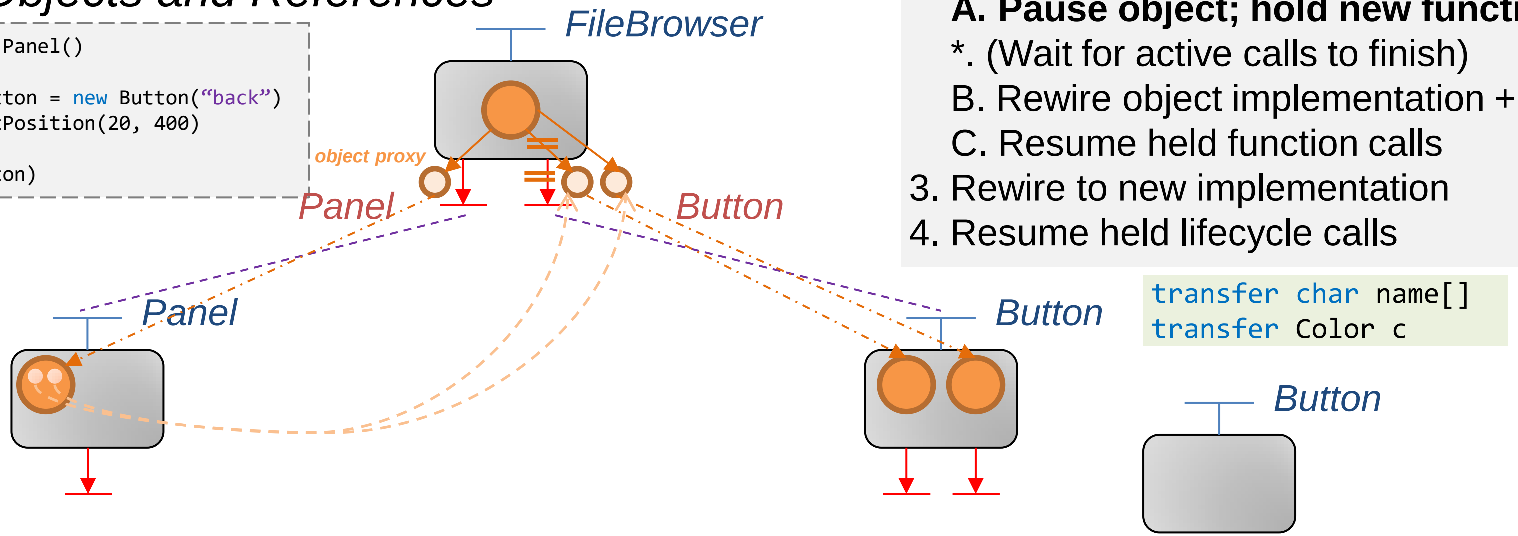
- Structural Mechanics of Generalised Adaptation

Components and Interfaces

▷ *Objects and References*

```
Panel p = new Panel()

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

p.add(backButton)
```

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

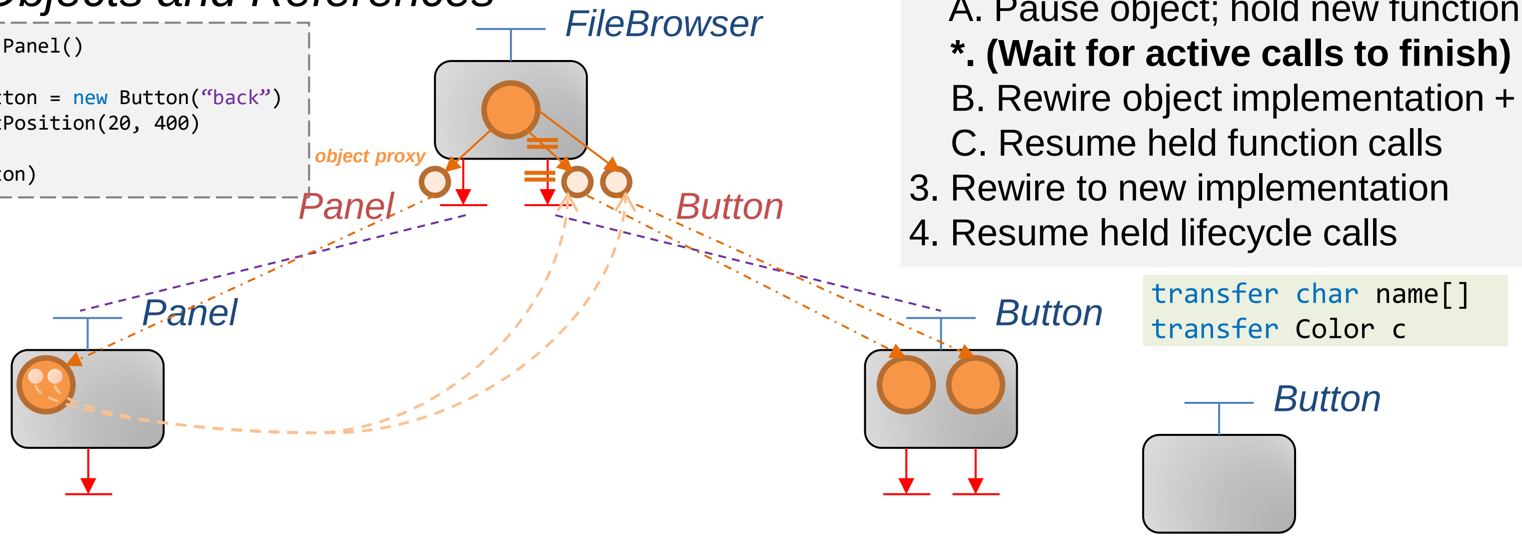
- Structural Mechanics of Generalised Adaptation

Components and Interfaces

▷ *Objects and References*

```
Panel p = new Panel()

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

p.add(backButton)
```

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

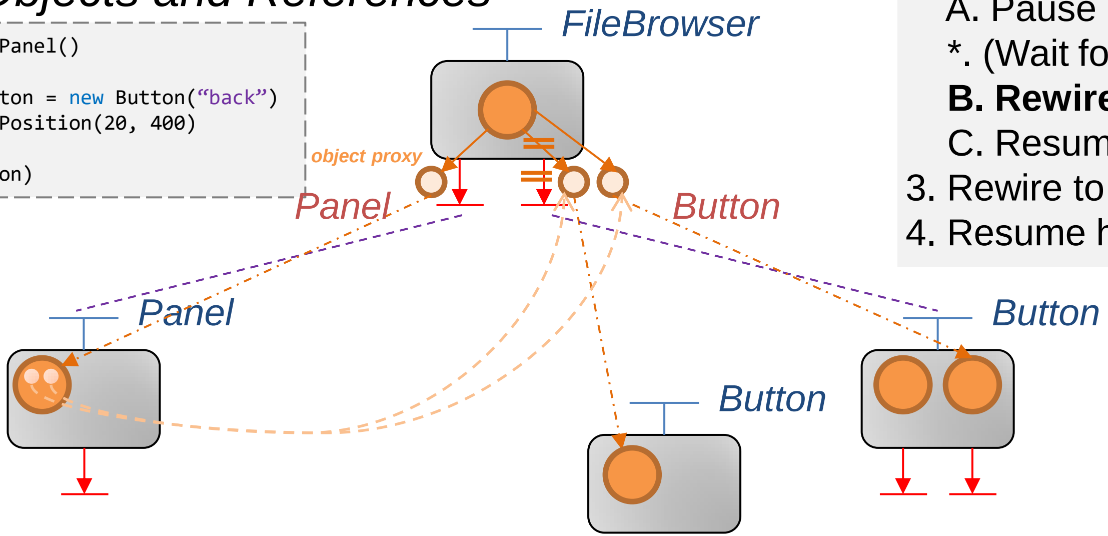
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

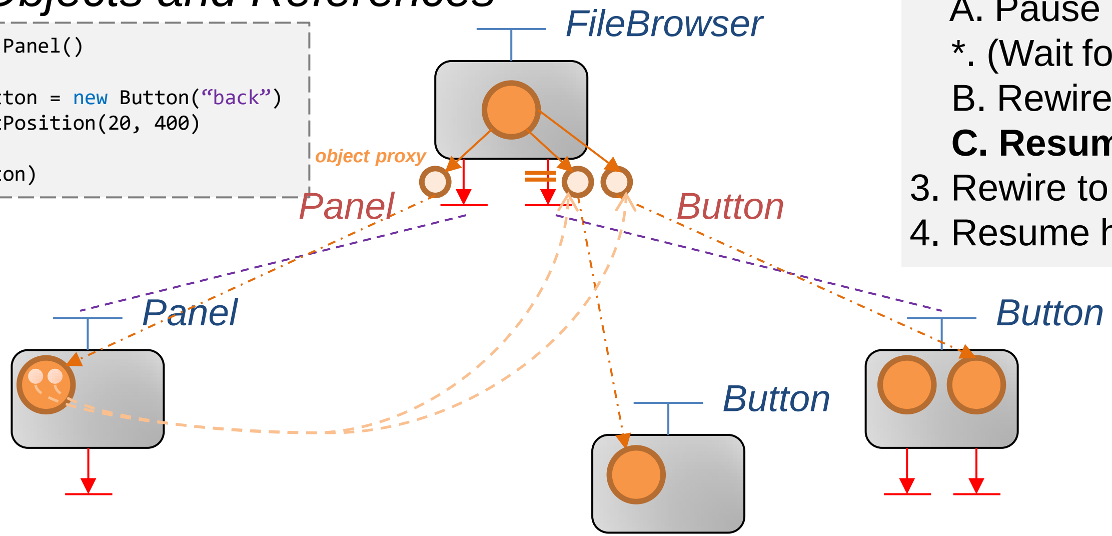
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

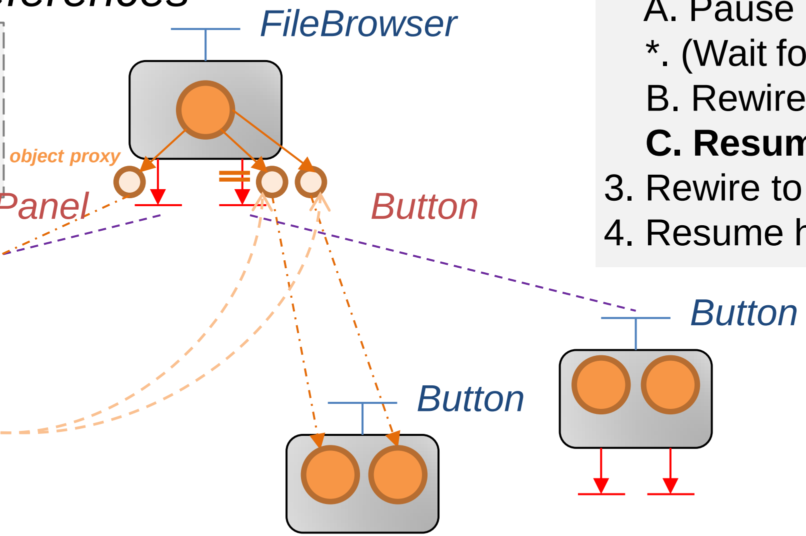
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

▷ *Objects and References*

```
Panel p = new Panel()

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

p.add(backButton)
```



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

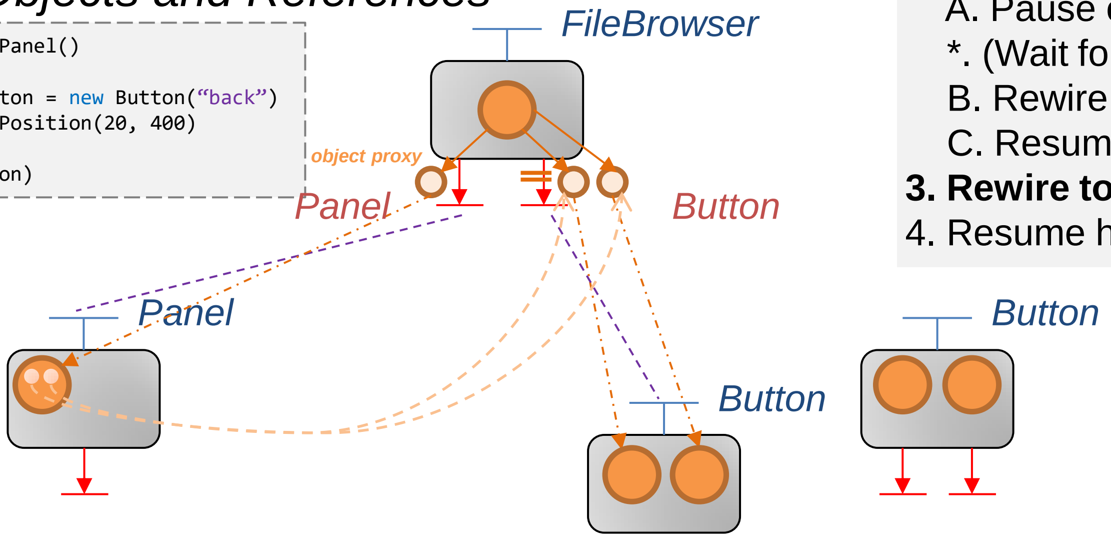
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

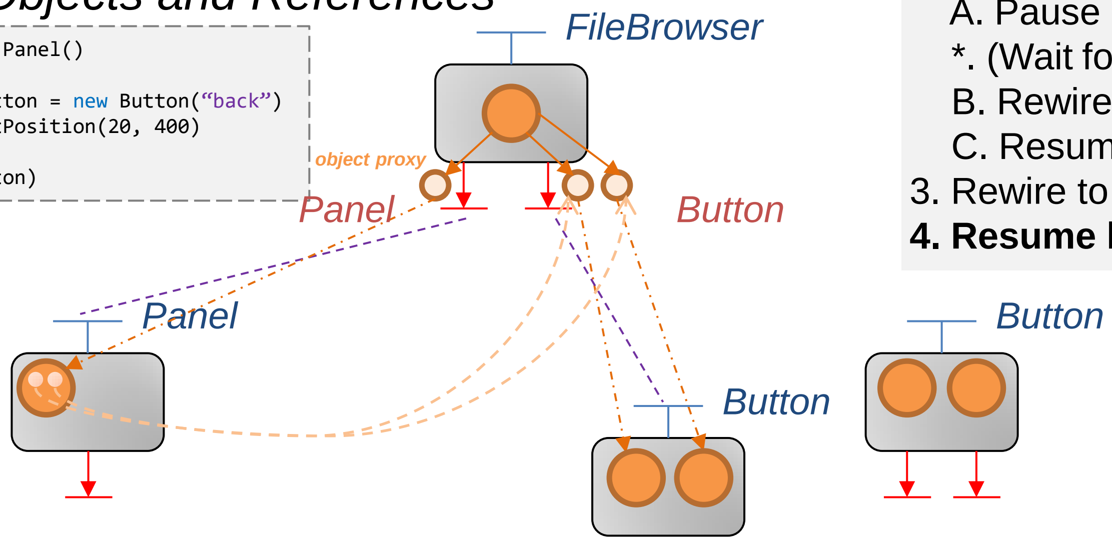
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces

Objects and References

```
Panel p = new Panel()

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

p.add(backButton)
```



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

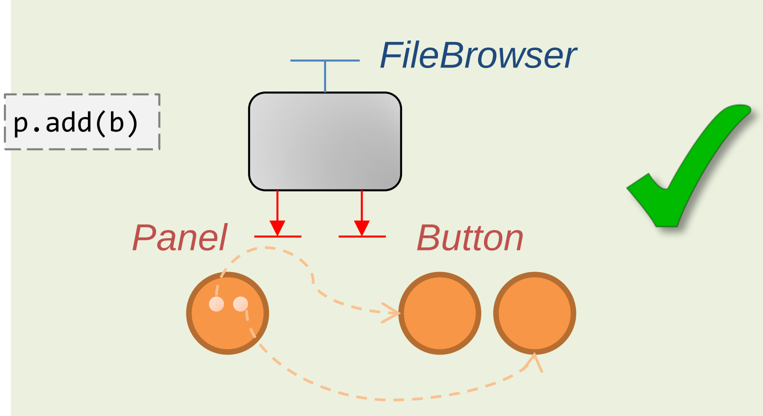
Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

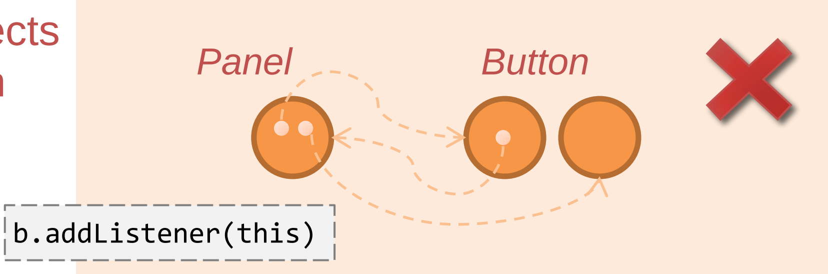
Components and Interfaces

▷ *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.



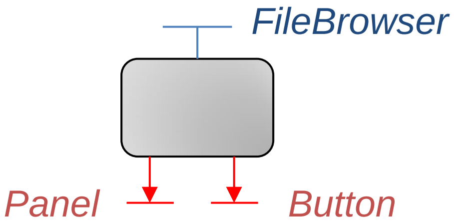
An object *F* can only pass references into objects that *F* itself created; otherwise we get cases in which perfect adaptation is impossible. The language model avoids common cases which would violate this, including self-references.



Core Concepts - Theory

- Structural Mechanics of Generalised Adaptation

Components and Interfaces
Objects and References
The Event Model




```
eventsink UIEvents(EventData ed) {
   if (ed.source === backButton)
      ...
}

Panel p = new Panel()

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

sinkevent UIEvents(backButton)
```

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

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

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

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

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

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

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

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)
   }
```

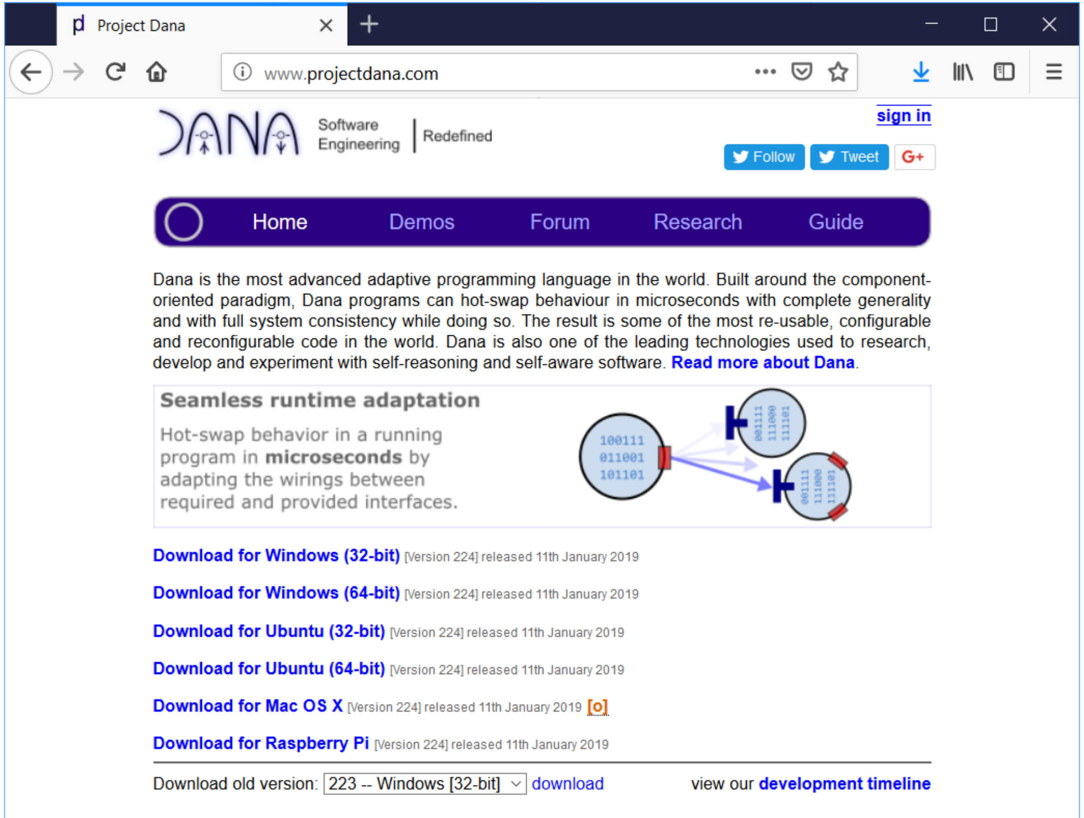
```
data Person {
   char name[]
   int age
   }
```

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.

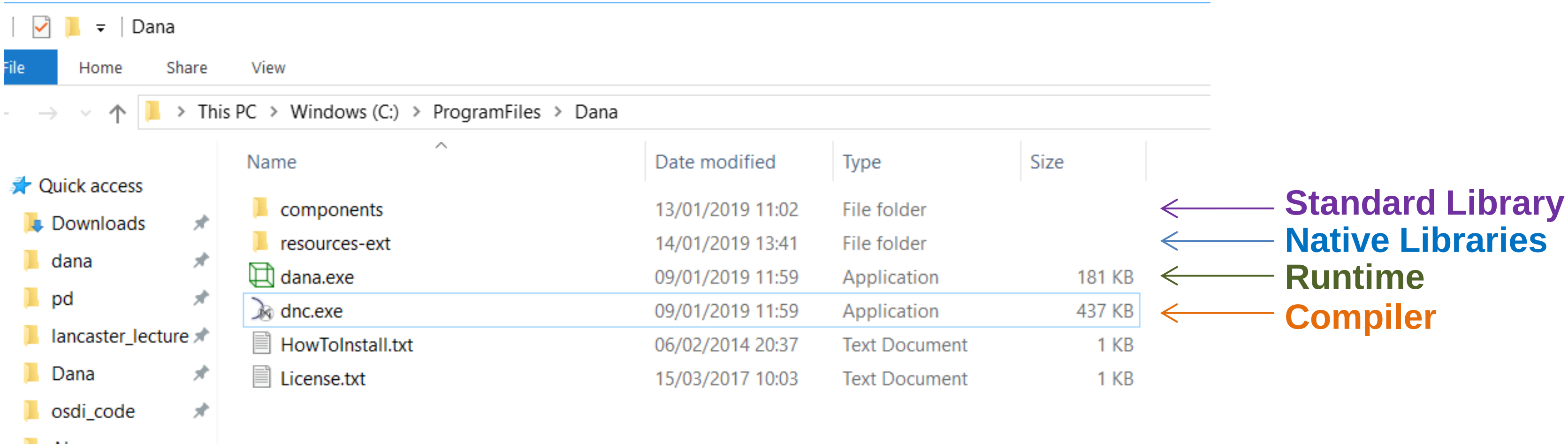
The Dana Language

Practice

Core Concepts - Practice



Core Concepts - Practice



Core Concepts - Practice

View

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

on

View

s PC > Windows (C:) > ProgramFiles > Dana > components > composition

Name	Date modified	Type	Size
Adapter.dn	05/06/2018 20:19	DN File	4 KB
Adapter.o	09/01/2019 12:00	O File	29 KB
ObjectReader.dn	17/01/2018 21:01	DN File	5 KB
ObjectReader.o	09/01/2019 12:00	O File	43 KB
ObjectWriter.dn	18/01/2018 14:38	DN File	9 KB
ObjectWriter.o	09/01/2019 12:00	O File	90 KB
RecursiveLoader.dn	10/08/2018 12:34	DN File	4 KB
RecursiveLoader.o	09/01/2019 12:00	O File	67 KB
Search.dn	02/10/2017 17:41	DN File	8 KB
Search.o	09/01/2019 12:00	O File	103 KB

Core Concepts - Practice

View

PC > Windows (C:) >

Name

- composition
- data
- doc
- encoding
- examples
- io
- locale
- media
- ml
- net
- os
- pal
- resources
- time
- ui
- util
- ws
- compinfo.dn
- compinfo.o
- docgen.dn
- docgen.o
- esher.dn
- esher.o

```
uses AdaptEvents

component provides Adapter {

    void adaptStatelessObject(IDC ofComponent, Object object, IDC source, char type[])
        {
        AdaptEvents ae
        if (dana.pauseObject(ofComponent, object))
            {
            // - construct a new object
            Object a = dana.constructObject(source :< type)
            // - notify the object that it's now the "inactive" copy
            if ((ae = dana.getObjectInterface(ofComponent, object, AdaptEvents)) != null) ae.inactive()
            // - rewire live object so calls now go to the new one (a becomes null)
            Object b = dana.rewireObject(object, a)
            // - notify the object that it's now the "active" copy
            if ((ae = dana.getObjectInterface(source, object, AdaptEvents)) != null) ae.active()
            // - allow new calls to proceed in the new object
            dana.resumeObject(object)
            // - wait for all in-progress calls to finish in the old object
            dana.waitForObject(b)
            // - wait for any in-progress asynchronous threads to finish
            dana.waitForObjectThreads(b)
            }
        }

    void adaptStatefulObject(IDC ofComponent, Object object, IDC source, char type[])
        {
        AdaptEvents ae
        if (dana.pauseObject(ofComponent, object))
```

4 KB
29 KB
5 KB
43 KB
9 KB
90 KB
4 KB
67 KB
8 KB
103 KB

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

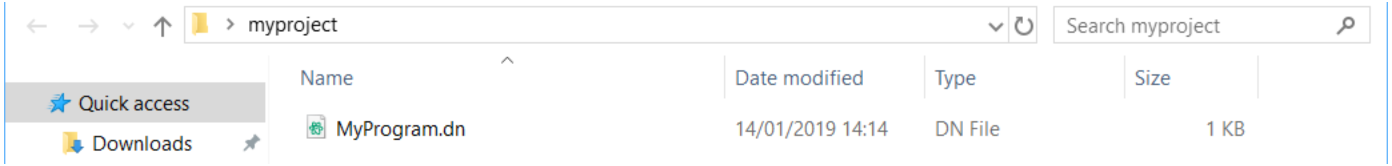
```
data AppParam{
     char string[]
     }

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

in standard library root package

```
interface Output{

     void print(char s[])

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

in standard library package 'io'

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

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

          return 0
          }

     }
```




```
> dnc MyProgram.dn
```

Core Concepts – First Program

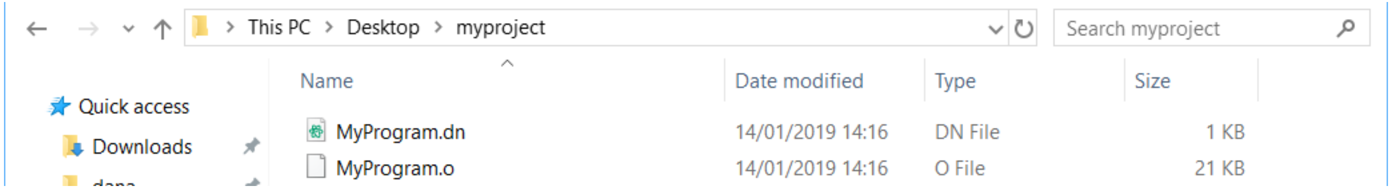
```
data AppParam{
	char string[]
	}

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'




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

Core Concepts – First Program

```
data AppParam{
    char string[]
    }

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

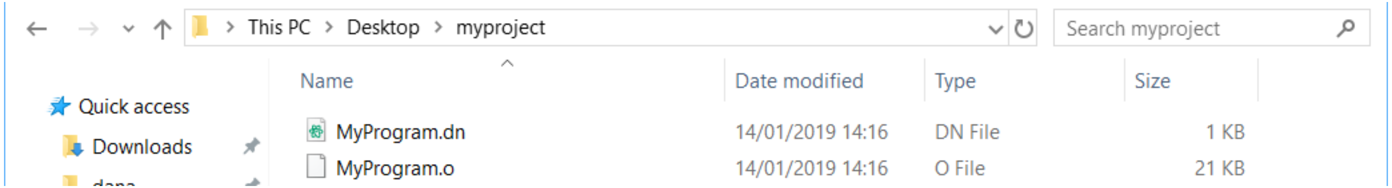
in standard library root package

```
interface Output{

    void print(char s[])

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

in standard library package 'io'

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

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

        return 0
        }

}
```

We can wire the `io.Output` required interface to *any other compatible implementation* – such as one which writes output to a file, or streams over a network – *without changing the source code*.




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

An adaptive program

- Crash course in adaptation 101

```
interface Thing {

	transfer int value

	void doThing()

	}
```

```
component provides App requires io.Output out, time.Timer timer, Thing {

	int App:main(AppParam params[])
		{
		Thing myThing = new Thing()

		while (true)
			{
			myThing.doThing()
			timer.sleep(300)
			}

		return 0
		}

	}
```

```
component provides Thing requires io.Output out, data.IntUtil iu {

	void Thing:doThing()
		{
		value ++
		out.println("value: $value")
		}

	}
```

```
component provides Thing requires io.Output out, data.IntUtil iu {

	void Thing:doThing()
		{
		value += 10
		out.println("value: $value")
		}

	}
```

An adaptive program

```
component provides App requires composition.RecursiveLoader loader,
time.Timer timer, composition.Adapter adapter {

    int App:main(AppParam params[])
        {
        IDC comA = loader.load("Main.o").mainComponent
        IDC comB = loader.load("Thing.o").mainComponent
        IDC comC = loader.load("ThingB.o").mainComponent

        App q = new App() from comA :< App

        asynch::q.main(null)

        bool mode = false

        while (true)
            {
            if (mode)
                adapter.adaptRequiredInterface(comA, "Thing", comC)
                else
                adapter.adaptRequiredInterface(comA, "Thing", comB)

            mode = !mode

            timer.sleep(1000)
            }

        return 0
        }

    }
```

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 hyper-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)

Hands-on section

- 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/saso2019tutorial