EMERGENT SOFTWARE SYSTEMS

Summer School

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

Funded by The Royal Society Newton Fund

THE ROYAL SOCIETY

Dana Advanced Topics

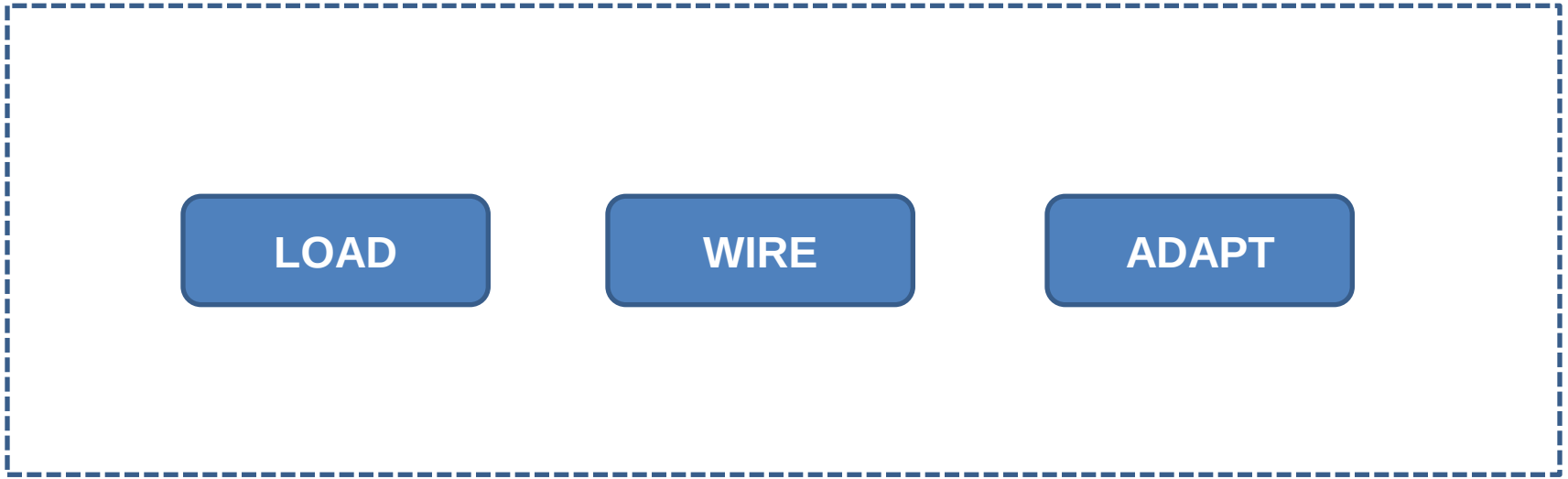
- **Adaptation in practice**
  - Assembly process;
  - Load, wire, adapt;
  - Adaptation mechanics;

- **General advanced features of Dana**
  - Concurrency;
  - Events;
  - Reflection;

Adaptation

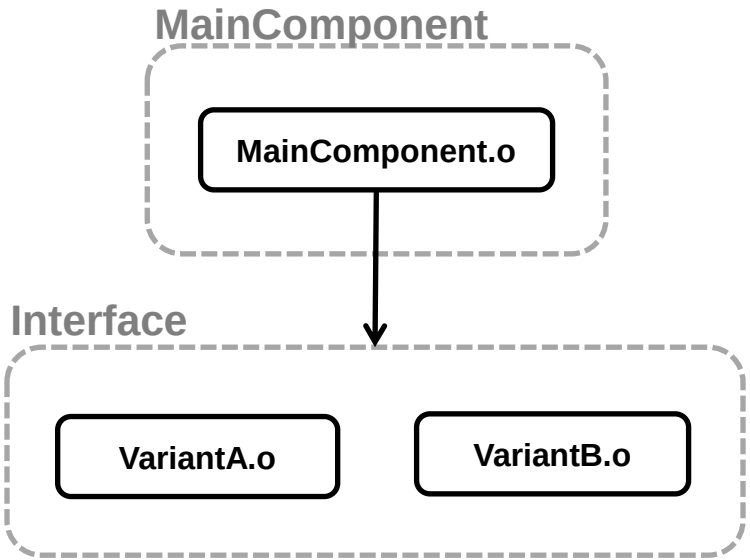
Adaptation

Assembly Process



Adaptation

Example



Adaptation

Example

```
1 interface MainComponent {
2     void method()
3 }
```

```
1 interface Interface {
2     void method2()
3 }
```

Adaptation

Example

```
interface MainComponent {
    void method()
}
```

```
interface Interface {
    void method2()
}
```

```
MainComponent.o:
component provides MainComponent requires Interface intf {
    void MainComponent:method() {
        intf.method2()
    }
}
```

Adaptation

```
1 interface Interface {
2     void method2()
3 }
```

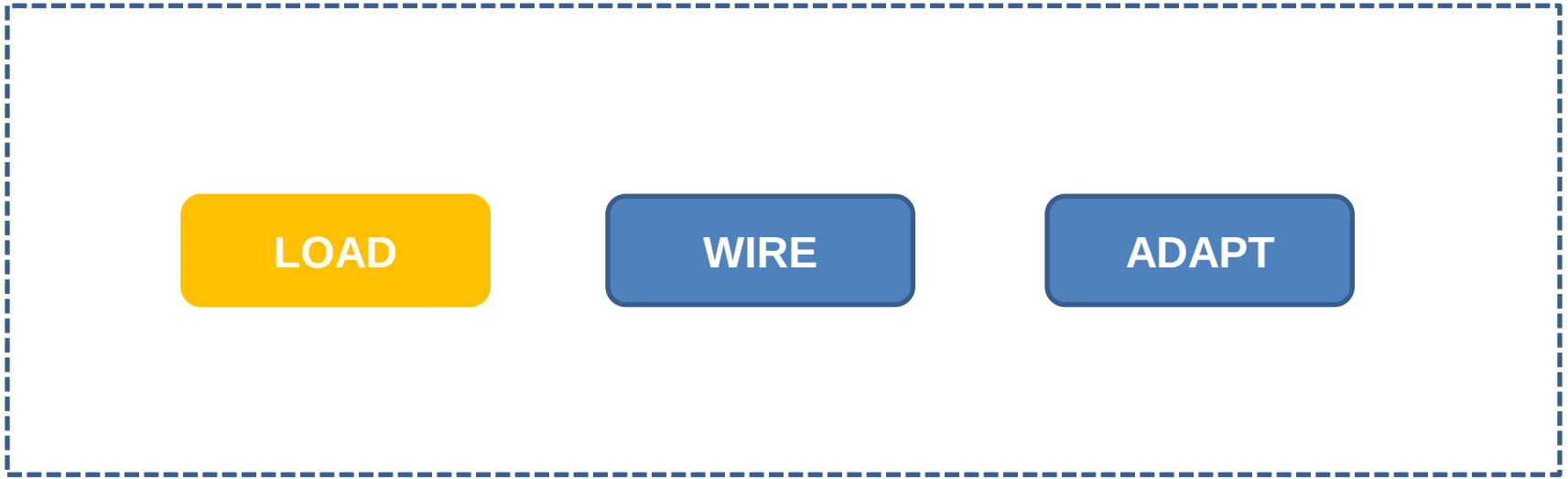
Example

```
1 VariantA.o:
2 component provides Interface requires io.Output out {
3     void Interface:method2() {
4         out.println("VariantA")
5     }
6 }
```

```
1 VariantB.o:
2 component provides Interface requires io.Output out {
3     void Interface:method2() {
4         out.println("VariantB")
5     }
6 }
```

Adaptation

Assembly Process

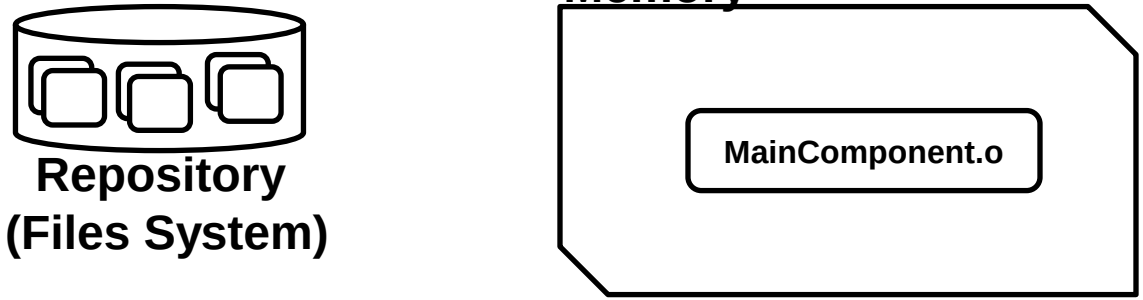


Adaptation

- **LOAD**
  - The ‘load’ stage is responsible to load a component to memory;
  - This is performed by calling the function *load* from *Loader*;
  - The load function returns an instance of **IDC**, which is a Dana data type with all the necessary information of the loaded component;

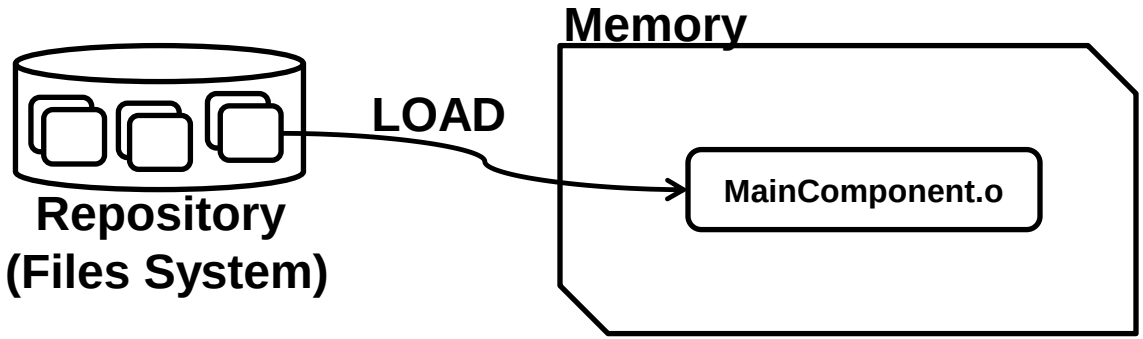
Adaptation

- **LOAD**
  - The ‘load’ stage is responsible to load a component to memory;
  - This is performed by calling the function *load* from *Loader*;
  - The load function returns an instance of **IDC**, which is a Dana data type with all the necessary information of the loaded component;



Adaptation

- **LOAD**
  - The ‘load’ stage is responsible to load a component to memory;
  - This is performed by calling the function *load* from *Loader*;
  - The load function returns an instance of **IDC**, which is a Dana data type with all the necessary information of the loaded component;



Adaptation

- **LOAD**
  - The ‘load’ stage is responsible to load a component to memory;
  - This is performed by calling the function *load* from *Loader*;
  - The load function returns an instance of **IDC**, which is a Dana data type with all the necessary information of the loaded component;

```
1 component provides App requires Loader loader {
2     int App:main(AppParam param[]) {
3         IDC component = loader.load("Component.o")
4     }
5 }
```

Adaptation

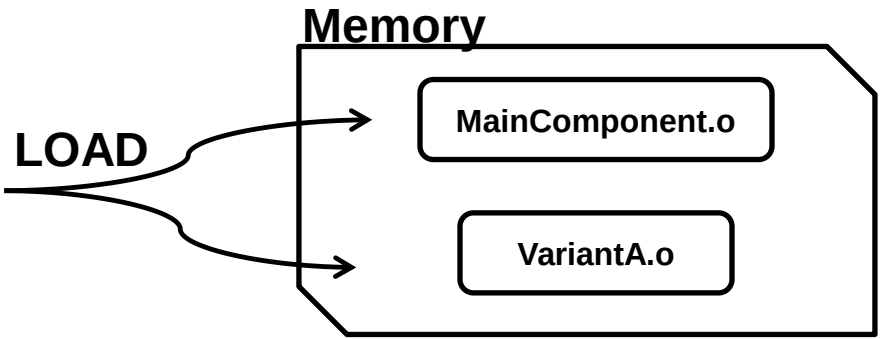
Assembly Process

Adaptation

- **WIRE**
  - The ‘wire’ stage wires (connects) a component that requires a specific interface to another component that provides that interface;
  - This is performed by calling the function *rewire*, a function provided by the Dana interpreter*;*

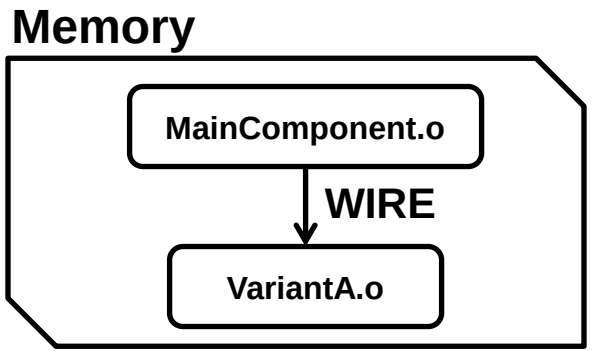
Adaptation

- **WIRE**
  - The ‘wire’ stage wires (connects) a component that requires a specific interface to another component that provides that interface;
  - This is performed by calling the function *rewire*, a function provided by the Dana interpreter*;*



Adaptation

- **WIRE**
  - The ‘wire’ stage wires (connects) a component that requires a specific interface to another component that provides that interface;
  - This is performed by calling the function *rewire*, a function provided by the Dana interpreter*;*



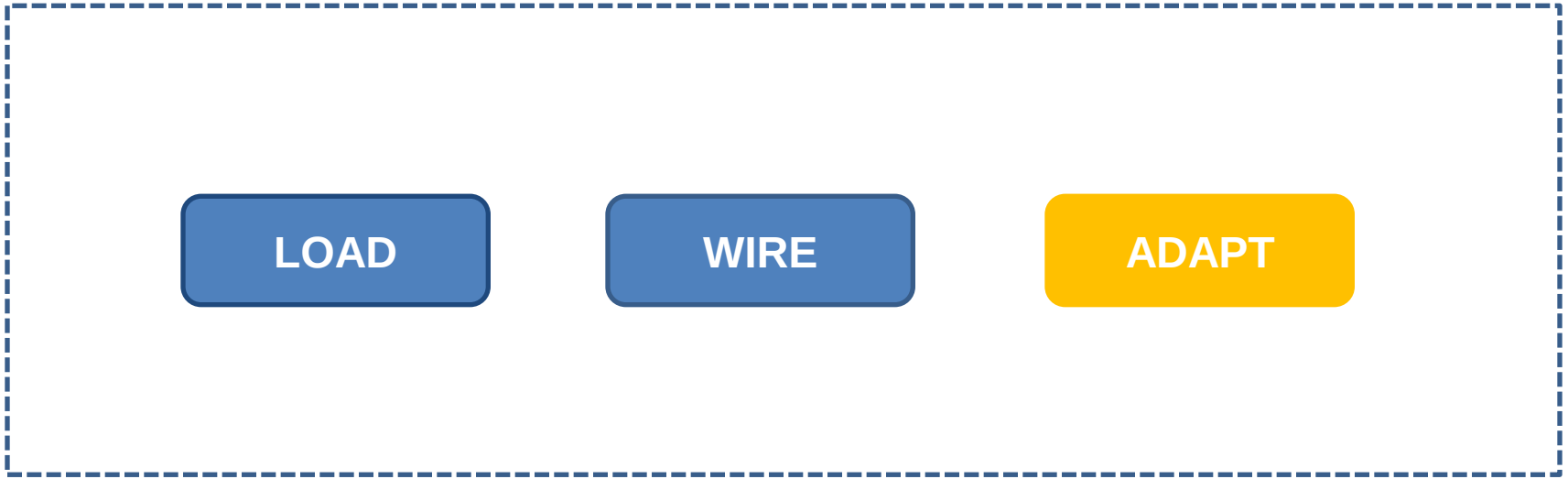
Adaptation

- **WIRE**
  - The ‘wire’ stage wires (connects) a component that requires a specific interface to another component that provides that interface;
  - This is performed by calling the function *rewire*, a function provided by the Dana interpreter;

```
1  component provides App requires Loader loader {
2      int App:main(AppParam param[]) {
3          IDC mainComponent = loader.load("MainComponent.o")
4          IDC variantA = loader.load("VariantA.o")
5          IDC variantB = loader.load("VariantB.o")
6
7          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
8
9          MainComponent obj = new MainComponent() from mainComponent
10         obj.method()
11     }
12 }
```

Adaptation

Assembly Process

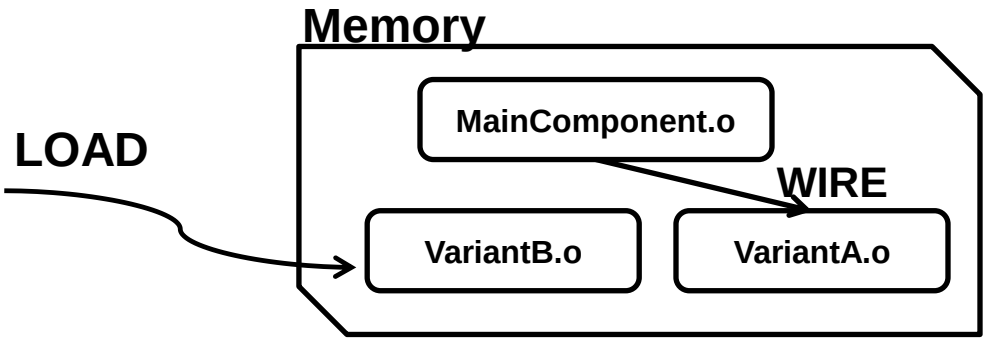


Adaptation

- **ADAPT**
  - The ‘adapt’ stage is trigger only when an adaptation is desired. It basically rewires a connected component to its variant;
  - This is performed by calling the function *adaptRequiredInterface* from *Adapter;*

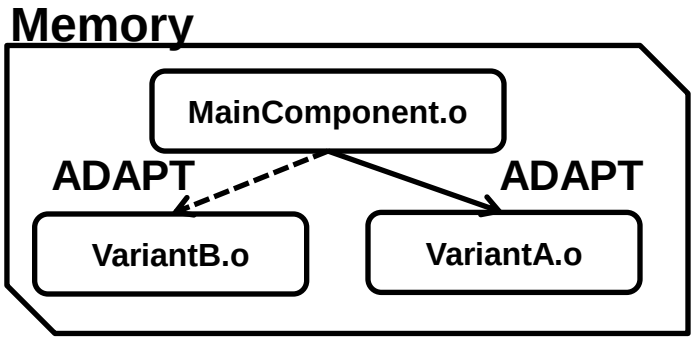
Adaptation

- **ADAPT**
  - The ‘adapt’ stage is trigger only when an adaptation is desired. It basically rewires a connected component to its variant;
  - This is performed by calling the function *adaptRequiredInterface* from *Adapter;*



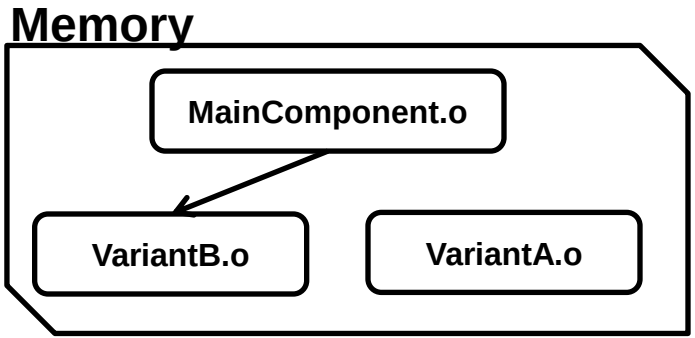
Adaptation

- **ADAPT**
  - The ‘adapt’ stage is trigger only when an adaptation is desired. It basically rewires a connected component to its variant;
  - This is performed by calling the function *adaptRequiredInterface* from *Adapter;*



Adaptation

- **ADAPT**
  - The ‘adapt’ stage is trigger only when an adaptation is desired. It basically rewires a connected component to its variant;
  - This is performed by calling the function *adaptRequiredInterface* from *Adapter;*



Adaptation

```
1  component provides App requires Loader loader,
2   composition.Adapter adapter {
3      int App:main(AppParam param[]) {
4          IDC mainComponent = loader.load("MainComponent.o")
5          IDC variantA = loader.load("VariantA.o")
6          IDC variantB = loader.load("VariantB.o")
7
8          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
9
10         MainComponent obj = new MainComponent() from mainComponent
11         obj.method()
12
13         adapter.adaptRequiredInterface(mainComponent,
14             "Interface",
15             variantB)
16         obj.method()
17     }
18 }
```

Adaptation

EXAMPLE

Adaptation

What is the problem with this code?

```
1  component provides App requires Loader loader,
2   composition.Adapter adapter {
3      int App:main(AppParam param[]) {
4          IDC mainComponent = loader.load("MainComponent.o")
5          IDC variantA = loader.load("VariantA.o")
6          IDC variantB = loader.load("VariantB.o")
7
8          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
9
10         MainComponent obj = new MainComponent() from mainComponent
11         obj.method()
12
13         adapter.adaptRequiredInterface(mainComponent,
14             "Interface",
15             variantB)
16         obj.method()
17     }
18 }
```

Adaptation

- **Problems with the adaptation code**
  - The component path has to be known to be loaded.

Adaptation

What is the problem with this code?

```
1  component provides App requires Loader loader,
2   composition.Adapter adapter {
3      int App:main(AppParam param[]) {
4          IDC mainComponent = loader.load("MainComponent.o")
5          IDC variantA = loader.load("VariantA.o")
6          IDC variantB = loader.load("VariantB.o")
7
8          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
9
10         MainComponent obj = new MainComponent() from mainComponent
11         obj.method()
12
13         adapter.adaptRequiredInterface(mainComponent,
14             "Interface",
15             variantB)
16         obj.method()
17     }
18 }
```

Adaptation

- **Problems with the adaptation code**
  - The component path has to be known to be loaded

How can we search for components given a specific interface?

Adaptation

What is the problem with this code?

```
1  component provides App requires Loader loader,
2   composition.Adapter adapter {
3      int App:main(AppParam param[]) {
4          IDC mainComponent = loader.load("MainComponent.o")
5          IDC variantA = loader.load("VariantA.o")
6          IDC variantB = loader.load("VariantB.o")
7
8          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
9
10         MainComponent obj = new MainComponent() from mainComponent
11         obj.method()
12
13         adapter.adaptRequiredInterface(mainComponent,
14             "Interface",
15             variantB)
16         obj.method()
17     }
18 }
```

Adaptation

- **Problems with the adaptation code**
  - The component path has to be known to be loaded

How can we search for components given a specific interface?

  - The interface also has to be known to (re)wire or adapt components.

Adaptation

What is the problem with this code?

```
1  component provides App requires Loader loader,
2   composition.Adapter adapter {
3      int App:main(AppParam param[]) {
4          IDC mainComponent = loader.load("MainComponent.o")
5          IDC variantA = loader.load("VariantA.o")
6          IDC variantB = loader.load("VariantB.o")
7
8          dana.rewire(myComponent :> "Interface", variantA :< "Interface")
9
10         MainComponent obj = new MainComponent() from mainComponent
11         obj.method()
12
13         adapter.adaptRequiredInterface(mainComponent,
14             "Interface",
15             variantB)
16         obj.method()
17     }
18 }
```

Adaptation

- **Problems with the adaptation code**
  - The component path has to be known to be loaded

How can we search for components given a specific interface?

  - The interface also has to be known to (re)wire or adapt components.

How can we get a list of required and provided interfaces from a specific component?

Adaptation

- Extract information from object files
  - Given an object file get a list of required and provided interfaces.
- Searching for components given an interface
  - Given a specific interface, find components that provide it.

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.
2. Once a list of required interface is extracted from *mainComponent.o*, look for components which provide (implement) ‘Interface’;

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.
2. Once a list of required interface is extracted from *mainComponent.o*, look for components which provide (implement) ‘Interface’;
3. Loads *mainComponent.o*;

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.
2. Once a list of required interface is extracted from *mainComponent.o*, look for components which provide (implement) ‘Interface’;
3. Loads *mainComponent.o*;
4. Once you have the path to the components which provide ‘Interface’, load them;

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.
2. Once a list of required interface is extracted from *mainComponent.o*, look for components which provide (implement) ‘Interface’;
3. Loads *mainComponent.o*;
4. Once you have the path to the components which provide ‘Interface’, load them;
5. Continue with WIRE, and at some point ADAPT.

Adaptation

1. Given the *mainComponent* path, extract information from *mainComponent.o*.

```
char rootCompPath[] = "main_component/MainComponent.o"

// extracting info from mainComponent
ObjectWriter objWriter = new ObjectWriter(rootCompPath)
InfoSection infoSection = objWriter.getInfoSection("DNIL", "json")
Section sec = jsonEncoder.jsonToData(infoSection.content, typeof(Section), null)
```

Adaptation

```
char rootCompPath[] = "main_component/MainComponent.o"

// extracting info from mainComponent
ObjectWriter objWriter = new ObjectWriter(rootCompPath)
InfoSection infoSection = objWriter.getInfoSection("DNIL", "json")
Section sec = jsonEncoder.jsonToData(infoSection.content, typeof(Section), null)
```

```
data Section {
    Intf providedInterfaces[]
    Intf requiredInterfaces[]
}

data Intf {
    char package[]
    char alias[]
    Function functions[]
}
```

```
data Function {
    char name[]
    char returnType[]
    Parameter parameters[]
}

data Parameter {
    char type[]
}
```

Adaptation

2. Once a list of required interface is extracted from *mainComponent.o*, look for components which provide (implement) ‘Interface’;

```
// looking for components that provide requiredInterface
String compsPath[] = search.getComponents(sec.requiredInterfaces[0].package)
```

Adaptation

3. Loads *mainComponent.o*;

```
//load mainComponent.o
IDC mainComponent = loader.load(rootCompPath)
```

Adaptation

4. Once you have the path to the components which provide ‘Interface’, load them;

```
// loading components that provide requiredInterface
Variants variants[] = new Variants[compsPath.arrayLength]
for (int count = 0; count < compsPath.arrayLength; count++) {
    variants[count] = new Variants()
    variants[count].providedIntf = sec.requiredInterfaces[0].package
    variants[count].comp = loader.load(compsPath[count].string)
}
```

```
data Variants {
    char providedIntf[]
    IDC comp
}
```

Adaptation

5. Continue with WIRE, and at some point ADAPT.

```
// continuing with WIRE(ing)
dana.rewire(mainComponent :> sec.requiredInterfaces[0].package,
    variants[0].comp :< sec.requiredInterfaces[0].package)

MainComponent obj = new MainComponent() from mainComponent
obj.method()

// and ADAPT(ing)
adapter.adaptRequiredInterface(mainComponent,
    sec.requiredInterfaces[0].package, variants[1].comp)
obj.method()
```

```
int App:main(AppParam param[]) {
    char rootCompPath[] = "main_component/MainComponent.o"

    // extracting info from mainComponent
    ObjectWriter objWriter = new ObjectWriter(rootCompPath)
    InfoSection infoSection = objWriter.getInfoSection("DNIL", "json")
    Section sec = jsonEncoder.jsonToData(infoSection.content, typeof(Section), null)

    // looking for components that provide requiredInterface
    String compsPath[] = search.getComponents(sec.requiredInterfaces[0].package)

    //load mainComponent.o
    IDC mainComponent = loader.load(rootCompPath)

    // loading components that provide requiredInterface
    Variants variants[] = new Variants[compsPath.arrayLength]
    for (int count = 0; count < compsPath.arrayLength; count++) {
        variants[count] = new Variants()
        variants[count].providedIntf = sec.requiredInterfaces[0].package
        variants[count].comp = loader.load(compsPath[count].string)
    }

    // continuing with WIRE(ing)
    dana.rewire(mainComponent :> sec.requiredInterfaces[0].package,
        variants[0].comp :< sec.requiredInterfaces[0].package)

    MainComponent obj = new MainComponent() from mainComponent
    obj.method()

    // and ADAPT(ing)
    adapter.adaptRequiredInterface(mainComponent,
        sec.requiredInterfaces[0].package, variants[1].comp)
    obj.method()

    return 0
}
```

Adaptation

EXAMPLE

ADVANCED DANA FEATURES

Advanced Dana Features

- Concurrency
  - Starting threads;
  - Synchronisation between threads;
  - Mutual exclusion between multi-threads;
- Events;
  - Defining events;
  - Triggering events;
  - Catching events;
- Reflection;
  - Field selection;
  - Querying structures;
  - Creating instances of data types;

Concurrency

Concurrency is part of every modern system. Here we present ways to trigger and control threads in Dana.

Concurrency

- Starting threads

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        asynch::printMany("thread", 100)
        return 0
    }
}
```

Concurrency

- Starting threads

What is the problem with this code?

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        asynch::printMany("thread", 100)
        return 0
    }
}
```

Concurrency

EXAMPLE

Concurrency

- Synchronisation
  - *join()*

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        Thread t = asynch::printMany("thread", 100)
        t.join()
        return 0
    }
}
```

Concurrency

EXAMPLE

Concurrency

- Synchronisation
  - *join()*

This solves our problem!

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        Thread t = asynch::printMany("thread", 100)
        t.join()
        return 0
    }
}
```

Concurrency

- Synchronisation
  - *wait()*

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        asynch::printMany("thread", 100)
        this.thread.wait()
        return 0
    }
}
```

Concurrency

- Synchronisation
  - *wait()*

What is the problem with this code?

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
    }

    int App:main(AppParam param[]) {
        asynch::printMany("thread", 100)
        this.thread.wait()
        return 0
    }
}
```

Concurrency

EXAMPLE

Concurrency

- Synchronisation
  - *signal()*

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
        mainThread.signal()
    }

    int App:main(AppParam param[]) {
        mainThread = this.thread
        asynch::printMany("thread", 100)
        this.thread.wait()
        return 0
    }
}
```

Concurrency

EXAMPLE

Concurrency

- Synchronisation
  - *signal()*

This solves our problem!

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

    void printMany(char string[], int times) {
        for (int count = 0; count < times; count++) {
            out.println("$(string)")
        }
        mainThread.signal()
    }

    int App:main(AppParam param[]) {
        mainThread = this.thread
        asynch::printMany("thread", 100)
        this.thread.wait()
        return 0
    }
}
```

Events

Event is a way to notify multiple parts of the system when something occurs. In Dana, it is also used as a work around the fact that we cannot pass an instance of a component itself (using the key work *this*) to different components to implement call-backs. In these cases, we trigger an event, and everyone who's registered a method to handle the event gets invoked.

Events

- Defining events;
- Triggering events;
- Catching events;

Events

- Defining events;
  - Events are defined in interfaces.

```
1 interface Event {
2     event eventExample()
3     void trigger()
4 }
```

Events

- Triggering events;

```
1 component provides events.Event requires
2  time.Timer timer {
3     void Event:trigger() {
4         while (true) {
5             timer.sleep(1000)
6             emitevent eventExample()
7         }
8     }
9 }
```

Events

- Triggering events;

```
interface Event {
    event eventExample()
    void trigger()
}
```

```
component provides events.Event requires
 time.Timer timer {
    void Event:trigger() {
        while (true) {
            timer.sleep(1000)
            emitevent eventExample()
        }
    }
}
```

Events

- Catching events;

```
1  uses events.EventData
2
3  component provides App requires io.Output out,
4   events.Event {
5
6      eventsink Events(EventData ed) {
7          if (ed.type == Event.[eventExample]) {
8              out.println("Caught event here!")
9          }
10     }
11
12     int App:main(AppParam params[]) {
13         Event myEvent = new Event()
14         sinkevent Events(myEvent)
15
16         myEvent.trigger()
17
18         this.thread.wait()
19
20         return 0
21     }
22 }
```

Events

EXAMPLE

Reflection

Reflection allows you to query a type's structure, and to create new instances of a data type that have a given structure, which is really useful for instance to parse json strings into Dana data types. It also enables field selection to implement, for instance, generic sorting algorithms.

Reflection

- Field selection;
- Querying structures;
- Creating instances of data types;

Reflection

- Field selection;

```
1  data Person {
2      char name[]
3      int age
4  }
5
6  component provides App requires io.Output out {
7      int App:main(AppParam params[]) {
8          Person p = new Person("Sam", 19)
9          TypeField tf = Person.[name]
10
11         out.println("$(p:.tf) = $(p.name)")
12         p:.tf = "John"
13         out.println("$(p:.tf) = $(p.name)")
14         return 0
15     }
16 }
```

Reflection

EXAMPLE

Reflection

- Field selection;

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

component provides App requires io.Output out {
    void greaterThan(Data a, Data b, TypeField field) {
        if (a:.field > b:.field) {
            out.println("Yes!")
        } else {
            out.println("No!")
        }
    }

    int App:main(AppParam params[]) {
        Person p1 = new Person("Sam", 19)
        Person p2 = new Person("John", 20)

        greaterThan(p1, p2, Person.[age])
        greaterThan(p2, p1, Person.[age])

        return 0
    }
}
```

Reflection

EXAMPLE

Reflection

- Querying structures;

```
1  uses reflect.Type
2
3  data Person {
4      char name[]
5      int age
6  }
7
8  component provides App requires io.Output out {
9
10     int App:main(AppParam params[]) {
11         Person p = new Person("Sam", 19)
12         Type type = typeof(p)
13         for (int count = 0; count < type.fields.arrayLength; count++) {
14             out.println("$(type.fields[count].name)")
15         }
16         return 0
17     }
18 }
```

Reflection

- Querying structures;

```
1  uses reflect.Type
2
3  data Person {
4      char name[]
5      int age
6  }
7
8  component provides App requires io.Output out {
9
10     int App:main(AppParam params[]) {
11         Person p = new Person("Sam", 19)
12         Type type = typeof(p)
13         for (int count = 0; count < type.fields.arrayLength; count++) {
14             out.println("$(type.fields[count].name)")
15         }
16         return 0
17     }
18 }
```

```
data Type {
    const byte INTEGER      = 1
    const byte DECIMAL      = 2
    const byte DATA         = 3
    const byte OBJECT       = 4
    const byte ARRAY        = 5
    const byte THREAD       = 6
    const byte FUNCTION     = 7
    const byte EVENT        = 8
    const byte COMPONENT    = 9
    const byte F_BOOL       = 0x1
    const byte F_CHAR       = 0x2
    byte class
    byte flags
    int size
    Field fields[]
}

data Field {
    const int FLAG_RECURSION = 0x1
    Type type
    char name[]
    byte flags
}
```

Reflection

EXAMPLE

Reflection

- Creating instances of data types;

```
1  uses reflect.Type
2
3  data Person {
4      char name[]
5      int age
6  }
7
8  component provides App requires io.Output out, data.IntUtil iu {
9      Data createData(Type t) {
10         Data d = new Data() from t
11         return d
12     }
13
14     int App:main(AppParam params[]) {
15         Person p = createData(typeof(Person))
16         p.name = "Sam"
17         p.age = 19
18
19         out.println("$(p.name), $(iu.intToString(p.age))")
20         return 0
21     }
22 }
```

Reflection

EXAMPLE

Summary

- **Adaptation** is a sub-process of the **assembly process**;
- **Assembly process** is the process that enables Dana users to dynamically compose and adapt *local* Dana programs;
- **Assembly process** runs on the meta-level
  - This means the assemble process controls the target application, and it is not the target application;
- **Adaptation** is basically the process of **rewiring** components;
- **(Re)wiring** is the process of connecting components;
  - A component is connected to another following the provided-required interface policy defined by the component-based model;

Practical Assignment

We expect you to write a Dana component responsible to locate other components, wire them up into a functioning program and run the program. After the program is running, we expect you to perform adaptation between multiple component variants. More information on the practical assignment is in “Practical 2 – Dana Advanced.pdf” file, which can be found in our **github** repository.