Java Communicating Sequential Processes

‘Design Of JCSP AWT Classes’

Paul Austin
pda1@ukc.ac.uk
University Of Kent Canterbury
BSc Computer Science with an Industrial Year
3rd Year Project

Contents

Figures

1 Introduction

This document describes the functionality and design of the classes in the Java Communicating Sequential Processes [1] package that are used to provide a CSP [2] Channel interface to the Java [3] Abstract Window Toolkit (AWT) user interface components. The design will be described using the Unified Modelling Language [4] (UML) notation with descriptive text describing why certain decisions were made. The discussion will assume knowledge of the facilities and concepts of the CSP model in the OCCAM [5] programming language and of programming in the Java programming language.

Previous work was done at the University of Kent at Canterbury by David Beckett to provide Channel interfaces to two of the AWT Components, *`Button`* and *`Scrollbar`*. These are used as a starting point to developing Channel interfaces for most of the classes in the `jcsp.awt` package.

2 Functionality

This section describes the functionality that the library must implement to provide features similar to those offered by the AWT but using the JCSP `Channel` for event notification and configuration.

The following list summarises the components that should be implemented.

- Applet
- Button
- Canvas
- Checkbox
- CheckboxMenuItem
- Choice
- Component
- Container
- Dialog
- FileDialog
- Frame
- Label
- List
- MenuItem
- Panel
- ScrollPane
- Scrollbar
- TextArea
- TextField
- Window

2.1 CSP Model

In the CSP model, channels are used instead of methods to send messages to an object. The advantages of this approach are.

1. The state of the object cannot be changed behind its back.
2. Objects can be joined together without each one knowing each others type.

Channels are also used to send event notifications for a component. When an event is generated, an object representing that event is sent down a channel by the component. This can then be used by other processes to handle the event.

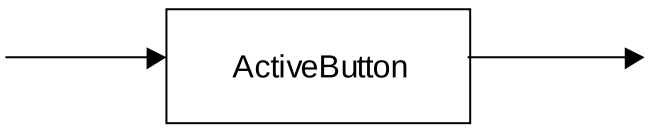

Figure 1 - Button Model

3 Design and Implementation

The design of the new CSP style components is discussed below in three sections.

1. How to convert the Java Event Model into a Channel model.
2. How to convert the Java method based configuration model into a Channel model.
3. How to bring both parts together to implement the full components in an extendible way.

3.1 Event Handling

3.1.1 Java Event Model

This section serves as an introduction to event handling in the Java programming language. This will be used as a base for the discussions later in this document on how to implement the CSP channel model over the top of the Java model.

There are three parts to this section.

1. Description of the original model used in Java 1.0 API.
2. Description of the new event model introduced in the Java 1.1 API
3. A comparison of the two approaches, with the reasons for choosing one above the other.

3.1.1.1 JDK 1.0 Model

In the Java 1.0 library, the event handling is performed using an inheritance-based model for event handling. Each *`Component`* (i.e. *`Button`*, *`Frame`*, *`TextField`* etc) in the AWT is a subclass of the class *`Component`* which defines the methods required to handle events. To handle events for a *`Component`* using this model it is necessary to subclass either the actual *`Component`* or it's parent *`Container`* and override the event methods with the event handling code.

The diagram below shows part of the AWT inheritance hierarchy. Note all the classes inherit from the *`Component`* class, which defines the event handling methods with no implementation. The *`Container`* class is the super class of all the *`Component`* subclasses that contain other *`Components`*. The *`OKButton`* class is a *`Button`* with the `action()` method overridden to handle the button press events.

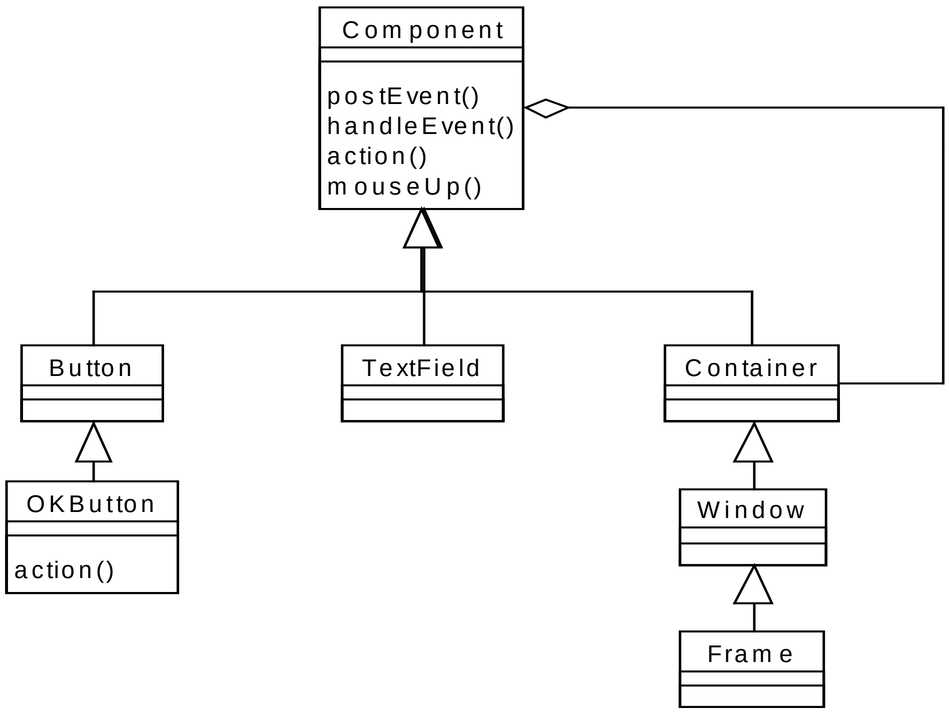

Figure 2 - Relationship between 1.0 AWT Components

The diagram below shows which methods are called for different event types.

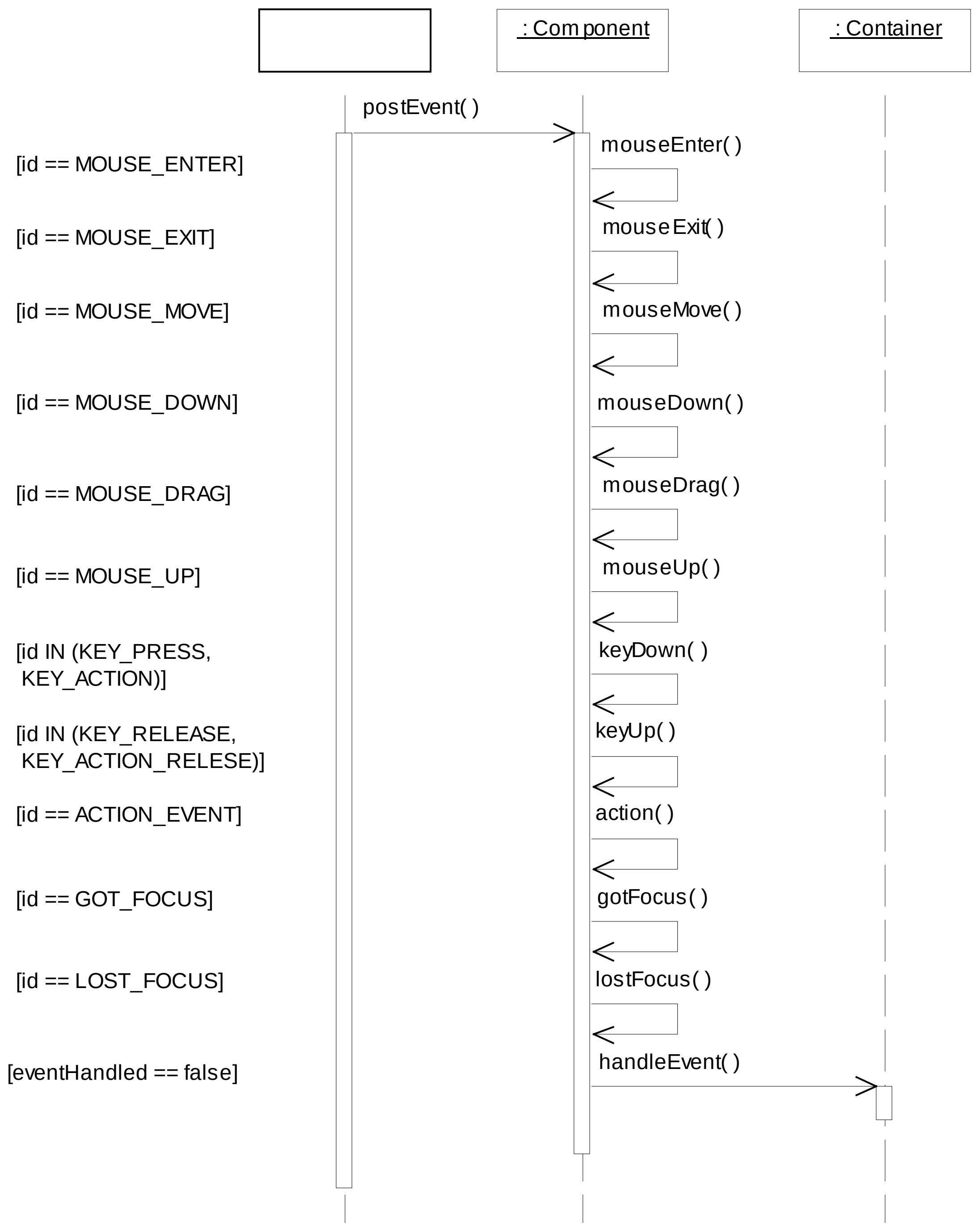

Figure 3 - Interaction between event handling methods

When an event happens to a component a new `Event` object is generated and the event handling thread invokes the `postEvent()` method on that component. The role of this method is co-ordinate the event handling process. First, it invokes the `handleEvent()` method on itself to give the subclass of the component a chance to handle the event. If the event if not fully handled and the component has a parent container the `handleEvent()` method is invoked on the parent container. The `handleEvent()` method contains a switch statement which will invoke utility methods to handle different events depending on the type of the event. For example, if the event type is MOUSE_UP the `mouseUp(e, e.x, e.y)` method will be invoked.

In this model, there are two options if you want to handle an event for a component in an application. Both examples will assume we are handling events for two buttons 'OK' and 'Cancel' contained within a *`Frame`*.

The first option is to create a new subclass of *`Button`* for each of the two buttons. With this option, two methods can be overridden to provide the event handling code. The two methods are `action(Event, Object)` or `handleEvent(Event)`. For the *`OKButton`* the `action(…)` method will be overridden with the code to perform the 'OK' functionality. For the *`CancelButton`* the `handleEvent(…)` method will be overridden. The body of this method will check to see if the *`Event`* was for an action if so the code to perform the 'Cancel' button will be executed otherwise propagate it up the containment hierarchy will be executed. The following Code example demonstrates how to do this.

```
public class DemoFrame extends Frame {
  public DemoFrame() {
    super("Event Handling Demo");
    setLayout(new FlowLayout());
    add(new OKButton());
    add(new CancelButton());
    pack();
    show();
  }
}

class OKButton extends Button {
  public OKButton() {
    super("OK");
  }

  public boolean action(Event e, Object arg) {
    // Perform the ok action here
    return true; // the event has been fully handled
  }
}

class CancelButton extends Button {
  public CancelButton() {
    super("Cancel");
  }

  public boolean handleEvent(Event e) {
    if (e.id == ACTION_EVENT) {
      // Perform the cancel action here
      return true; // the event has been fully handled
    }
    return super.handleEvent(); // call the standard event handling
  }
}
```

Code 1 - Event handling using 1.0 Event Model (Example 1)

The second option is to handle the events for both buttons in a subclass of the *`Frame`* that contains the two buttons. We can either handle the events in the `action()` or `handleEvent()` methods in the same way as we did for the buttons. In the example we are going to override the `action()` method as it produces cleaner code.

```
public class DemoFrame extends Frame {
  public DemoFrame() {
    super("Event Handling Demo");
    setLayout(new FlowLayout());
    add(new Button("OK"));
    add(new Button("Cancel"));
    pack();
    show();
  }

  public boolean action(Event e, Object arg) {
    if (arg.equals("OK") {
      // Perform the continue action here
      return true; // the event has bee fully handled
    }
    if (arg.equals("Cancel") {
      // Perform the cancel action here
      return true; // the event has bee fully handled
    }
    return super.action(e, arg);
  }
}
```

Code 2 - Event handling using 1.0 Event Model (Example 2)

3.1.1.2 JDK 1.1 Delegation Model

The Java 1.1 library introduced a radically different model for handling events[1], which separates the Application logic from the GUI. The new model uses the Observer design pattern to notify one or more event handlers that have been registered with the component.

In the new model, there is one event object for each type of event rather than just one event object. The event objects contain the relevant information about the event and the source component that generated it. The types of event are split into two conceptual groups shown below.

1. Semantic events
   - *`ActionEvent`*
   - *`AdjustmentEvent`*
   - *`ItemEvent`*
   - *`TextEvent`*
2. low-level events
   - *`ComponentEvent`* - component moved, resized, shown or hidden
   - *`FocusEvent`* - component lost focus or gained focus
   - *`KeyEvent`* - key pressed, key released or key typed on the component
   - *`MouseEvent`* - mouse clicked, pressed, released, entered, exited, dragged or moved
   - *`ContainerEvent`* - component added or removed from the container
   - *`WindowEvent`* - window opened, closed, closing, activated, deactivated, activated, deiconified or iconified

For each event type there is an event listener interface that defines the methods that event handlers must define. Each component has a method to register an event listener of the form `addXXXEventListener(XXXListener)` for each event type it generates. The low-level events are generated for all components and the `addXXXEventListener(…)` methods for these are defined in the *`Component`* class.

To handle an event the developer must create a class (or use an existing class) that implements the required event listener. The class must then define the body of the relevant methods to handle the events. The event listener must then be registered with the component using the appropriate method.

The following class diagram shows the relationships between the components and the event listeners.

[1] The 1.1 Java library still retains the compatibility with the 1.0 inheritance based model, but for the purpose of this discussion we will assume that the 1.0 model is not available in the 1.1 library.

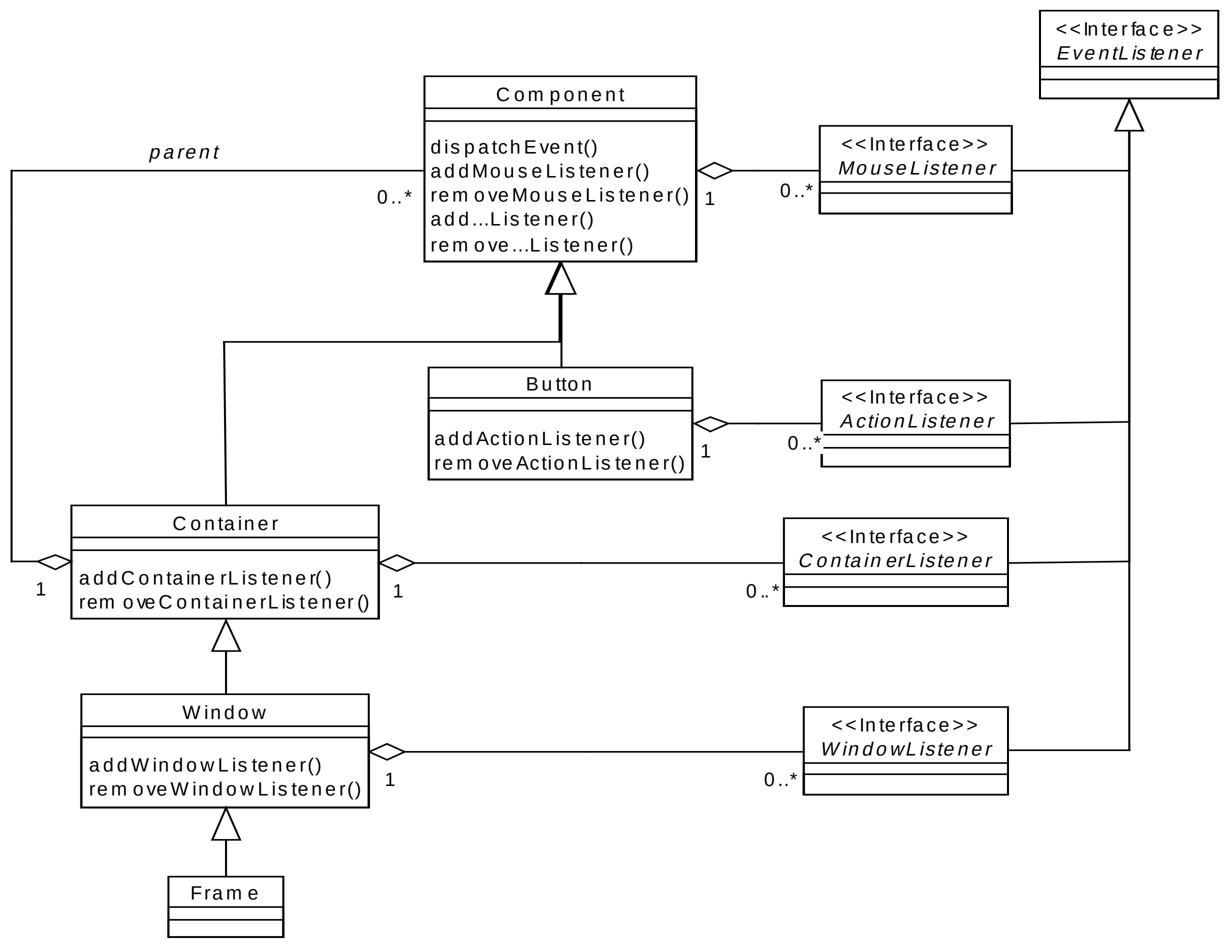

Figure 4 - Relationship between 1.1 AWT Components

The following example shows how to use event listeners to handle events.

```
public class DemoFrame extends Frame {
  public DemoFrame() {
    super("Event Handling Demo");
    setLayout(new FlowLayout());
    Button ok = new Button("OK");
    ok.addActionListener(OKListener);
    add(ok);
    Button cancel = new Button("Cancel");
    cancel.addActionListener(CancelListener);
    add(cancel);
    pack();
    show();
  }
}

class OKListener implements ActionListener {
  public OKListener () {
  }
```

continued…

```
  public void actionPerformed(ActionEvent e) {
    // Perform the continue action here
    return true; // The event has bee fully handled
  }
}

class CancelButton extends Button {
  public CancelButton() {
  }

  public void actionPerformed(ActionEvent e) {
    // Perform the cancel action here
  }
}
```

Code 3 - Event handling using 1.1 Event Model (Example 1)

3.1.1.3 Comparison & Summary

3.1.1.3.1 JDK 1.0 API

3.1.1.3.1.1 Advantages

1. All Java enabled web browsers, Java development environments support the 1.0 API and therefore the developer and user base would be larger.

3.1.1.3.1.2 Disadvantages

1. The code to handle the event notification must be duplicated for each component.
2. Adding new components requires a new subclass to be created with the event handling code.
3. It does not scale to large applications.

3.1.1.3.2 JDK 1.1 API

3.1.1.3.2.1 Advantages

1. Only one class is requires per type of event (i.e. *`ActionEvent`*, *`AdjustableEvent`*) rather than one per component.
2. Adding a new component does not require a new subclass to be created to handle the event notification, instead an *`EventListener`* can be registered with the component.
3. The method can be scaled to large applications that may change during execution.

3.1.1.3.2.2 Disadvantages

1. There are only two browsers that support the 1.1 API, the 'SunSoft Hot Java Browser' and the 'Microsoft Internet Explorer 4.0'.

3.1.2 Implementation

The implementation of the CSP event notification will use the JDK 1.1 API. This is because it reduces the duplication of code and enables the reuse of the event handlers for new components such as the new swing library, which is being introduced in the 1.2 API.

3.1.2.1 Event Notification

This section discusses the various stages the design of the general method of event notification using a *`Channel`* was developed.

All the examples in this section discus the implementation for a subclass of *`Button`*, the *`ActiveButton`* class. The examples are a simplified version of what was implemented; this serves to highlight the approach and any undesirable features. The diagram below shows the process diagram for an active component.

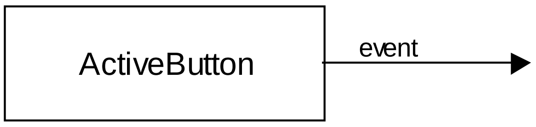

Figure 5 – Process Diagram – event notification

3.1.2.1.1 Passive Approach

The simplest approach is passive and does not require an extra *`CSProcess`*. In this approach, the constructor has an extra parameter `event` of type *`ChannelOutput`* that will be used to send the event notifications down. This parameter is stored as a `private` attribute in the class so it can be used by the other methods. As the class uses the 1.1 API it implements the *`ActionListener`* interface and defines the `actionPerformed(ActioneEvent)` method that will be invoked when the button is pressed. The constructor registers this class as an action listener using the `addActionListener(ActionListener)` method.

The body of the `actionPerformed(…)` method implements the event notification down the *`Channel`*. This is done by getting the action command name from the *`ActionEvent`* instance and writing this to the event *`Channel`*.

```
import java.awt.*;
import java.awt.event.*;
import jcsp.lang.*;

public class ActiveButton extends Button implements ActionListener {
  private ChannelOutput event;
  public ActiveButton(ChannelOutput event, String s) {
    super(s);
    this.event = event;
    addActionListener(this);
  }
                                                              continued…
  public synchronized void actionPerformed(ActionEvent e) {
    event.write(e.getActionCommand());
  }
}
```

Code 4 – Event Handling – Passive Approach

The main problem with this method is that the main Java event thread that is executing the `actionPerformed(…)` method is blocked until the process that is reading from the event *`Channel`* reads the request. As one of the design goals of using a *`Channel`* interface is to not stop further event notifications being blocked while the code for the event is being executed, this approach is not acceptable.

3.1.2.1.2 Active Approach With A Polling Loop

The next approach attempts to solve the problem of blocking the event thread. It is based on the implementation of the *`ButtonCh`* class developed by David Beckett.

The class implements the *`CSProcess`* interface and therefore is an active process and should be executed as part of a Parallel construct.

A private attribute `clicked` of type `boolean` is added to the class that indicates that the button has been pressed. Another private attribute `e` of type *`ActionEvent`* is also added to store a reference to the event. The `actionPerformed(…)` method will store the reference to the event in the `e` attribute and set `clicked` to true. The method is synchronised.

The `testAndResetClicked()` will return true if `clicked` is true and set `clicked` to false, this is used to provide the notification. The method is synchronised.

The body of the process in the `run()` method has an infinite loop that uses `testAndResetClicked()` to check if an event has occurred, if it has the action command from the event will be sent down the `event` *`Channel`*. In either case, the process will then sleep for a tenth of a second before looping.

```
import java.awt.*;
import java.awt.event.*;
import jcsp.lang.*;

public class ActiveButton extends Button
    implements CSProcess, ActionListener {
  private boolean clicked = false;
  private Channel event;
  private ActionEvent e;

  public ActiveButton(Channel event, String s) {
    super(s);
    this.event = event;
    addActionListener(this);
  }

                                                continued…
  public synchronized void actionPerformed(ActionEvent e) {
    this.e = e;
    clicked = true;
  }

  public synchronized boolean testAndResetClicked() {
    boolean result = clicked;
    clicked = false;
    return result;
  }

  public void run () {
    while (true) {
      if (testAndResetClicked()) {
        event.write(e.getActionCommand());
      }

      try {
        Thread.sleep(100);
      }
      catch (InterruptedException ie)
      {
      }
    }
  }
}
```

Code 5 – Event Handling – Active Approach with A Polling Loop

The main problem with this approach is that the body of the process has a polling loop that is executed 10 times a second. This has two undesirable effects. The first is it consumes processing power even when the button has not been pressed. The second is that only a maximum of 10 event notifications can be sent a second, this could be solved by decreasing the sleep time but this would then effect the performance.

3.1.2.1.3 Active Approach With wait/notify

The next approach builds on the previous one but removes the polling approach by using the wait/notify synchronisation on the object monitor.

The clicked attribute and the `testAndResetClicked()` method are removed.

The `actionPerformed(…)` method sets the `e` attribute to the event parameter and then invokes `notify()` on the monitor for this *`Object`*.

The `run()` method has an infinite loop that invokes `wait()` on the monitor for this *`Object`*. The method will block until the `notify()` method has been called. Upon waking up the process will output the `e` attribute down the event *`Channel`* and then loop.

NOTE: The `wait()` *method is wrapped in a try catch block that ignores the* *`InterruptedException`*. *This is safe, as this will never be raised when using the Parallel construct to execute the process.*

```
import java.awt.*;
import java.awt.event.*;
import jcsp.lang.*;

public class ActiveButton extends Button
    implements CSProcess, ActionListener {
  private ChannelOutput event;
  private ActionEvent e;

  public ActiveButton(ChannelOutput event, String s) {
    super(s);
    this.event = event;
  }

  public synchronized void actionPerformed(ActionEvent e) {
    this.e = e;
    notify();
  }

  public void run () {
    while (true) {
      try {
        wait();
      }
      catch (InterruptedException ie)
      {
      }
      event.write(e);
    }
  }
}
```

Code 6 – Event Handling – Active Approach with wait/notify

3.1.2.1.4 Active Approach with Channels

The final method came from a realisation of what was actually trying to be achieved here was a synchronised communication between two processes. The obvious mechanism available to perform this kind of synchronisation provided by the *`Channel`* classes developed as part of this package. This was so obvious it took four versions and about three months to notice it. This is however reassuring in the respect that the *`Channel`* does have a very practical use that simplifies a design.

The synchronisation required between the processes is for the event handling process in the *`ActiveButton`* to wait for the event thread to notify it when an event has happened and to pass it the event parameter to it. The implementation does not want to block while the event handling process is writing to the event *`Channel`* so that any notifications should be discarded in this case. This functionality can be implemented by using a *`Channel`* that has an *`OverWrittingBuffer`* that only stores one *`Object`* and overwrites this if it has not been read.

The implementation of the event handling is now simplified to have an extra attribute `eventNotify` that is an instance of an *`One2OneChannel`* with an *`OverWritingBuffer`* of size 1. The `actionPerformed(…)` method will write the *`ActionEvent`* instance down the `eventNotify` *`Channel`*. The process body reads from the `eventNotify` *`Channel`* and the writes the action command down the `event` *`Channel`*. Any further calls to `actionPerformed(…)` while the even handling process is writing will be discarded.

As the synchronisation is being performed by the *`Channel`*, it is safe to remove the method synchronisation.

NOTE: This does assume that the process is executed in one thread only and that the event thread is the only one to call the `actionPerformed(…)` method. As the actual implementation does not reveal a reference to the process or the method this cannot occur therefore it is safe.

```
import java.awt.*;
import java.awt.event.*;
import jcsp.lang.*;
import jcsp.util.*

public class ActiveButton extends Button implements CSProcess,
    ActionListener {
  private Channel eventNotify =
    new One2OneChannel(new OverWritingBuffer(1));
  private ChannelOutput event;

  public ActiveButton(ChannelOutput event, String s) {
    super(s);
    this.event = event;
  }

  public void actionPerformed(ActionEvent e) {
    eventNotify.write(e);
  }

  public void run () {
    while (true) {
      ActionEvent e = (ActionEvent)eventNotify.read();
      event.write(e.getActionCommand());
    }
  }
}
```

Code 7 – Event Handling – Active Approach with Channels

3.1.2.2 Class Design

After implementing the *`ActiveButton`* class, it became apparent that it would not be necessary to implement the event handling code for each component. Instead, it was possible to develop one event handler for each type of event. The subclasses of the components would then only need to create an instance of this class and register it as a listener and add it to the *`Parallel`* construct for the class (see section 3.1.3).

The rest of this section describes each of the event handlers for the different event types. There is one handler per *`EventListener`* interface provided in the `java.awt.event` package. If a new event Listener were added then a new event handler class would be required. The following table summarises the event handlers.

Class	**Listener**	**Event**	**Components**
ActionEventHandler	*ActionListener*	*ActionEvent*	*Button,* *List,* *MenuItem,* *TextField*
AdjustmentEventHandler	*AdjustmentListener*	*AdjustmentEvent*	*Scrollbar*
ItemEventHandler	*ItemListener*	*ItemEvent*	*Checkbox,* *CheckboxMenuItem,* *Choice,* *List*
TextEventHandler	*TextListener*	*TextEvent*	*TextArea,* *TextField*
ComponentEventHandler	*ComponentListener*	*ComponentEvent*	*Component*
ContainerEventHandler	*ContainerListener*	*ContainerEvent*	*Container*
FocusEventHandler	*FocusListener*	*FocusEvent*	*Component*
KeyEventHandler	*KeyListener*	*KeyEvent*	*Component*
MouseEventHandler	*MouseListener*	*MouseEvent*	*Component*
MouseMotionEventHandler	*MouseMotionListener*	*MouseMotionEvent*	*Component*
WindowEventHandler	*WindowListener*	*WindowEvent*	*Window*

In general, the event handlers for the semantic events send data that is associated with the event (such as the name of the button) down the `event` *`Channel`*. The event handlers for the low-level events send the event object down the `event` *`Channel`*.

3.1.2.2.1 ActionEventHandler

The *`ActionEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates an *`ActionEvent`*. The class implements the *`ActionListener`* interface so that it can be registered with any component that has an `addActionListener(ActionListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

The data that is sent down the `event` Channel is the action command for the event obtained using the `getActionCommand()` method on the *`ActionEvent`* instance for the event.

```
package jcsp.awt.event;
import java.awt.event.*
import jcsp.lang.*;
import jcsp.util.*;

public class ActionEventHandler implements CSProcess, ActionListener {
  private Channel eventNotify =
    new One2OneChannel(new OverWrittingBuffer(1));
  private ChannelOutput event;

  public ActionEventHandler(ChannelOutput event) {
    this.event  = event;
  }
                                                                  continiued…
  public void actionPerformed(ActionEvent e) {
    eventNotify.write(e);
  }

  public void run () {
    while (true) {
      ActionEvent e = (ActionEvent)eventNotify.read();
      event.write(e.getActionCommand());
    }
  }
}
```

Code 8 – ActionEventHandler

3.1.2.2.2 AdjustmentEventHandler

The *`AdjustmentEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates an *`ActionEvent`*. The class implements the *`AdjustmentListener`* interface so that it can be registered with any component that has an `addAdjustmentListener(AdjustmentListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

The data that is sent down the `event` *`Channel`* is an *`Integer`* representing the value of the adjustable component that generated the event. The value is obtained using the `getValue()` method on the *`AdjustmentEvent`* instance for the event.

The value is sent down the channel rather than the *`AdjustmentEvent`* instance to simplify the implementation of processes that perform some action based on this value. This does mean some information about the event is lost but generally, this information is not required.

```
package jcsp.awt.event;
import java.awt.event.*
import jcsp.lang.*;
import jcsp.util.*;

public class AdjustmentEventHandler
    implements CSProcess, AdjustmentListener {
  private ChannelOutput event;
  private Channel eventNotify =
    new One2OneChannel(new OverWritingBuffer(1));

  public AdjustmentEventHandler(Channel event) {
    this.event  = event;
  }

  public void adjustmentValueChanged(AdjustmentEvent e) {
    eventNotify.write(e);
  }

                                                       continiued…
  public void run () {
    while (true) {
      AdjustmentEvent e = (AdjustmentEvent)eventNotify.read();
      event.write(new Integer(e.getValue()));
    }
  }
}
```

Code 9 – AdjustmentEventHandler

3.1.2.2.3 ItemEventHandler

The *ItemEventHandler* class is a process that can be used to send event notifications down the `event` *Channel* when a component generates an *ItemEvent*. The class implements the *ItemListener* interface so that it can be registered with any component that has an `addItemListener(ItemListener)` method. The class implements the *CSProcess* so the process can be run as part of a *Parallel* construct.

The data that is sent down the `event` *Channel* consists of two separate objects. The first is a *Boolean* that either has the value `Boolean.TRUE` or `Boolean.FALSE` depending on whether the event was for an item being selected or deselected. The second is an object that was selected or deselected. The item is obtained using the `getItem()` method on the *ItemEvent* instance for the event.

As the channel communication sends two objects per event, the *Channel* should only be an *One2OneChannel*, otherwise the objects may be read by different processes thus causing non-determinism.

```
package jcsp.awt.event;
import java.awt.event.*
import jcsp.lang.*;
import jcsp.util.*;

public class ItemEventHandler
    implements CSProcess, ItemListener {
  private ChannelOutput event;
  private Channel eventNotify =
    new One2OneChannel(new OverWritingBuffer(1));

  public ItemEventHandler(ChannelOutput event) {
    this.event  = event;
  }

  public void itemStateChanged(ItemEvent e) {
    eventNotify.write(e);
  }
                                                 continiued…
  public void run () {
    while (true) {
      ItemEvent e = (ItemEvent)eventNotify.read();
      if (e.getStateChange() == ItemEvent.SELECTED) {
        event.write(Boolean.TRUE);
      }
      else {
        event.write(Boolean.FALSE);
      }
      event.write(e.getItem());
    }
  }
}
```

Code 10 – ItemEventHandler

3.1.2.2.4 TextEventHandler

The *`TextEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates a *`TextEvent`*. The class implements the *`TextListener`* interface so that it can be registered with any component that has an `addTextListener(TextListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

The data that is sent down the `event` *`Channel`* is the current text value from the component. The text is obtained using the `getText()` method on the source component from the *`TextEvent`* instance for the event.

NOTE: *This is currently the way it is implemented, this in not necessarily the best way to handle text events as a lot of information about the event is lost and it may be useful to software constructors. A future version may send the source component down the Channel.*

```
package jcsp.awt.event;
import java.awt.event.*
import jcsp.lang.*;
import jcsp.util.*;

public class TextEventHandler
    implements CSProcess, TextListener {
  private ChannelOutput event;
  private Channel eventNotify =
    new One2OneChannel(new jcsp.util.OverWritingBuffer(1));

  public TextEventHandler(ChannelOutput event) {
    this.event = event;
  }

  public void textValueChanged(TextEvent e){
    eventNotify.write(e);
  }
                                                 continiued…
  public void run () {
    TextEvent e = (TextEvent)eventNotify.read();
    event.write(((TextComponent)e.getSource()).getText());
  }
}
```

Code 11 – TextEventHandler

3.1.2.2.5 ComponentEventHandler

The *ComponentEventHandler* class is a process that can be used to send event notifications down the event *Channel* when a component generates a *ComponentEvent*. The class implements the *ComponentListener* interface so that it can be registered with any component that has an addComponentListener(ComponentListener) method. The class implements the *CSProcess* so the process can be run as part of a *Parallel* construct.

NOTE: The Component class has an addComponentListener(…) *method. Therefore, all components can generate a ComponentEvent.*

The data that is sent down the event *Channel* is the *ComponentEvent*.

3.1.2.2.6 ContainerEventHandler

The *ContainerEventHandler* class is a process that can be used to send event notifications down the event *Channel* when a component generates a *ContainerEvent*. The class implements the *ContainerListener* interface so that it can be registered with any component that has an addContainerListener(ContainerListener) method. The class implements the *CSProcess* so the process can be run as part of a *Parallel* construct.

NOTE: The Container class has an addContainerListener(…) *method.*

The data that is sent down the event *Channel* is the *ContinerEvent*.

3.1.2.2.7 FocusEventHandler

The *FocusEventHandler* class is a process that can be used to send event notifications down the event *Channel* when a component generates a *FocusEvent*. The class implements the *FocusListener* interface so that it can be registered with any component that has an addFocusListener(FocusListener) method. The class implements the *CSProcess* so the process can be run as part of a *Parallel* construct.

NOTE: The Component class has an addFocusListener(…) *method.*

The data that is sent down the event *Channel* is the *FocusEvent*.

3.1.2.2.8 KeyEventHandler

The *`KeyEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates a *`KeyEvent`*. The class implements the *`KeyListener`* interface so that it can be registered with any component that has an `addKeyListener(KeyListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

NOTE: The `Component` *class has an* `addKeyListener(…)` *method.*

The data that is sent down the `event` *`Channel`* is the *`KeyEvent`*.

3.1.2.2.9 MouseEventHandler

The *`MouseEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates a *`MouseEvent`*. The class implements the *`MouseListener`* interface so that it can be registered with any component that has an `addMouseListener(MouseListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

NOTE: The `Component` *class has an* `addMouseListener(…)` *method.*

The data that is sent down the `event` *`Channel`* is the *`MouseEvent`*.

3.1.2.2.10 MouseMotionEventHandler

The *`MouseMotionEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates a *`MouseMotionEvent`*. The class implements the *`MouseMotionListener`* interface so that it can be registered with any component that has an `addMouseMotionListener(MouseMotionListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

NOTE: The `Component` *class has an* `addMouseMotionListener(…)` *method.*

The data that is sent down the `event` *`Channel`* is the *`MouseMotionEvent`*.

3.1.2.2.11 WindowEventHandler

The *`WindowEventHandler`* class is a process that can be used to send event notifications down the `event` *`Channel`* when a component generates a *`WindowEvent`*. The class implements the *`WindowListener`* interface so that it can be registered with any component that has an `addWindowListener(WindowListener)` method. The class implements the *`CSProcess`* so the process can be run as part of a *`Parallel`* construct.

NOTE: The `Widnow` *class has an* `addWindowListener(…)` *method.*

The data that is sent down the `event` *`Channel`* is the *`WindowEvent`*.

3.2 Configuration

3.2.1 AWT Configuration

In Java components can have their properties (i.e. colour, size) configured by invoking methods on an instance of the component. Most of these methods are defined in the *`Component`* class and therefore can be used for all the components. The subclasses of *`Component`* may define extra methods to configure the properties specific to that component. For example, the *`Scrollbar`* class provides methods to change the scale and value of the *`Scrollbar`*.

It is possible that several threads try to invoke these methods on an instance *`Component`* at the same time. The implementations for these methods update the state and during execution, the state may be in an inconsistent state. The methods are synchronised so that only one method may be updating the state at any time.

The following table summarises all of the configuration methods provided by each class for configuration.

`Component`	
`setBackground(Color)`	Sets the background color of this component.
`setBounds(int, int, int, int)`	Moves and resizes this component.
`setBounds(Rectangle)`	Moves and resizes this component to conform to the new bounding rectangle r.
`setCursor(Cursor)`	Set the cursor image to a predefined cursor.
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setFont(Font)`	Sets the font of this component.
`setForeground(Color)`	Sets the foreground color of this component.
`setLocale(Locale)`	Sets the locale of this component.
`setLocation(int, int)`	Moves this component to a new location.
`setLocation(Point)`	Moves this component to a new location.
`setName(String)`	Sets the name of the component to the specified string.
`setSize(Dimension)`	Resizes this component so that it has width d.width and height d.height.
`setSize(int, int)`	Resizes this component so that it has width width and height.
`setVisible(boolean)`	Shows or hides this component depending on the value of parameter b.
`Button`	
`setActionCommand(String)`	Sets the command name for the action event fired by this button.
`setLabel(String)`	Sets the button's label to be the specified string.
`Canvas`	
`CheckBox`	
`setCheckboxGroup(CheckboxGroup)`	Sets this check box's group to be the specified check box group.
`setLabel(String)`	Sets this check box's label to be the string argument.
`setState(boolean)`	Sets the state of this check box to the specified state.

CheckboxMenuItem	
`setLabel(String)`	Sets this check box's label to be the string argument.
`setState(boolean)`	Sets the state of this check box to the specified state.
Choice	
`add(String), addItem(String)`	Adds an item to this Choice menu.
`insert(String, int)`	Inserts the item into this choice at the specified position.
`remove(int)`	Removes an item from the choice menu at the specified position.
`remove(String)`	Remove the first occurrence of item from the Choice menu.
`removeAll()`	Removes all items from the choice menu.
`select(int)`	Sets the selected item in this Choice menu to be the item at the specified position.
`select(String)`	Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.
Label	
`setAlignment(int)`	Sets the alignment for this label to the specified alignment.
`setText(String)`	Sets the text for this label to the specified text.
List	
`add(String), addItem(String)`	Adds an item to this scrolling list.
`add(String, int), addItem(String, int)`	Adds the specified item to the scrolling list at the specified position.
`remove(int)`	Removes an item from the scrolling list at the specified position.
`remove(String)`	Remove the first occurrence of item from the scrolling list.
`removeAll()`	Removes all items from the scrolling list.
`replaceItem(String, int)`	Replaces the item at the specified index in the scrolling list with the new string.
`select(int)`	Sets the selected item in this Choice menu to be the item at the specified position.
`setMultipleMode(boolean)`	Sets the flag that determines whether this list allows multiple selections.
Scrollbar	
`setBlockIncrement(int)`	Sets the block increment for this scroll bar.
`setMaximum(int)`	Sets the maximum value of this scroll bar.
`setMinimum(int)`	Sets the minimum value of this scroll bar.
`setOrientation(int)`	Sets the orientation for this scroll bar.
`setUnitIncrement(int)`	Sets the unit increment for this scroll bar.
`setValue(int)`	Sets the value of this scroll bar to the specified value.
`setValues(int, int, int, int)`	Sets the values of four properties for this scroll bar.
`setVisibleAmount(int)`	Sets the visible amount of this scroll bar.
TextComponent	
`selectAll()`	Selects all the text in this text component.
`setCaretPosition(int)`	Sets the position of the text insertion caret for this text component.
`setEditable(boolean)`	Sets the flag that determines whether this text component is editable.
`setSelectionEnd(int)`	Sets the selection end for this text component to the specified position.

`setSelectionStart(int)`	Sets the selection start for this text component to the specified position.
`setText(String)`	Sets the text that is presented by this text component to be the specified text.
TextArea	
`append(String)`	Appends the given text to the text area's current text.
`insert(String, int)`	Inserts the specified text at the specified position in this text area.
`replaceRange(String, int, int)`	Replaces text between the indicated start and end positions with the specified replacement text.
`setColumns(int)`	Sets the number of columns for this text area.
`setRows(int)`	Sets the number of rows for this text area.
TextField	
`setColumns(int)`	Sets the number of columns for this text area.
`setEchoChar(char)`	Sets the echo character for this text field.

The classes also provide methods to obtain the current state of the component. These methods are also synchronised so that the state is not changed while it is being executed.

3.2.2 Implementation

The implementation of configuration for the active components currently does not provide the same level of configuration as provided by the AWT. The inspection of the state has also not been implemented. The reasons for the non-completeness of functionality were mainly due to time constraints and the desire for simplicity of the interface. In future versions the library will be extended to provide the same level of configuration.

Several methods can be used to implement the configuration. Each of these has a process within the component that will read from one or more channels and update the state of the component based on the value received. Two methods were considered one Channel per component or one Channel per type of configuration.

Only a subset of the configuration is provided, the table below summarises what has been implemented.

NOTE: The original configuration provided by the methods on the components can also be used but their use should be restricted to when the components are constructed.

Component	
Button	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setLabel(String)`	Sets the button's label to be the specified string.
Canvas	
CheckBox	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setLabel(String)`	Sets this check box's label to be the string argument.
`setState(boolean)`	Sets the state of this check box to the specified state.

CheckBoxMenuItem	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setLabel(String)`	Sets this check box's label to be the string argument.
`setState(boolean)`	Sets the state of this check box to the specified state.
Choice	
`select(int)`	Sets the selected item in this Choice menu to be the item at the specified position.
`select(String)`	Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
Label	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setText(String)`	Sets the text for this label to the specified text.
List	
`select(int)`	Sets the selected item in this Choice menu to be the item at the specified position.
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
Scrollbar	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setValue(int)`	Sets the value of this scroll bar to the specified value.
TextComponent	
`setText(String)`	Sets the text that is presented by this text component to be the specified text.
TextArea	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`append(String)`	Appends the given text to the text area's current text.
TextField	
`setEnabled(boolean)`	Enables or disables this component, depending on the value of the parameter b.
`setText(String)`	Sets the text that is presented by this text component to be the specified text.

3.2.2.1 Configuration

3.2.2.1.1 One Channel per Component

In this approach, one event channel is used to configure the component. The type of configuration performed depends upon the type of the object sent down the channel. The following diagram shows an active component with one configuration channel.

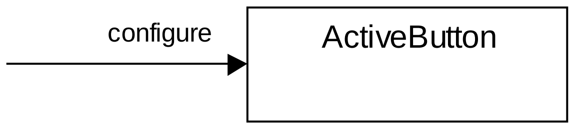

Figure 6 – Process Diagram – single channel configuration

The following example shows how configuration using a single channel can be implemented.

```
import java.awt.*;
import jcsp.lang.*;
import jcsp.util.*

public class ActiveButton extends Button implements CSProcess {
  private ChannelInput configure;

  public ActiveButton(ChannelInput configure, String s) {
    super(s);
    this.configure = configure;
  }

  public void run () {
    while (true) {
      Object message = configure.read();
      if (message instanceof String) {
        setLabel((String)message);
      }
      else if (message instanceof Boolean) {
        if (message == Boolean.TRUE) {
          setEnabled(true);
        }
        else if (message == Boolean.FALSE) {
          setEnabled(false);
        }
        else {
        }
      }
    }
  }
}
```

Code 12 – Configuration – Single Channel

In the constructor the channel is passed into the component and stored in a private attribute. The component implements the *`CSProcess`* interface and defines the `run()` method to perform the configuration.

The `run()` method has an infinite loop that reads one value from the `configure` *`Channel`* into a local variable of type *`Object`*. The method then has an `if else if` ladder that performs the configuration based on either the type of the object or by comparing it against a know constant. In the case of the *`ActiveButton`* if the value is an instance of *`String`* the `setLabel(…)` method will be invoked with the value. If the value is an instance of *`Boolean`* one of three things can happen. If it is the instance `Boolean.TRUE` the *`Button`* will be enabled, the instance `Boolean.FALSE` the *`Button`* will be disabled, otherwise it will be ignored. If the value is of any other type it will be ignored.

NOTE: the constants `Boolean.TRUE` *and* `Boolean.FALSE` *are used to save on garbage collection and this frees up other* `Boolean` *values to be used for other configuration.*

The disadvantage with this approach is that it is only possible to perform one type of configuration per type of object sent down the channel. For example, two methods have *`Color`* as a parameter `setForeground(Color)` and `setBackground(Color)`. If a *`Color`* was received on the channel what type of configuration should be performed? This can be solved by either wrapping these values up in a protocol class that has an identifier of the type of configuration required or a constant value could be sent before the value. Also if configuration requires more than one value these will need to be wrapped up or sent in sequence down the channel.

If the values were wrapped up in a protocol this would complicate the interface. If the values were sent in sequence, the configuration would be restricted to using an *`One2OneChannel`* otherwise the order of the messages could not be guaranteed (this would not be a problem if claims[2] could be made on channels).

3.2.2.1.2 One Channel per configuration

In this approach, one event channel per type of configuration is used to configure the component. The type of configuration performed depends upon which channel the message is received. The following diagram shows an active component with two configuration channels.

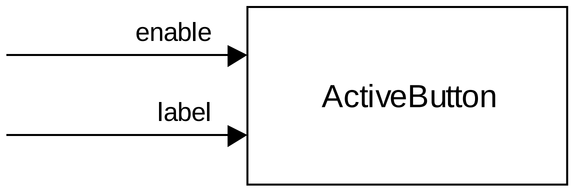

Figure 7 – Process Diagram – multiple channel configurations

The following example shows how configuration using multiple channels can be implemented.

[2] The concept of claiming a channel was introduced in OCCAM3 for shared channels. Once a channel has been claimed it can only be used by that channel for output (grant is used to claim for input). The messages can then be guaranteed to arrive in the order sent. This will be added to the JCSP library in future releases.

```
import java.awt.*;
import jcsp.lang.*;
import jcsp.util.*

public class ActiveButton extends Button implements CSProcess {
  private ChannelInput enable;
  private ChannelInput label;

  public ActiveButton(ChannelInput enable, ChannelInput label,
                      String s) {
    super(s);
    this.enable = enable;
    this.label = label;
  }

  public void run () {
    Alternative alt = new Alternative();
    ChannelInput chans = {enable, label};
    while (true) {
      switch (alt.select(chans)) {
        case 0:
          Boolean b = (Boolean)enable.read();
          setEnabled(b.booleanValue());
        break;
        case 1:
          String l = (String)label.read();
          setLabel(l);
        break;
      }
    }
  }
}
```

Code 13 – Configuration – Multiple Channels

In the constructor the channels used to configure the label and to enable the component are passed into the component and stored in private attributes. The component implements the *`CSProcess`* interface and defines the `run()` method to perform the configuration.

The `run()` method has an infinite loop. The loop alternates on the `enable` and `label` channels. If the `enable` channel is selected a *`Boolean`* is read from the channel and the `setEnable(Boolean)` method is invoked with the `booleanValue()` of the *`Boolean`*. If the `label` channel is selected a *`String`* is read from the channel and the `setLabel(String)` method is invoked with the value.

NOTE: If any other type than the expected type is received on the channels a `ClassCastException` will be raised and the process will terminate abnormally.

The disadvantage with this approach is that the constructor may become very large if many different types of configurations are performed. This also means that the software constructor will have to create one *`Channel`* for each configuration type even if it is not going to be used. This can be solved by either defining the channels as attributes on the component, providing `setXXXChannel(..)` methods for each configuration rather than a constructor or the constructor expects an array of configuration channels.

This approach also suffers from the problem of multiple data configuration that the single channel method suffers from.

3.2.2.2 Class Design

The implementation uses the one channel per component approach as it makes the interface easier to use. This does however restrict the configurations that can be performed (this is why not all the configurations are implemented).

As each component has different configuration requirements each component must implement the required configuration as a process dedicated to that component (sub classes may be able to reuse the configuration process).

NOTE: There are too many classes to describe the implementation of them. Refer to the on-line documentation for each component to see what configuration is provided.

3.3 Active Components

This section discusses how the event handling and configuration described above can be joined together to make the active components. All the components use the same approach so the discussion will only consider the *`ActiveButton`*.

The following diagram shows the external channel interface to a component along with the internal processes.

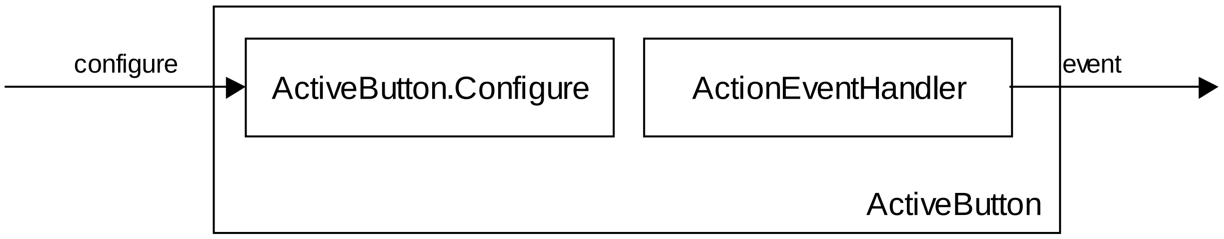

Figure 8 – Process diagram – active components

3.3.1 Implementation

Each component implements the *`CSProcess`* interface. The `run()` method invokes `run()` on the `par` attribute (of type *`Parallel`*) that contains all the processes created by this component.

The constructor of the component is passed the `configure` and `event` channels. If the `event` channel is not `null` a new *`ActionEventHandler`* process is created, registered as an *`ActionListener`* with this component and added to the `par` construct. If the `configure` channel is not `null` a new *`Configure`* process will be created and added to the `par` construct.

The following example shows the *`ActiveButton`* is implemented (all components are similar).

```
package jcsp.awt;
import java.awt.*;
import jcsp.lang.*;
import jcsp.awt.event.*;

public class ActiveButton extends Button implements CSProcess {
  protected Parallel par;

  public ActiveButton(ChannelInput configure, ChannelOutput event,
                      String s) {
    super(s);
    par = new Parallel();

    if (event != null) {
      ActionEventHandler handler = new ActionEventHandler(event);
      addActionListener(handler);
      par.addProcess(handler);
    }
```
continued…

```
    if (configure != null) {
      par.addProcess(new Configure(configure));
    }
  }

   public void run() {
    par.run();
  }

  // Definition of other methods

  // Definition of configure class
}
```

Code 14 – ActiveButton class

Each component defines a nested class `Configure` that implements the `CSProcess` interface and contains the code in the `run()` method to configure the component upon receiving values. The following code shows how this is implemented for the `ActiveButton` component.

```
  protected class Configure implements CSProcess {
    private ChannelInput configure;

    public Configure(ChannelInput configure) {
      this.configure = configure;
    }

    public void run() {
      while (true) {
        Object message = configure.read();
        if (message instanceof String) {
          setLabel((String)message);
        }
        else if (message instanceof Boolean) {
          if (message == Boolean.TRUE) {
            setEnabled(true);
          }
          else if (message == Boolean.FALSE) {
            setEnabled(false);
          }
        }
      }
    }
  }
}
```

Code 15 – ActiveButton – Configure class

It is also necessary to provide a mechanism to have a channel interface to the low-level events. As this is not always required when constructing a component one of the `addXXXEventChannel(…)` methods can be used to create a new event handler for the component with the `Channel` specified. These methods must be invoked **before** the component is executed as part of a `Parallel` construct, otherwise the handler will not be executed. The following code shows how these methods are implemented.

NOTE: Windows and containers will have extra methods to listen for those events.

The methods check to see if the `Channel` is not `null`. Then it creates a new event handler for that event type. The event handler is registered as an event listener with this component. Finally the process is added to the `Parallel` construct for the component.

```
  public void addComponentEventChannel(ChannelOutput event) {
    if (event != null) {
      ComponentEventHandler handler =
        new ComponentEventHandler(event);
      addComponentListener(handler);
      par.addProcess(handler);
    }
  }

  public void addFocusEventChannel(ChannelOutput event) {
    if (event != null) {
      FocusEventHandler handler = new FocusEventHandler(event);
      addFocusListener(handler);
      par.addProcess(handler);
    }
  }

  public void addKeyEventChannel(ChannelOutput event) {
    if (event != null) {
      KeyEventHandler handler = new KeyEventHandler(event);
      addKeyListener(handler);
      par.addProcess(handler);
    }
  }

  public void addMouseEventChannel(ChannelOutput event) {
    if (event != null) {
      MouseEventHandler handler = new MouseEventHandler(event);
      addMouseListener(handler);
      par.addProcess(handler);
    }
  }

public void addMouseMotionEventChannel(ChannelOutput event) {
  if (event != null) {
    MouseMotionEventHandler handler =
      new MouseMotionEventHandler(event);
    addMouseMotionListener(handler);
    par.addProcess(handler);
  }
}
```

Code 16 - ActiveButton – addXXXEventChannel() methods

Bibliography

[1] Paul. D. Austin. 1998, *’Design of JCSP Language classes’*, University of Kent Canterbury
[2] JavaSoft. 1997, *'JDK™ 1.1 Documentation'*, Sun Microsystems.
`http://www.javasoft.com/products/jdk/1.1/docs/`
[3] C. A. R. Hoare. 1985, *'Communicating Sequential Processes'*, Prentice Hall.
[4] Martin Fowler with Kendall Scott. 1997, *'UML Distilled'*, Addison Wesley, ISBN 0-201-32563-2
[5] CGS-THOMSON Microelectronics Ltd. 1995, *'OCCAM 2.1 reference manual'*, Prentice Hall International (UK) Ltd.
[6] Gamma, Helm, Johnson, Vlissides. 1995, *Design Patterns (Elements of Reusable Object-Oriented Software)*, Addison-Wesley