An Extensible Trainable Classification Workflow Management System Demonstrating Hand Gesture Recognition

A dissertation submitted in partial fulfilment of the requirements for the MSc in Computer Science

by Samuel Wright

Department of Computer Science and Information Systems
Birkbeck College, University of London

September 2013

Abstract

We review various 2D camera-based hand gesture recognition strategies and demonstrate their applicability to being used in a workflow paradigm, and note that most strategies implement supervised learning and therefore large labelled datasets are prerequisite to their development. We see that the choice of strategy often depends on characteristics of the image taken (e.g. background complexity), and suggest supervised learning as a valid technique to select which of the available strategies should be used based on the success rates of similar training images. We demonstrate that this can be implemented in a workflow paradigm with some modifications to the way data is processed, and showcase our prototype workflow system by implementing a simple hand gesture recognition strategy.

1 Background

1.1 Supervised Learning

Supervised learning is a branch of machine learning which deals with inferring functions that map an input dataset to a corresponding output dataset. In machine learning-parlence, this is known as a labelled (or tagged) dataset, where the label (or tag) is the output data. If the output data are a set of classes of which each input datum is a member of exactly one, then the function inferred is known as a classifier. Once trained, it can be given an unseen input datum and determine to which class it belongs.

The manual alternative to supervised learning is for a human to form a general understanding of the input data, then use intuition to produce a set of computable rules to classify it. Forming a general understanding of the input data can be difficult or even impossible, whereas a supervised learning algorithm needs only to infer the mapping between the input and output data, allowing it to grasp complex relationships. In other words, the algorithm solves only as much of the problem as is needed.

1.2 Scientific Workflow Systems

Workflows are abstract sequences of units of work that define the order of execution of, and flow of data between, those units. They have become widely used in scientific research [1] for their ability to manage complicated and concurrent processing of large datasets in a simple graphical user interface, allowing users to apply their domain knowledge to the workflow's design without concerning themselves with low-level execution details such as how to distribute the data and tasks across a network of computers.

The high-level facade the workflow paradigm provides allows the user to easily compare the effectiveness of different approaches, and perform high-level automations (such as parameter sweeping [2]) in an understandable way. By integrating validation into the workflow the user can easily detect problems before execution, which reduces the computational cost of implementing and testing an approach (which can be considerable for large datasets).

Here we briefly review some available scientific workflow systems.

1.2.1 Kepler

The Kepler scientific workflow system [3] is a popular tool designed for scientists to use for processing raw data from experiments. It can be used to orchestrate distributed data access and task execution, and emphasises workflow reusability.

1.2.2 Orange

Orange is a workflow-based data mining and exploration software suite [4], which seeks to simplify the use of various machine learning and data mining algorithms to allow its users to intuitively explore and find reason in large distributed heterogeneous datasets.

1.2.3 Taverna Workbench

Taverna Workbench is a workflow system designed specifically for the field of bioinformatics [5] and primarily manages and executes tasks through open web interfaces, making it easily extended.

1.3 Human Computer Interaction (HCI)

HCI is a rich and broad field due to the innumerable tasks a computer can perform and the variety of information required by and from the user for each. Concepts such as direct manipulation of graphical objects and gesture recognition have been heavily researched since the 1960s [6], and have led to the development and general adoption of mice, touchpads, and touchscreens.

There are numerous use cases where such devices are unnatural or inconvenient, such as interacting with a 3D virtual environment or interpreting sign language [7]. Interpreting hand gestures can provide a more natural interface [7], and a data-rich input given that humans have fine control of the 27 degrees of freedom each hand offers [8]. As such, hand gesture recognition has become a prominent and exciting subfield of HCI.

1.4 Applications of Hand Gesture Recognition.

For a gesture recognition system to be successful it must be able to recognise enough gestures to sufficiently serve a compelling purpose, however this functionality will only be apparent if the system is adequately usable [9]. This means it must be robust enough to reliably recognise gestures regardless of the complexity of the background and lighting conditions, operate in real-time with a minimum of lag, and be tolerant of user error [10]. The matching of gestures with their functionality must also be obvious to allow for easy learning and adoption of the system [11].

To understand what is meant by a hand gesture in the field of HCI, we now examine the existing applications of hand gesture recognition (from [10]).

1.4.1 Virtual Reality

Hand gestures are used as a means to manipulate 3D objects in software as if the objects were real (for example, transforming complex molecules as a means to better understand their properties [12]). Online, dynamic, 3D gestures are required in order to manipulate the objects (e.g. rotation, translation, and scaling).

1.4.2 Robotics and telepresence

Hand gestures are used to control remote objects through a computerised intermediary (for example, a surgeon is able to control a laparoscopic camera via a computer using hand gestures to allow for fine control [13]). The gestures often used are similar to those used in virtual reality applications.

1.4.3 Desktop and tablet applications

Hand gestures are used to manipulate standard PC applications without the use of a mouse (for example, surgeons are able to directly control where to zoom in on images taken during an endoscopy without the need to operate a mouse or relay instructions via an assistant [14]). All manner of gestures (online or offline, dynamic or static, 2D or 3D) could find compelling uses, depending on the specific application and task at hand.

1.4.4 Gaming

Hand gestures are used to control in-game characters or issue commands. The controller for the Nintendo Wii contains a triaxial accelerometer, allowing for in-game control of hand-held objects such as swords. The Xbox Kinect contains a set of cameras that can track hand motions in 3D, allowing for direct interaction with the in-game environment. As with desktop applications, the types of gestures required depend on the specifics of the game.

1.4.5 Sign language

Hand gestures (alongside arm and face gestures) represent a language that can be interpreted as commands or translated to a written or spoken language. The use of computers can, for example, help people learn sign language by interpreting and verifying their actions (for example, as part of a game [15]). The gestures are complex, 3D and dynamic.

1.5 Motivation

With the rise in popularity of capacitive touchpads, multitouch gestures are becoming more popular. There are a number of software packages (e.g. BetterTouchTool [16]) which allow the user to map multitouch gestures to actions like window management (e.g. maximise the current window), control media, or simulate keyboard button presses. The benefit of such a system is that each user can develop their own use cases. To our knowledge, there is no such equivalent software for hand gestures detected using a webcam.

This is likely due to the complexity of hand gesture recognition. There are many elements to a gesture recognition system (as described in the next section) and often the best choice of elements is dependent on the use case. There currently does not exist a framework for evaluating and comparing different strategies - even academics who publish their ideas about new strategies will only compare against a handful of others in single-use scripts.

The motivation for this project was to develop such framework using the workflow paradigm, but tailored to problems such as hand gesture recognition; specifically those with solutions that have to run on a local computer for performance and latency reasons, and involve multiple stages of optimisation using supervised learning. The next section details gesture recognition algorithms and their applicability to this criteria.

To demonstrate the usefulness of this framework, we then implemented a simple hand gesture recognition strategy inside it.

2 Gesture Recognition Algorithms

This section details various gesture recognition strategies, to demonstrate that any strategy involves multiple steps which require both optimising by hand (by seeing the result of each step in realtime) and automated optimisation (by performing parameter sweeps). In addition we hope to show that some circumstances find certain strategies more effective than others, demonstrating the need to intelligently choose strategies in realtime as the workflow executes.

The general workflow employed is data capture, feature extraction, then gesture classification.

2.1 Data Capture

There are two categories of sensors for capturing hand gestures: hand-mounted and vision-based [10] which we now discuss.

2.1.1 Hand-mounted sensors

Hand-mounted sensors (for example those found on a "data glove" [17]) provide each digit's joint orientations (either through mechanical or optical sensors) in real-time. While accurate and reliable, these require the user to wear bulky equipment which inhibits their adoption for general use. They have proven useful in learning sign languages [15], where the gestures are complex enough for the glove's reliability and accuracy to be useful, and are used for long-enough periods of time to make them worthwhile.

2.1.2 Vision-based sensors

This approach does not require the user to wear any sensors, allowing for a more natural and convenient input of data (although not entirely natural, since the gestures must be made within the camera's field of view). Hands have proven to be notably difficult to detect and interpret from visual data [18] because they are homogenous in colour and texture, lack static features (compared to a face, for example) and have many degrees of freedom. Depth-sensing [19] or IR cameras [20] can be used to separate the hand from other objects while providing extra data which can be used to increase the fidelity of the features extracted. Such cameras are still expensive and are not as widely available as 2D cameras.

2.2 Feature Extraction

If the input device is a data glove, the data it captures (i.e. joint orientations for all digits) can be immediately used as features. If instead a camera is used, the data it captures requires substantial processing, typically involving hand detection, pose estimation and hand tracking [21]. The correct method of feature acquisition to use in a gesture recognition system is a function of its desired application, the environment in which it will be used, and the notion of what is reasonable to ask of the user in terms of money and preparation time.

2.2.1 Hand detection

Detecting which pixels in a 2D image make up a hand presents a number of difficulties. The hand's silhouette is complicated and dynamic, and its colour is dependent on the specific user's skin colour and the lighting conditions. A hand's colour is relatively uniform throughout gesturing (as opposed to its edges), so is a good and computationally-cheap initial step in hand detection.

Skin colours and non-skin colours can be separated using classifiers working in the hue-saturation-intensity (HSI) colour-space [22] and variations in detected colour due to lighting can be circumvented using colour-space statistics [23]. Face-detection (which is easily done due to the face's static shape) can be used to track the subject's skin colour more accurately [24]. More advanced learning-based techniques also exist, for example using a self-organising map to reduce the number of dimensions of an image's colour space as a means of classifying skin-coloured objects [25].

Once the skin-coloured areas of an image have been identified, one could simply mask the rest of the image. However, this might introduce noise into the hand's sub-image if there were abnormally dark shadows (i.e. a false negative) or it might find small skin-coloured areas outside of the hand (i.e. a false positive). An alternative approach is to create a image where the intensity of each pixel is the scaled probability of the corresponding pixel from the camera being skin-coloured, then using an algorithm such as MSER [23] which attempts to find large regions of interest (in this case, large skin-coloured objects). Many algorithms exist which find regions of interest, though MSER has been shown to be one of the most effective [26].

Detecting skin colours will also pick up faces and other body parts, so more discrimination is required for such complex backgrounds. A workaround would be to require the user to make a pre-determined hand shape (e.g. an open palm facing the camera) before performing their gesture, which could be detected by fitting a simplistic model of the hand's shape to the image [27]. Such a workaround adds inconvenience to using a gesture recognition system, so should be avoided if possible. Facial recognition can again be used to avoid confusion with a hand, even when the hand partially occludes the face [28]. Smith et al [29] have proposed an interesting method that uses an analogue from physics (force and energy fields) to detect regions of complexity in an image such as a hand.

Another popular [30, 31, 32, 33] method decomposes the image (which can have a complex background) into Haar-like features (which describe local image-intensity distributions across the image) before being passed to a set of weak classifiers that have been trained on a large set of hand images, and together form a strong classifier for detecting the presence of a hand. The AdaBoost algorithm [34], which improves the performance of the classifier, is often used. This approach has shown promise with small vocabularies [31] but training large vocabularies is computationally expensive [35]. The algorithm used to generate Haar-like features [30] relies on using integral images [36] which are scale invariant, so generating the Haar-like features on different scales does not require recalculating the integral images, making this a very efficient technique. Classifying the features is done in a "cascade" [30] of weak classifiers, which allows for easy parallelisation of the fail-fast classifiers, so very little time is wasted processing parts of the image that don't contain a hand.

2.2.2 Pose estimation

After the hand has been detected, a feature set must be extracted from it that will be used to recognise the gesture being made. There are two approaches [8], model-based and appearance-based, which we now discuss in turn.

Using a 3D computerised model of a hand which faithfully mimics the available ranges of motion (i.e. a kinematic model) a variety of poses and viewpoints can be tried to find the 2D projection that matches the image of the hand from the sensor [37]. Any number of metrics can then be taken from the model hand and used as features for gesture recognition. This provides as much data as the data glove, but requires the edges of the hand to be accurately recorded (which can be difficult if the background does not contrast with the hand, or the hand self-occludes) otherwise the computerised model can lose much of its fidelity [10]. Another issue is the complexity of finding the 3D model that produces the measured silhouette (an example of inverse kinematics), which is a non-linear problem due to the trigonometric functions required [35]. If subsequent images in the video show only small differences in the hand's pose, these functions can be approximated linearly, but otherwise can require non-trivial amounts of computational time. The additional use of depth-sensing cameras [18] lifts these restrictions and improves the reliability of the metrics generated, although depth-sensing cameras are still non-standard and expensive compared to RGB cameras.

Alternatively, the appearance of the hand as the camera sees it can be used without inferring knowledge about the state of the hand. For example, Gupta et al. [38] use the EigenTracking algorithm [39] to generate an eigenspace representation of the hand which is invariant under affine transformation. This effectively condenses a large sequence of *a priori* hand images into a small set of images which represent hand-like qualities, and form the basis vectors of the eigenspace. The image to be tested is then decomposed into a linear sum of these basis vectors, resulting in the image's coordinates in the eigenspace which can then be used as a feature set describing the hand shape (regardless of how it is rotates or translates). However, the compactness of the eigenspace (which is required for an efficient description of the hand state) depends on the gesture vocabulary being small. [35].

Another appearance-based approach is to generate Haar-like features (as mentioned in section 2.2.1). Instead of detecting whether there is or is not a hand, the features would be used to classify the hand pose, thus requiring even more training than was required for hand detection.

2.2.3 Hand tracking

The hand-detection algorithms can be used in every frame to keep track of the hand, but doing so would ignore the coupling between image frames in a video. This would be good if the framerate is too low for the speed at which the hand is moving, but otherwise it ignores a wealth of data that could be used to track the hand. By using a separate hand tracking algorithm which uses this data, errors in either algorithm can be smoothed out.

A popular object-tracking technique is the CONDENSATION algorithm [40] which is capable of tracking curves against complex backgrounds, as in [38], and has been shown to be computationally efficient. Another technique is the Camshift algorithm [41] which has been shown to perform well with noise, distractors (e.g. other hands), and partial occlusion [42].

2.3 Classification

Classification compares the features extracted from a gesture against those from a vocabulary of gestures, selecting the most similar. In this proposal we are only concerned with classifying static gestures (i.e. classifying the hand pose in each frame separately).

The features on which to classify a hand might describe concrete data about the hand, such as digit joint angles or an image of the hand's silhouette, or they might describe more abstract data such as Haar-like features of the image. The best features to use will be invariant to translation, rotation, scaling, and changes in lighting (or at least robust to such changes), though a lack of invariance can be overcome by a more complex classifier and a more diverse training set [32].

The correct classifier to use, and its morphology, must be determined empirically by monitoring performance and computational efficiency. The AdaBoosted cascade of weak classifiers, as described in section 2.2.1, has shown to be a good classifier for high-dimensional features [30]. Another popular classifier [43, 44] is the support vector machine (SVM), which is a statistical approach to segregating the feature space. Others [45, 46] have used artificial neural networks, which comprise highly-connected processing units inspired by neurones that are trained iteratively through error-minimisation.

3 Machine Learning in Workflows

3.1 The Problem

As section 2 demonstrates, it would be useful to be able to decide on a strategy based on the supplied image. This could be accomplished by the algorithm developer understanding the strengths and flaws of each approach, then applying that general understanding to the problem of how to determine which approach best suits which kinds of image, and finally implementing conditional splits at the start of the workflow to appropriately direct any image to the most appropriate subworkflow. However, relying on the developer to form a concrete general understanding of the strengths and weaknesses of all the available approaches and all their configurations would be a poor choice; as would be relying on the developer to formalise the rules governing which workflow works best for any image.

Since all of the approaches so far discussed have involved a classification step based on supervised learning, we know that there is already a requirement to have labelled test images. As such, the aforementioned problem has an obvious solution in the form of supervised learning. Instead of formalising a set of rules governing which subworkflow an image should be processed along, we need only rely on the developer to select salient features of the image as inputs to a classifier. We could then automatically pass the entire training set through all subworkflows, then train the classifier at the point of divergence on which subworkflows successfully recognised each image. When an unseen image reaches the divergence point, its salient features would then be extracted and classified as being best suited to the subworkflow that worked well for similar test images.

This solution could be implemented in existing workflow systems, but would require a node for the classifier, a node for the conditional splitter (which acts upon the classifier's output), and edges from the final classifiers at the ends of each subworkflow to the classifier (for it to know which subworkflows successfully processed each training image). It would therefore require some time and concentration from the developer, and would add clutter to the work space. If the developer wanted to perform a similar split in one of the subworkflows, they would have to repeat this arrangement and add even more clutter.

Let us now suppose that the subworkflows all join together before going to the final classifier, which is likely since all of the subworkflows end with classifiers which might be the same type with the same settings. One might naively think that this reduces clutter and avoids repeating the latter sections of the subworkflows without expense; however, two problems emerge.

The first is how the classifier at the divergence point knows which subworkflow the image was processed by when it is returned from the final classifier. This could be circumvented by somehow adding a label to the image describing which subworkflow it went through, although this might become complicated by subworkflows containing more divergence points. In any case, it would require more nodes and edges, further cluttering the workspace.

The second problem is that the features the final classifier is trained with might be different for images coming from different subworkflows. Even if they are selected using the same algorithm (e.g. Haar-like features) the fact that the images were processed differently before the features were selected suggests that their features might no longer be compatible. This situation could confuse the classifier during training, leading to suboptimal gesture recognition.

The problem as described seems to be one of integration. Since the outcome of the workflow is always a classification (i.e. the recognised gesture) the edges coming back from the final classifier(s) offer the developer no insight. The classifier at the divergence point should have a way to determine which subworkflow the image went through.

To avoid these problems the developer might elect to not join the subworkflows, which might lead to repetition further along the workflow.

3.2 Proposed Solution

Our proposed solution to the first problem is to keep in the image's metadata an object representing its processing history, whereby two images processed the same way will have the same object. This way, a node like the classifier needs only to group images by this object, then create and train a classifier for each node independently.

To solve the second problem we propose giving the workflow two extra processing modes. The first is the processing of training data, whereby nodes process the data as normal, except for the conditional splits which instead process the data along all of its subworkflows. The second is the processing of a completed training batch, whereby all training data at the end of the workflow are marked as either successful or not, and then sent backwards through the workflow. We intend for each processed datum's metadata to contain a link to its unprocessed version, so most nodes will simply return these unprocessed versions (which are the previous node's processed data). It is at this point that classifier nodes are able to train their classifiers. When a divergence point passes backwards a completed training batch, it will know at that point which combinations of images and subworkflows lead to successful classification, so can train its classifier (or classifiers if there's another divergence point before it). Once trained, it will know which of the subworkflows it would have used to process each of the training data, and uses this to determine which of its input images went on to be successful before passing backward the completed training batch. A review of the literature suggests that this approach in a scientific workflow system is novel.

A final addition in our proposed solution is the ability to observe nodes and be notified when they process data. The purpose here is not to perform any step in the workflow, but for the observer node to be able to select features from a node earlier in the workflow, which it might eventually use in training a classifier (or if the processed data is not a training sample, it would use its trained classifier to classify it). Without being able to observe nodes, a classifier node would have to look back in its training data's historic values until it found a version of the data that it can select features from, but this is less obvious and appears to be more prone to error.

4 Design

4.1 Structure

Instead of the standard directed acyclic graph representation used by most graphical workflow systems, we elected to use a heterogeneous tree representation, whereby a `Workflow` is a linear sequence of `Element` objects (i.e. nodes) and a special type of element called a `WorkflowContainer` can contain multiple workflows. Using this structure means that every `Processor` (which workflows and elements are examples of) takes exactly one input and produces exactly one output, making the three processing modes described in section 3.2 simple to implement as recursive algorithms.

Each Datum is passed through the workflow inside a `Mediator` object, which contains a link to the previous mediator that was processed to give this one. It also contains a `History` object that is the same for all mediators that were only processed by the same processors in the same order.

Splitting and joining workflows is instead implemented as a single workflow container, of which there are three main types. First is the `TopWorkflowContainer` which is the top element in the system (i.e. the only processor without a parent).

Second is the `ChooserWorkflowContainer` which is an abstract class that functions like the example split and join nodes in 3.2. It can contain any number of workflows, and when asked to process data it must choose one of them to do it. When asked to process training data, it does so using all of its workflows (thus creating one output training data per workflow). When asked to process a `CompletedTrainingBatch`, it performs some processing on it to produce a collection showing which training samples succeeded in which workflows, which it then lets its concrete implementor handle (e.g. to train a classifier). At this point it is able to choose which workflow a sample should be processed by, and therefore whether or not each of the training samples would have gone on to be successfully classified (by following their chosen workflows).

Third is the the `SplitJoinWorkflowContainer` which is another abstract workflow container class that's designed for concurrently executing a number of workflows, after which the concrete implementor decides how to join the results into one output datum. When processing a training sample, it is given to all of its workflows to process as a training sample, meaning they might return multiple training samples. The workflow container takes the lists of training data returned from the workflows and performs a cartesian product across them, giving all possible combinations of training data from the workflows which the container subsequently joins and returns. When asked to process a completed training batch, it passes the batch to all of its workflows to process, then combines them and returns it.

All processors can optionally have a controller set, which is notified when the processor performs any processing function or is replaced (see section 4.3). I implemented these controllers as JavaFX control classes. With directed acyclic graphs, typical workflow systems rely on the user to move nodes around the workspace into a sensible arrangement and draw edges between them. In my heterogeneous tree structure however, the processors are deterministically laid out without the need for user interaction by using an `HBox` (horizontal box) for each workflow and a `VBox` (vertical box) for each workflow container.

4.2 Validation

Every processor has a `TypeData` object, which contains the input and output types that it requires and returns respectively. Workflows defer to their parent workflow containers for their type data, allowing the containers to intelligently adapt their type data. For example, the ChooseWorkflowContainer's input type is set to the output type of the previous element in its parent workflow, and its output type is set to the lowest common ancestor of the output types of the final elements in its workflows (LCA algorithm adapted from [47]).

It is then the responsibility of the enclosing workflow to ensure that its children elements have valid type data (i.e. in a workflow where element `A` comes before element `B`, `A`'s output must be castable to `B`'s input). If any of the elements in the workflow are invalid (either by mismatched types or the element described itself as invalid) the workflow can't process data. Visual cues show where the invalid processors are, and if it is due to mismatched types then those types are displayed next to their elements, to inform the user of the issue.

4.3 Mutability

When designing the system, we had a choice over the mutability of the workflow model. Mutable processors (whereby changes affect the processor directly) are the easiest to implement, but using them in a workflow system with multiple threads (i.e. the user interface thread and multiple workflow execution threads) would require a variety of locks, semaphores and synchronisation. In addition, using immutable objects makes implementing undo and redo functionality much easier.

As such we investigated the available methods of implementing immutability. The obvious way to do it in java would be to create a class with `private final` member variables which are set in the constructor. This approach works well for small classes, but otherwise results in a long, complicated constructor. It would be easy to subclass this immutable class and add more `private final` member variables to extend it, but again the constructor would have to be made larger and more complicated.

An alternative to forcing the user to use a large constructor is to write a `Object withX(X x)` method for each member variable `x` of type `X` which returns a clone of the object, but with variable `x` set in the method. Each `withX` method returns a call to the long and complicated constructor, passing it its own member variables except for `x` which is passed from the local field inside the wither method. The `Wither` class from Project Lombok[48] automatically generates these "wither" (i.e. `withX`, `withY`, etc...) methods, making it a strong candidate. However, extending the class to a subclass would break the wither methods as they point to the first class' constructor, thus calling a wither method on the derived class would return an object of the superclass. We intended for the processor classes to be easily extendable, so this approach was not applicable.

Another common approach is to wrap the immutable class (with the long constructor) inside a builder class. The builder has `setX(X x)` methods for all of the member variables in the immutable class, to which the member variables are passed when the builder is commanded to `build()`. Again, extending the builder or its immutable class is not simple.

Instead we chose an eventually-immutable approach, whereby the member variables are not `final` and the object is initially set as mutable, meaning `Object withX(X x)` simply sets the member variable `x` and returns `this`. The object can then be set as immutable, after which the wither methods return a mutable clone with the variable `x` set as desired. The mutable clone is created using `Object createMutableClone()` which simply passes itself to the copy constructor of its own class. Extending the class requires overriding the `Object createMutableClone()` method to instead use its copy constructor, and using covariant return types avoids the client code being forced to downcast the returned object. Below is an example eventually immutable class.

```
public class A extends EventuallyImmutable {
	@Getter private int a;

	public A() { a = 0; }		// !isMutable()
	public A(A oldA) { super(oldA); a = oldA.getA(); } // isMutable()

	@Override
	public A createMutableClone() { return new A(this); }

	public A withA(int a) {
		if (isMutable()) {
			this.a = a;
			return this;
		} else {
			return createMutableClone().withA(a);
		}
```

```
        }
}
```

Extending this class is then simple, as demonstrated below.

```
public class B extends A {
        @Getter private int b;

        public B() { b = 0; }         // !isMutable()
        public B(B oldB) { super(oldB); b = oldB.getB(); } // isMutable()

        @Override
        public B createMutableClone() { return new B(this); }

        public B withB(int b) {
                if (isMutable()) {
                        this.b = b;
                        return this;
                } else {
                        return createMutableClone.withB();
                }
        }

        @Override
        public B withA(int a) { return (B) super.withA(a); }
}
```

Client code using these classes would be self documenting, for example:

```
B b1 = new B();
B b2 = b1.withA(2);
B b3 = b1.withA(3).withB(5);
```

In our tree framework, the act of replacing any one processor implies that all processors in the framework need replacing (since links to the parent or children need to be updated, using appropriately named `withChildren(..)` and `withParent(..)` methods). I therefore elected to make the transition from mutable to immutable happen when replacing an older version (i.e. `obj2.replace(obj1)`), with four main stages. Firstly, eventually immutable subclasses can prepare themselves before replacing and becoming immutable by overriding the `replace(..)` command and inserting instructions before the call to `super.replace(..)`. This is where the old object's children and parents are prompted to replace themselves to point to the new object, and is done as a recursive algorithm defined in the `ChildManager` and `ParentManager` classes that causes every processor to be replaced.

The second stage is the replacement itself whereby the processors mark themselves as immutable, and inherits the old object's UUID (or universally-unique identifier) and controller. The UUID is shared between all versions of a processor, and by inheriting the old object's UUID the new object registers itself as the current version of the UUID with a central registry in the `ModelLoader` class. As an aside, undo and redo commands work by setting older or newer versions of the processors as the current version with the central registry.

The third stage happens when all of the other processors in the tree have been replaced, and instructs each of the new processors to run `afterReplacement()` which allows them to make any last-minute changes that couldn't be made before whilst the framework was being replaced. An example is the observer list maintained by each element, which is updated to reflect the newer versions of its observers after they have all registered with `ModelLoader`.

The final stage is updating the `TopController` (if it exists) that the model has been updated. In our implementation it then updates all of its children controllers with the new processor data.

4.4 Inversion of Control

Since all the Processors can be cloned, we took a prototype-based approach to inversion of control, whereby processors are registered with the `ModelLoader` class against a string identifier (found by calling `processor .getModelIdentifier()`, which by default returns `processor.getClass().getName()`). Whenever a processor is saved to xml (either as a file for saving or a string for putting on the dragboard) this model identifier is stored with it. When loading from xml, the model identifier is used to look up the prototype processor that was registered in `ModelLoader` which is subsequently cloned.

The controllers aren't explicitly registered in `ModelLoader`, but can instead by assigned as the controller to a prototype clone that is registered in `ModelLoader`. When the prototype processor is cloned, so too is its controller (if it has one). This way the model can always run without a controller, or with a different controller to that used when saving to xml.

Another example of inversion of control is in our JavaFX implementation of the controllers, in which we gave every element controller a `PreviewPane`, with which `DataViewer` objects are registered that are capable of displaying in realtime the element's output. When an element processes data, its `PreviewPane` looks in its class hierarchy for a superclass that a registered `DataViewer` can display.

4.5 XML

Given the tree structure inherent to our workflow system, serialising any part or all of it to XML is trivial. The processor to serialise creates an XML node and fills it with its member variables, and then if it has child processors it has each child produce an XML node to add to a `<Children>` tag. Those children might have children, in which case they will also populate a `<Children>` tag, thereby using the tree structure to allow for a simple recursive algorithm. Concrete processor implementations need only add their member variables to the XML node that their superclass returns, and the `XMLHelper` class provides useful methods to help with this.

4.6 Training

The `TopWorkflowContainer` initiates a training job by passing an empty mediator object to each of its workflows as training data. The first element must therefore be a data loader which loads the entire training set, and the training data processes through the workflow as per the description in section 3.2. When the training data reaches the end of the top workflow, it is placed in a `CompletedTrainingBatch` object that contains a set of all the training data, and a set of all the successful training data. At this stage all of the training data is marked as successful, so classifiers can chain together to mark data as unsuccessful. The completed training batch is then passed backwards through the workflow in order to train its elements, as per section 3.2.

Our implementation contained three trainable elements. The first is the `NNClassifier` class, which implements a neural network from the OpenCV framework[49] for each unique `History` object in the training set. The second is a `Optimiser` class, which is a `ChooserWorkflowContainer` that statically chooses the workflow that worked best for training data with the same `History` object. The third is the `KeyboardActor` class that enumerates all of the tags of the training set images so a keyboard actions can be assigned to each tag, which are executed when an input image is classified.

Both the `Optimiser` and `NNClassifier` elements discriminate data samples based on their `History` objects, and to show this graphically we placed a button on each element for each history object it trained for. The button label shows the success rate of training samples with that history object, and clicking the button selects all processors in the object's history. This demonstrates to the end-user which paths through the framework work well and for which data.

4.7 Extensibility

We have designed the framework to be easily extended. To create a new element, a developer can extend `AbstractElement`, `SplitJoinWorkflowContainer`, or `ChooserWorkflowContainer`, and register the model with `ModelLoader`. Their controllers can be created by extending `ElementController` or `WorkflowContainerControllerImpl`. Any new data types that should be viewable should create a data viewer that extends `DataViewer` and is registered with `PreviewPane`.

4.8 Hand Gesture Implementation

For the purposes of demonstrating the effectiveness of the workflow paradigm with respect to hand gesture recognition, we chose an exemplar strategy where each step produces a visible and understandable change to the image. To this end, we chose not to use Haar-like features or object tracking, though these could easily be implemented in the workflow. Given the time restraints of this project, we elected to construct a strategy that works well in ideal circumstances (i.e. the camera frame contains only a hand and a simple background) without looking at more complicated or realistic situations so we could at least demonstrate the workflow's potential, if not create a recognition system ready for deployment to end-users.

The first step in our approach was to convert the image to HSV space and filter out the background. The controller for this element allowed for the upper and lower HSV boundaries to be set by dragging sliders whilst watching the input and output image of the filter, making the process much easier.

The next step was to find the outer contours from the image and select the longest, which represents the outer contour of the hand. This contour was then simplified to a polygon using the Douglas-Peucker algorithm [50].

The contour was then concurrently analysed to find both the palm's centre using image moments [51], and the fingertip positions by finding a convex hull [52] around the hand and selecting for the five largest convexity defects. When joined, these data resulted in the vectors between the centre of the palm and the fingertips, whose lengths were used as features for the neural network classifier to train on using the leave-one-out cross-validation method.

Finally the output classification went to the keyboard actor, which simulates the configured key being pressed.

5 Results

In repeat experiments, the neural network's success rate was in the range of 96% to 100%.

To demonstrate the importance of the contour simplification step an `Optimiser` was used, whereby one of its workflows contained a `SimplifyContour` element with an accuracy parameter of 20.0, the next contained another `SimplifyContour` element with an accuracy parameter of 4.0, the next had another `SimplifyContour` with an accuracy parameter of 0.2, and the last had no elements in it (i.e. the input contour passed through it unchanged). The workflows with the `SimplifyContour` elements with accuracy parameters of 20.0 and 4.0 scored 100%, whereas the workflow with the `SimplifyContour` element with an accuracy parameter of 0.2 and the empty workflow scored between 88% to 96%.

6 Analysis and Critical evaluation

The high rate of success is unsurprising, since we specifically chose hand gestures where the fingertips and spaces between the fingers were never occluded, and took the photos in front of a background that could be easily filtered out in the HSV colour space. Using more complex images would have resulted in features uncorrelated with the hand's pose, so would not have been useful in demonstrating the machine learning integrated into the

The ability to see the effects of changes to the workflow in realtime was invaluable in the development of the algorithm, and using the `Optimiser` to compare different strategies side-by-side was very useful.

It is unfortunate that there was insufficient time to implement more complex hand gesture recognition strategies. With more strategies, we could have demonstrated the effectiveness of selecting a strategy based on which proved most effective for test samples with similar features. With a more resilient algorithm, it would have been worthwhile to create a user-friendly user interface like BTT [16] to allow for custom gestures to be linked to custom actions without needing to see the underlying workflow elements.

References

[1] Ewa Deelman and Ann Chervenak. *Data Intensive Distributed Computing.* IGI Global, January 2012.

[2] Fernando Chirigati, Vítor Silva, Eduardo Ogasawara, Daniel de Oliveira, Jonas Dias, Fábio Porto, Patrick Valduriez, and Marta Mattoso. Evaluating parameter sweep workflows in high performance computing. *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies - SWEET '12*, pages 1–10, 2012.

[3] Ilkay Altintas, Chad Berkley, and Efrat Jaeger. Kepler: an extensible system for design and execution of scientific workflows. *Scientific and ...*, 2004.

[4] M Záková and V Podpecan. Advancing data mining workflow construction: A framework and cases using the orange toolkit. *... Generation Data Mining ...*, 2009.

[5] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics (Oxford, England)*, 20(17):3045–54, November 2004.

[6] BA Myers. A Brief History of Human-Computer Interaction Technology. *interactions*, pages 44–54, 1998.

[7] Ying Wu and TS Huang. Vision-based gesture recognition: A review. *Urbana*, 51:103–115, 1999.

[8] Ying Wu and TS Huang. For Vision-Based Human Computer Interaction. *IEEE Signal Processing Magazine*, (May):51–60, 2001.

[9] Fakhreddine Karray, Milad Alemzadeh, JA Saleh, and MN Arab. Human-computer interaction: Overview on state of the art. *International Journal On Smart Sensing and Intelligent Systems*, 1(1):137–159, 2008.

[10] GRS Murthy and RS Jadon. A review of vision based hand gestures recognition. *International Journal of Information Technology and Knowledge Management*, 2(2):405–410, 2009.

[11] Ping Zhang and Dennis F Galletta. *Human Computer Interaction And Management Information Systems: Foundations.* ME Sharpe Incorporated, 2006.

[12] R Sharma and TS Huang. Speech/Gesture Interface to a Visual Computing Environment for Molecular Biologists. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference*, pages 964–968. 1996.

[13] G H Ballantyne. Robotic surgery, telerobotic surgery, telepresence, and telementoring. Review of early clinical results. *Surgical endoscopy*, 16(10):1389–402, October 2002.

[14] C Grätzel, T Fong, S Grange, and C Baur. A non-contact mouse for surgeon-computer interaction. *Technology and health care : official journal of the European Society for Engineering and Medicine*, 12(3):245–57, January 2004.

[15] Zahoor Zafrulla, Helene Brashear, Pei Yin, Peter Presti, Thad Starner, and Harley Hamilton. American Sign Language Phrase Verification in an Educational Game for Deaf Children. *2010 20th International Conference on Pattern Recognition*, pages 3846–3849, August 2010.

[16] Andreas Hegenberg. BetterTouchTool <www.boastr.de>.

[17] Thomas G Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. A hand gesture interface device. *SIGCHI Bull.*, 17(SI):189–192, May 1986.

[18] Michael Van Den Bergh and L Van Gool. Combining RGB and ToF cameras for real-time 3D hand gesture interaction. *Proceedings of the 2011 IEEE Workshop on Applications of Computer Vision*, pages 66–72, 2011.

[19] K. Fujimura. Hand gesture recognition using depth data. *Sixth IEEE International Conference on Automatic Face and Gesture Recognition, 2004. Proceedings.*, pages 529–534, 2004.

[20] Y. Sato, Y. Kobayashi, and H. Koike. Fast tracking of hands and fingertips in infrared images for augmented desk interface. *Proceedings Fourth IEEE International Conference on Automatic Face and Gesture Recognition (Cat. No. PR00580)*, pages 462–467, 2000.

[21] Georg Hackenberg, R McCall, and W Broll. Lightweight palm and finger tracking for real-time 3D gesture control. In *Virtual Reality Conference (VR), 2011 IEEE*, number March 2010, pages 19–26. 2011.

[22] MJ Jones and JM Rehg. Statistical color models with application to skin detection. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference*, pages 274–280. 1999.

[23] Michael Donoser and Horst Bischof. Real time appearance based hand tracking. *2008 19th International Conference on Pattern Recognition*, pages 1–4, December 2008.

[24] J Fritsch and S Lang. Improving adaptive skin color segmentation by incorporating results from face detection. *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop*, (September), 2002.

[25] Ying Wu, Qiong Liu, and TS Huang. An Adaptive Self-Organizing Color Segmentation Algorithm with Application to Robust Real-time Human Hand Localization. In *Proc. of Asian Conference on Computer Vision*, pages 1106—-1111. 2000.

[26] K. Mikolajczyk, T. Tuytelaars, C. Schmid, a. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool. A Comparison of Affine Region Detectors. *International Journal of Computer Vision*, 65(1-2):43–72, October 2005.

[27] Taehee Lee and Tobias Hollerer. Handy AR: Markerless Inspection of Augmented Reality Objects Using Fingertip Tracking. *2007 11th IEEE International Symposium on Wearable Computers*, pages 1–8, October 2007.

[28] Matilde Gonzalez, Christophe Collet, and R Dubot. Head tracking and hand segmentation during hand over face occlusion in sign language. *Trends and Topics in Computer Vision*, pages 234–243, 2012.

[29] Paul Smith, Niels da Vitoria Lobo, and Mubarak Shah. Resolving hand over face occlusion. *Image and Vision Computing*, 25(9):1432–1448, September 2007.

[30] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, 1:I–511–I–518, 2001.

[31] Qing Chen, Nicolas D Georganas, and Emil M Petriu. Real-time vision-based hand gesture recognition using haar-like features. In *Instrumentation and Measurement Technology Conference Proceedings*, pages 1–6. 2007.

[32] ALC Barczak and Farhad Dadgostar. Real-time hand tracking using a set of cooperative classifiers based on Haar-like features. *Res. Lett. Inf. Math. Sci*, 7:29–42, 2005.

[33] R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. *Proceedings. International Conference on Image Processing*, 1:I–900–I–903, 2002.

[34] Y Freund and RE Schapire. A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting. *Journal of computer and system sciences*, 1997.

[35] Pragati Garg, Naveen Aggarwal, and Sanjeev Sofat. Vision Based Hand Gesture Recognition. *World Academy of Science, Engineering and Technology*, 49(1):972–977, 2009.

[36] Franklin C. Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, July 1984.

[37] B Stenger. Model-based 3D tracking of an articulated hand. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conferenc*, pages II–310. 2001.

[38] N Gupta, P Mittal, and SD Roy. Developing a gesture-based interface. *Journal of the Institution of Electronics and Telecommunication Engineers*, 48(3):237–244, 2002.

[39] MJ Black and AD Jepson. Eigentracking: Robust matching and tracking of articulated objects using a view-based representation. *International Journal of Computer Vision*, 1998.

[40] Michael Isard and Andrew Blake. CONDENSATION - Conditional Density Propagation for Visual Tracking. *International journal of computer vision*, 29(1):5–28, 1998.

[41] GR Bradski. Computer vision face tracking for use in a perceptual user interface. *Intel Technology Journal*, 1998.

[42] S M Nadgeri, S D Sawarkar, and a D Gawande. Hand Gesture Recognition Using CAMSHIFT Algorithm. *2010 3rd International Conference on Emerging Trends in Engineering and Technology*, pages 37–41, November 2010.

[43] Yuelong Chuang, Ling Chen, Gangqiang Zhao, and Gencai Chen. Hand posture recognition and tracking based on Bag-of-Words for human robot interaction. *2011 IEEE International Conference on Robotics and Automation*, pages 538–543, May 2011.

[44] Yu Ren and Fengming Zhang. Hand Gesture Recognition Based on MEB-SVM. *2009 International Conference on Embedded Software and Systems*, pages 344–349, 2009.

[45] E. Stergiopoulou and N. Papamarkos. Hand gesture recognition using a neural network shape fitting technique. *Engineering Applications of Artificial Intelligence*, 22(8):1141–1158, December 2009.

[46] D. K. Ghosh and S. Ari. A static hand gesture recognition algorithm using k-mean based radial basis function neural network. *2011 8th International Conference on Information, Communications & Signal Processing*, (i):1–5, December 2011.

[47] ? Adam. http://stackoverflow.com/questions/9797212/finding-the-nearest-common-superclass-or-superinterface-of-a-collection-of-cla.

[48] http://projectlombok.org/api/lombok/experimental/Wither.html.

[49] G Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[50] David H Douglas and Thomas K Peucker. Algorithms For The Reduction Of The Number Of Points Required To Represent A Digitized Line Or Its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, October 1973.

[51] MK Hu. Visual Pattern Recognition by Moment Invariants. *Information Theory, IRE Transactions on*, pages 66–70, 1962.

[52] Jack Sklansky. Finding the convex hull of a simple polygon. *Pattern Recogn. Lett.*, 1(2):79–83, December 1982.

[53] Samuel Audet. JavaCV: https://code.google.com/p/javacv/.

A Open Source Code Used

- OpenCV[49]
- JavaCV[53]
- Hand pose detection code in JavaCV: http://www.javacodegeeks.com/2012/12/hand-and-finger-detection-using-javacv.html
- Neural network classifier in JavaCV: http://projectagv.blogspot.co.uk/2008/12/sample-code-for-cvannmlp.html
- Lowest-Common-Ancestor algorithm: http://stackoverflow.com/questions/9797212/finding-the-nearest-common-superclass-or-superinterface-of-a-collection-of-cla
- How to nicely format XML documents: http://stackoverflow.com/questions/5142632/java-dom-xml-file-create-have-no-tabs-or-whitespaces-in-output-file

B Instructions for using the software

To run, you'll need JavaFX in your java path. Unfortunately this isn't done by default (but will be in Java 8). To fix this, run:

```
mvn com.zenjava:javafx-maven-plugin:1.2:fix-classpath
```

with privileges sufficient for creating files in the Java distribution (normally administrative).

You'll need `jdk1.7.0_17` (there's a bug on earlier versions of Java 7 which will cause the application to crash) and maven.

You'll need to install the OpenCV libraries and make them available to Java. To do this, see:

http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html#windows-install-prebuild

http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html

http://opencv.willowgarage.com/wiki/Mac_OS_X_OpenCV_Port

Once done, double click on "Gesture Framework 1.0.jar" to run. Alternatively, go to directory with pom.xml and run "mvn jfx:run".

If you get errors about unsatisfied links, or only have 5 elements in the toolbox (on the right), then OpenCV hasn't been installed on the java path properly. I've included videos of me using the system.

C Source Code

The entire project contains approximately 8500 lines of code, so printing it all would consume approximately 140 A4 pages. Instead, below are the interfaces. The full source code, and a print out in a pdf, are available on the attached CD and submitted by moodle.

```
package io.github.samwright.framework.controller;

import io.github.samwright.framework.model.Element;
import io.github.samwright.framework.model.Processor;
import io.github.samwright.framework.model.helper.Mediator;

import java.util.List;
```

```
/**
 * A controller assigned to manage a {@link Processor} object.
 * <p/>
 * When the assigned {@code Processor} is replaced with a newer version, processes data,
 * or is trained, this is notified.
 */
public interface ModelController {
    /**
     * Propose a new model for this to manage. It might be a new version of the
     * previously-managed {@link Processor}, or it might be a completely new one.
     * <p/>
     * When a model is proposed, it is not immediately accessible by {@code getModel()},
         nor is
     * {@code handleUpdatedModel()} immediately called. The {@link TopController} manages
         when
     * to call {@code handleUpdatedModel()}, and does so recursively from it down to the
         lowest
     * {@link Element}.
     *
     * @param proposedModel the {@code Processor} to propose.
     */
    void proposeModel(Processor proposedModel);

    /**
     * Creates a deep clone of this (but with no model).
     *
     * @return a complete clone of this (but with no model).
     */
    ModelController createClone();

    /**
     * Called when a new model has been proposed and the {@link TopController} has
         determined
     * that the time is right to handle it.
     */
    void handleUpdatedModel();

    /**
     * Gets the currently-managed model.
     * <p/>
     * If a new model is proposed, this method will only return it once
     * {@code handleUpdatedModel()} has been called. Until then it will return the latest
         model
     * to be handled by {@code handleUpdatedModel()}.
     *
     * @return the currently-managed model.
     */
    Processor getModel();

    /**
     * Called when the managed {@link Processor} successfully processes input data.
     * <p/>
     * NB. This will be called from the {@code Processor} object's processing thread,
     * which will wait until this method returns, and won't necessarily be the same thread
          that
     * this object was created with.
     *
     * @param processedData the data the managed {@code Processor} successfully processed.
     */
    void handleProcessedData(Mediator processedData);

    /**
     * Called when the managed {@link Processor} successfully processes a single training
         datum.
     * <p/>
     * NB. This will be called from the {@code Processor} object's processing thread,
     * which will wait until this method returns, and won't necessarily be the same thread
          that
     * this object was created with.
     *
     * @param processedTrainingData the data the managed {@code Processor} successfully
         processed
     * from a single input training datum.
     */
    void handleProcessedTrainingData(List<Mediator> processedTrainingData);

    /**
     * Called when the managed {@link Processor} has been successfully trained.
     * <p/>
     * NB. This will be called from the {@code Processor} object's processing thread,
     * which will wait until this method returns, and won't necessarily be the same thread
          that
     * this object was created with.
     */
    void handleTrained();

}
```

```
package io.github.samwright.framework.controller;

import io.github.samwright.framework.model.mock.TopProcessor;

/**
 * The top controller, which is the only type of controller that a {@link TopProcessor}
     can have.
 */
public interface TopController extends ModelController {

    /**
     * If the controlled {@link TopProcessor} catches an exception,
     * it will pass it here to be properly handled (eg. by showing the exception in a
         modal popup
     * window).
     *
     * @param e the exception thrown whilst processing.
     */
    void handleException(Exception e);
}
```

```
package io.github.samwright.framework.model.common;

/**
 * A child of the parent type {@code P}.
 */
public interface ChildOf<P extends ParentOf<? extends ChildOf<P>>> {
    /**
     * Gets the parent that this child sits within.
     * @return the parent that this child sits within.
     */
    P getParent();

    /**
     * Return a version of {@code this} with the given parent.
     * <p/>
     * If this object can be mutated it will be, with {@code this} being returned.
     * Otherwise a clone of {@code this} will be created with the given parent.
     *
     * @param newParent the parent to be given to the returned object.
     * @return a version of {@code this} with the given parent.
     */
    ChildOf<P> withParent(P newParent);
}
```

```
package io.github.samwright.framework.model.common;

import io.github.samwright.framework.controller.ModelController;

/**
 * A class whose instantiations can each have a controller.
 */
public interface Controllable {

    /**
     * Set the controller for this object, which is notified of replacements to this
         object.
     * <p/>
     * If future versions of this object already exist, they will all now use {@code
     * modelController} as their controller, and the controller will use this as its
         current model.
     *
     * @param modelController the controller for this object.
     */
    void setController(ModelController modelController);

    /**
     * Gets the controller for this object.
     *
     * @return the controller for this object.
     */
    ModelController getController();
}
```

```
package io.github.samwright.framework.model.common;

/**
 * A class whose instantiations are deletable.
 */
public interface Deletable {

    /**
     * Delete this.
     */
    void delete();
}
```

```java
package io.github.samwright.framework.model.common;

import io.github.samwright.framework.model.Element;
import io.github.samwright.framework.model.Processor;
import io.github.samwright.framework.model.helper.Mediator;

import java.util.List;

/**
 * An object the observes the processing of data in a {@link Element}.
 */
public interface ElementObserver {

    /**
     * Notify this {@code ElementObserver} that a {@link Processor} it observes is about
         to output
     * a {@link Mediator} object.
     *
     * @param processedData the {@link Mediator} the observed {@link Processor} is about
         to output.
     */
    void handleProcessedData(Mediator processedData);

    /**
     * Notify this {@code ElementObserver} that a {@link Processor} it observes is about
         to output
     * a list of {@link Mediator} objects it created from a single input training datum.
     *
     * @param processedTrainingData the list of {@link Mediator} objects the observed
     * {@code Processor} created from a single input training datum.
     */
    void handleProcessedTrainingData(List<Mediator> processedTrainingData);

}
```

```java
package io.github.samwright.framework.model.common;

/**
 * An object which eventually becomes immutable.
 * <p/>
 * It might be immutable from creation, or it might start as mutable and become immutable.
      Once
 * immutable, it can not become mutable.
 * <p/>
 * The convention to use is {@code X getX()} for accessors and
 * {@code EventuallyImmutable withX(newX)} for mutators.
 * <p/>
 * Whilst {@code this.isMutable()}, mutators mutate the internal data and return the same
     object.
 * Once immutable, mutators create a mutable clone and mutations are applied to it, i.e.
 * {@code createMutableClone().withX(newX)}.
 * <p/>
 * Mutations can therefore be strung together, e.g. {@code obj.withX(newX).withY(newY).
     withZ(newZ)}
 * will return {@code obj} with the mutations if {@code obj.isMutable()} or
 * {@code obj.createMutableClone()} with the mutations otherwise.
 *
 * @see io.github.samwright.framework.model.helper.MutabilityHelper MutabilityHelper
 */
public interface EventuallyImmutable {
    /**
     * Create a mutable clone of this object. It will remain mutable until passed as a
     * {@code replacement} to {@code replaceWith(replacement)}, at which point
     * {@code fixAsVersion(..)} will be called to effect the change to an immutable object
         .
     *
     * @return a mutable clone.
     * @throws RuntimeException if this object is still mutable.
     */
    EventuallyImmutable createMutableClone();

    /**
     * Returns true iff this object is mutable, meaning it was created as mutable and has
         not
     * been finalised.
     *
     * @return true iff this object is mutable.
     */
    boolean isMutable();
}
```

```
package io.github.samwright.framework.model.common;

import java.util.List;

/**
 * A Parent of the child type {@code C}.
 */
public interface ParentOf<C extends ChildOf<? extends ParentOf<C>> & Validatable> {

    /**
     * Gets the list of children of this parent.
     *
     * @return the list of children of this parent.
     */
    List<C> getChildren();

    /**
     * Return a version of {@code this} with the given children.
     * <p/>
     * If this object can be mutated it will be, with {@code this} being returned.
     * Otherwise a clone of {@code this} will be created with the given children.
     *
     * @param newChildren the children to be given to the returned object.
     * @return a version of {@code this} with the given children.
     */
    ParentOf<C> withChildren(List<C> newChildren);

    /**
     * Returns true iff the child elements are all valid.
     *
     * @return true iff the child elements are all valid.
     */
    boolean areChildrenValid();
}
```

```
package io.github.samwright.framework.model.common;

/**
 * An object that can be replaced by another (or deleted), whilst keeping track of earlier
      and
 * later versions.
 */
public interface Replaceable {
    /**
     * Propose a replacement for this object. How this is handled is determined by the
         concrete
     * class implementing this.
     * <p/>
     * This must have no next version, and the replacement must have no previous version -
     * otherwise a RuntimeException is thrown.
     * <p/>
     * As an example, a sequence of versions between {@code Replaceable}
     * objects {@code start} and {@code end} can be contracted using
     *
     * <pre>{@code
     * start.discardNext(); // start.getNext() == null
     * end.discardPrevious(); // end.getPrevious() == null
     * start.replaceWith(end);
     * }</pre>
     *
     * @param replacement the replacement to propose.
     * @throws RuntimeException if this object is still mutable,
     * or has already been replaced or deleted.
     */
    void replaceWith(Replaceable replacement);

    /**
     * Propose that this object replace the supplied object. If the given object is null,
     * this becomes the start of a new line of objects.
     *
     * @param toReplace the object to replace.
     */
    void replace(Replaceable toReplace);

    /**
     * This is called after all relevant objects have been fixed, and presents the
         implementing
     * class with the opportunity to perform last-minute changes to itself now that all
         related
     * objects have also been fixed.
     */
    void afterReplacement();

    /**
     * Returns true iff this object is currently replacing another.
     *
     * @return true iff this object is currently replacing another.
     */
    boolean isReplacing();
}
```

```java
package io.github.samwright.framework.model.common;

/**
 * Implementors can be valid or invalid.
 */
public interface Validatable {
    /**
     * Returns true iff this object, in itself, is valid.
     *
     * @return true iff this is valid.
     */
    boolean isValid();
}
```

```
package io.github.samwright.framework.model.common;

import java.util.UUID;

/**
 * Each {@link EventuallyImmutable} object has one {@code VersionInfo} object which
     describes
 * tracks its earlier and later versions.
 */
public interface Versioned {

    /**
     * Gets the next version of the {@link EventuallyImmutable} object. If this describes
         the
     * latest version, the method returns null.
     *
     * @return the next version of the {@link EventuallyImmutable} object.
     */
    Versioned getNext();

    /**
     * Gets the previous version of the {@link EventuallyImmutable} object. If this
         describes the
     * earliest (ie. first) version, the method returns null.
     *
     * @return the previous version of the {@link EventuallyImmutable} object.
     */
    Versioned getPrevious();

    /**
     * Discards any replacement to this object and ensures this object is not deleted,
         therefore
     * allowing for it to be replaced by another {@code Replaceable} object. In doing so,
         it makes
     * the next version discard its previous version.
     */
    void discardNext();

    /**
     * Discard older versions of this, so they may be freed by the garbage collector. In
         doing so
     * it makes the previous version discard its next version.
     */
    void discardPrevious();

    /**
     * Sets the universally unique identifier for this object (and all future versions of
         this).
     * <p/>
     * No checks for uniqueness are performed.
     * <p/>
     *
     * @param uuid the universally unique identifier to set for this object.
     */
    void setUUID(UUID uuid);

    /**
     * Gets the universally unique identifier for this object.
     */
    UUID getUUID();

    /**
     * Sets this object as the current version (along with objects associated with it),
         informing
     * all relevant controllers and registers to the change.
     */
    void setAsCurrentVersion();

    /**
     * Returns the current version of this object.
     *
     * @return the current version of this object.
     */
    Versioned getCurrentVersion();
}
```

```
package io.github.samwright.framework.model.common;

import io.github.samwright.framework.model.Processor;
import io.github.samwright.framework.model.helper.ModelLoader;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

import java.util.Map;
import java.util.UUID;

/**
 * Implementers can be serialised to an {@link Element}, or deserialised from one.
 */
public interface XMLSerialisable {

    /**
     * Serialises this object into an XML node for the given document.
     * <p/>
     * It is for the calling object to decide where in the document to append the returned
         node.
     *
     * @return this object as an XML node.
     */
    Element getXMLForDocument(Document doc);

    /**
     * Creates a mutable clone of this object, with the data defined in the supplied XML
     * {@link Node} where UUIDs in the node are first looked-up in the supplied dictionary
         . If
     * there is no match, the UUID is then looked-up in the {@link ModelLoader} class.
     * <p/>
     * When an object defined in the supplied {@code Node} is instantiated,
     * the new object will be given a new UUID, and an entry will be made in the supplied
     * dictionary to link from the old UUID and this new object. Subsequent references to
        the old
     * UUIDs are then dereferenced using the dictionary. UUIDs of external objects (which
     * are available to both the original and the new objects) are instead dereferenced in
         the
     * {@code ModelLoader.getProcessor(uuid)} method. Other references are assumed to be
     * transient (eg. relating to controllers and will be recreated when needed) and thus
        ignored.
     * <p/>
     * It is therefore imperative that the new objects are created and the dictionary
        fully
     * populated before any dereferencing of UUIDs is attempted.
     *
     * @param node the XML node containing the data to populate the new object with.
     * @param dictionary the dictionary to use for translating old UUIDs to new UUIDs. In
        most
     * cases, this should be an empty dictionary (which the function then
     * populates).
     */
    XMLSerialisable withXML(Element node, Map<UUID, Processor> dictionary);
    /**
     * Gets the model identifier for this object, which is saved when serialising to XML
         and is
     * registered with the {@link ModelLoader}.
     *
     * @return the model identifier.
     */
    String getModelIdentifier();

    /**
     * Gets the XML tag, which is used as the name of the node when serialising to XML.
     *
     * @return the XML tag.
     */
    String getXMLTag();
}
```

```
package io.github.samwright.framework.model.datatypes;

import java.util.List;

/**
 * Features set
 */
public interface Features {
    /**
     * Gets the enclosed features.
     *
     * @return the enclosed features.
     */
    List<Double> getFeatures();

    /**
     * The tag from the training data, or null if not training data.
     * @return tag from the training data, or null if not training data.
     */
    String getTag();
}
```

```java
package io.github.samwright.framework.model.datatypes;

import io.github.samwright.framework.model.mock.TopProcessor;

/**
 * The input type required by the {@link TopProcessor} object's workflows.
 */
public interface StartType {
}
```

```java
package io.github.samwright.framework.model;

import io.github.samwright.framework.model.common.ChildOf;
import io.github.samwright.framework.model.common.ElementObserver;
import io.github.samwright.framework.model.helper.TypeData;

import java.util.Set;

/**
 * An {@code Element} is a {@link Processor} which sits inside a {@link Workflow}.
 *
 * It can be observed by {@link ElementObserver} objects, which are registered with the
 * {@code withObservers(newObservers)} method.
 */
public interface Element extends Processor, ChildOf<Workflow> {

    /**
     * Gets the {@link ElementObserver} objects that are observing this. When this is
     * replaced, the observers are updated to their latest versions.
     *
     * @return the {@link ElementObserver} objects that are observing this.
     */
    Set<ElementObserver> getObservers();

    /**
     * Returns this (or if this is not mutable, a clone) with the given set of observers.
     *
     * @param newObservers the observers who's latest versions will be in the returned
     * {@code Element}.
     * @return an {@code Element} with the latest versions of the given {@code
        newObservers}.
     */
    Element withObservers(Set<ElementObserver> newObservers);

    /**
     * Return a clone of this with the given {@link TypeData}. It will only work if this
        is
     * immutable (since the parametric types must be set at instantiation).
     *
     * @param newTypeData the type data to put in the returned clone.
     * @return a clone of this with the given {@code TypeData}.
     * @throws RuntimeException if this is still mutable.
     */
    Element withTypeData(TypeData newTypeData);

    @Override
    Element withParent(Workflow newParent);

    @Override
    Element createMutableClone();

    @Override
    Element getCurrentVersion();
}
```

```java
package io.github.samwright.framework.model;

import io.github.samwright.framework.model.common.*;
import io.github.samwright.framework.model.helper.CompletedTrainingBatch;
import io.github.samwright.framework.model.helper.History;
import io.github.samwright.framework.model.helper.Mediator;
import io.github.samwright.framework.model.helper.TypeData;
import io.github.samwright.framework.model.mock.TopProcessor;
import org.w3c.dom.Element;

import java.util.List;
import java.util.Map;
import java.util.UUID;

/**
 * The top-level interface for all parts of the workflow framework.
 * <p/>
 * A {@code Processor} can process input data (inside a {@link Mediator}) and return
    output data
 * inside another {@code Mediator}.
 * <p/>
 * Before being asked to process a {@code Mediator}, it will first process a training
    batch. A
 * typical sequence that the {@code Processor} methods are called in is:
 * <p/>
 * - {@code notify(List<Mediator<O>>)} : from observed Processors as they process the
 * training batch. Features of the supplied data can be gathered here. NB. this is only
 * called if the {@code Processor} is a {@link ElementObserver}.
 * <p/>
 * - {@code processTrainingBatch(..)} : when it's this object's turn to process the
 * training batch.
 * <p/>
 * - {@code processCompletedTrainingBatch(..)} : when the completed training batch
 * is travelling backward over the workflow that created it. This is where the
 * {@code Processor} may learn from the training batch (possibly correlated against
 * previously-gathered data features).
 * <p/>
 * This sequence allows allows for the {@code Processor} to prepare itself,
 * for example it might choose to process the data in a certain way that worked well
    during
 * training.
 */
public interface Processor
       extends EventuallyImmutable, XMLSerialisable, Versioned,
              Replaceable, Controllable, Deletable, Validatable {


    /**
     * Process the input {@link Mediator} object and return the result in an output {@code
     * Mediator} object.
     *
     * @param input input data in a {@code Mediator} object.
     * @return result of processing the input data, inside a {@code Mediator<O>} object.
     * @throws ClassCastException if input cannot be cast to {@code Mediator<I>}.
 */
Mediator process(Mediator input);

/**
 * Given a {@link Mediator} object containing training data, produce all output {@code
 * Mediator} objects that this {@link Processor} could possibly produce. The input
    data will
 * be for
 * training, and if this {@code Processor} expects to be notified of prior {@code
    Processor}
 * objects' completions, it will be notified by the {@code notify(List<Mediator<?>>)}
    method,
 * which contains all training data.
 * <p/>
 * For elemental {@code Processors}, this means applying {@code process(input)} on
    each
 * input, and for {@code Processors} comprising multiple {@link Workflow} objects this
     means
 * processing each input using all appropriate {@code Workflows}.
 *
 * @param input set of {@code Mediator} objects containing input data.
 * @return all output {@code Mediators} that this could possibly produce,
 * indexed by each input mediator that created them.
 */
List<Mediator> processTrainingData(Mediator input);

/**
 * Gets the input/output {@link TypeData} required by this {@link Processor}.
 *
 * @return the input/output types required by this {@code Processor}.
 */
TypeData getTypeData();

/**
 * This method is called after a training batch has been processed to completion, and
    returns
 * a rolled-back version of the supplied {@link CompletedTrainingBatch}.
 * <p/>
 * The supplied {@code completedTrainingBatch} object contains the {@code Mediator}
    objects
 * returned from the last call to {@code processTrainingBatch(..)}, and which of those
     went on
 * to be successful.
 * <p/>
 * "Rolling-back" means reverting these output {@code Mediator<O>} objects to input
 * {@code Mediator<I>} objects, and correctly marking which of those went on to be
    successful.
 * <p/>
 * {@code Processor} objects containing multiple ways of processing the same input
    will have
 * created multiple outputs per input, in which case this method must choose which
    output
 * would have been created had the input been given to {@code process(input)}. Only if
     this
 * output was successful will its corresponding input mediator will be marked as
     successful.
 * <p/>
 * If this {@code Processor} needs training or optimising, this is the method to do it
     .
 * <p/>
 * If this {@code Processor} intends to train a classifier to be used in {Code process
     (input)}
 * to select the best strategy for each individual input data, it is advisable to
     create a
 * classifier for each input {@code Mediator} object's {@link History} object. This
     means
 * that data that is created differently is classified differently.
 * <p/>
 * One approach would be to use a {@code Map<History, Classifier>} object that is
     queried in
 * the {@code process(input)} method to get the appropriate
 * {@code Classifier} for the given
 * {@code input.getHistory()} object.
 * <p/>
 * It is left to the concrete class to decide whether to do this or not (eg. the
     training set
 * per classifier would be larger by not doing this).
 *
 * @param completedTrainingBatch the completed training batch, containing the {@link
     Mediator}
 * objects returned from the last call to
 * {@code processTrainingBatch(..)}, and which of those went on
 * to be successful.
 * @return the rolled-back {@code CompletedTrainingBatch}.
 */
CompletedTrainingBatch processCompletedTrainingBatch(
        CompletedTrainingBatch completedTrainingBatch);

/**
 * Gets the ultimate ancestor of this {@code Processor} if it is in a complete model (
     ie. the
 * ultimate ancestor is a {@link TopProcessor}. Otherwise returns null.
 *
 * @return the ultimate ancestor of this object if it is a {@code TopProcessor}
 */
TopProcessor getTopProcessor();

@Override
Processor createMutableClone();

@Override
Processor getCurrentVersion();

@Override
Processor withXML(Element node, Map<UUID, Processor> dictionary);

@Override
Processor getNext();
```

```
    @Override
    Processor getPrevious();

}
```

```java
package io.github.samwright.framework.model;

import io.github.samwright.framework.model.common.ChildOf;
import io.github.samwright.framework.model.common.ParentOf;

import java.util.List;

/**
 * A workflow is a linear list of {@link Element} objects which sits in a
 * {@link WorkflowContainer}.
 * <p/>
 * When a workflow processes an input, it has its {@code Element} objects process the data
 * sequentially, for example:
 * {@code input -> lement1 -> data1 -> Element2 -> data2 -> ... -> LastElement -> output}.
 * <p/>
 * The {@code Workflow's} input and output types are independent of its {@code Elements},
 * and are set at the {@code workflow's} construction.
 * <p/>
 * It is perfectly legal to have the wrong input/output types for the {@code workflow},
 * or to have neighbouring elements be type-incompatible. In these situations,
 * the workflow will return {@code false} for {@code isValid()}, and should have a way to
     inform
 * the user (eg. in the associated view there might be connectors between neighbouring
     elements
 * which are red if incompatible).
 * <p/>
 * NB. For two elements to be type-compatible, the previous {@code Element's} output type
     must be
 * castable to the next {@code Element's} input type.
 * <p/>
 * However, a {@code Workflow} MUST be type-compatible with its parent {@code
     WorkflowContainer}.
 * This is done so that all code related to checking neighbouring elements' data types is
     in
 * {@code Workflow} (and nowhere else). Otherwise all {@code WorkflowContainers} would
     have to
 * worry about whether its {@code Workflows} match it's data types. If this assertion is
     ever
 * broken, a {@code ClassCastException} is thrown.
 */
public interface Workflow extends Processor, ChildOf<WorkflowContainer>, ParentOf<Element
    > {

    /**
     * Returns the subset of this workflow's children which are invalid because their
         output type
     * doesn't match the next element's input type. If this workflow's input type doesn't
         match
     * the first element's input type, the list will also contain a {@code null}.
     *
     * @return the child elements that are invalid due to their order in this workflow.
     */
    List<Element> getInvalidlyOrderedElements();
    @Override
    Workflow withChildren(List<Element> newChildren);

    @Override
    Workflow withParent(WorkflowContainer newParent);

    @Override
    Workflow createMutableClone();

    @Override
    Workflow getCurrentVersion();
}
```

```
package io.github.samwright.framework.model;

import io.github.samwright.framework.model.common.ParentOf;
import io.github.samwright.framework.model.helper.Mediator;
import io.github.samwright.framework.model.helper.TypeData;

import java.util.List;

/**
 * A {@code WorkflowContainer} is an {@link Element} that contains one or more {@link
     Workflow}
 * objects.
 * <p/>
 * When a {@code WorkflowContainer} is asked to process a {@link Mediator},
 * it must choose one of its {@link Workflow} children to perform the processing,
 * so only one {@code Mediator} is returned.
 * <p/>
 * When asked to process a training batch, the {@code WorkflowContainer} must process each
      input
 * {@code Mediator} with every {@code Workflow} that could conceivably process it,
 * creating an output {@code Mediator} for each {@code Workflow} and for each input
 * {@code Mediator}.
 */
public interface WorkflowContainer extends Element, ParentOf<Workflow> {

    @Override
    WorkflowContainer withChildren(List<Workflow> newChildren);

    @Override
    WorkflowContainer withParent(Workflow newParent);

    @Override
    WorkflowContainer withTypeData(TypeData newTypeData);

    @Override
    WorkflowContainer createMutableClone();

    @Override
    WorkflowContainer getCurrentVersion();
}
```