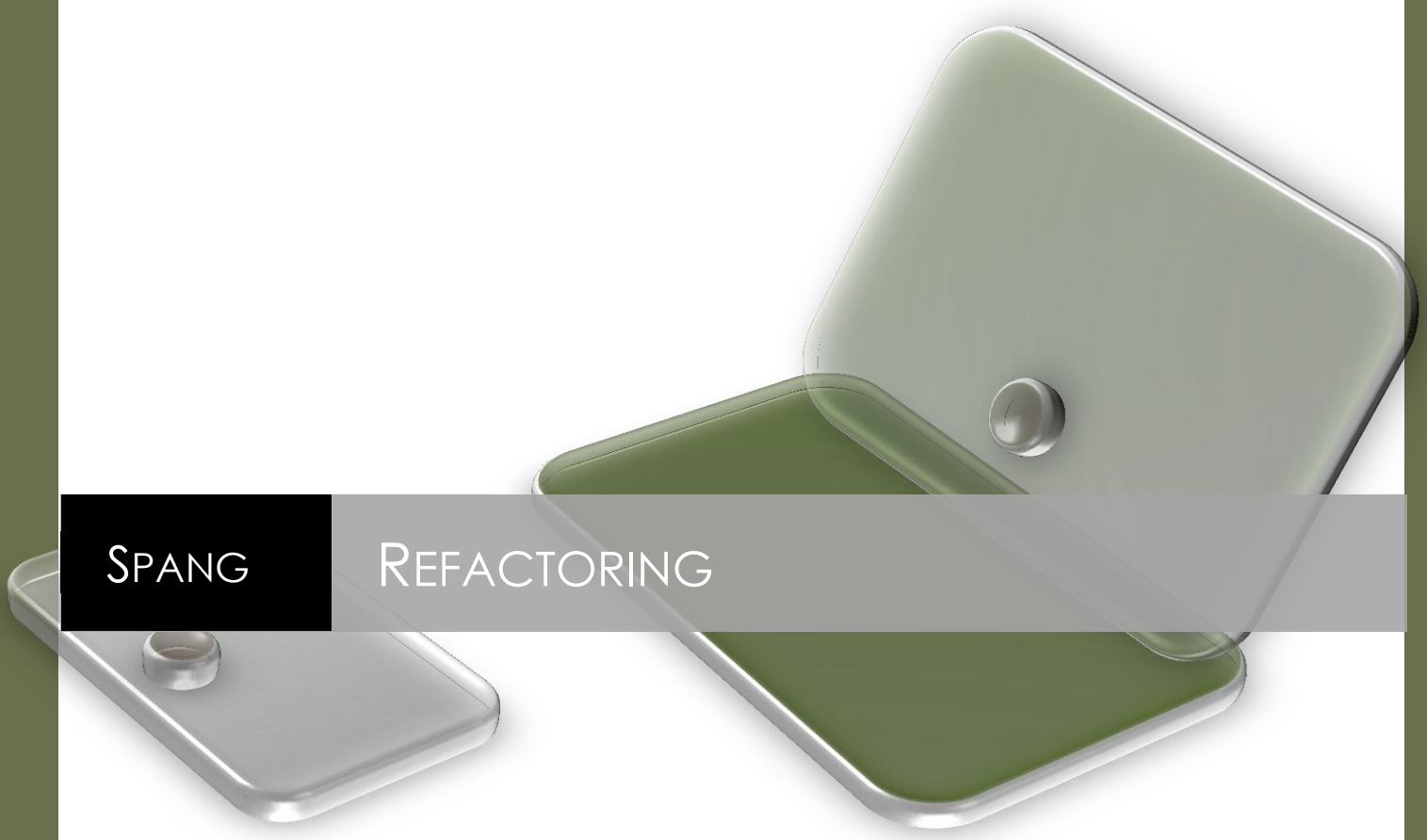
10/5/2012



SPANG REFACTORING

Version 1.9 | Spang

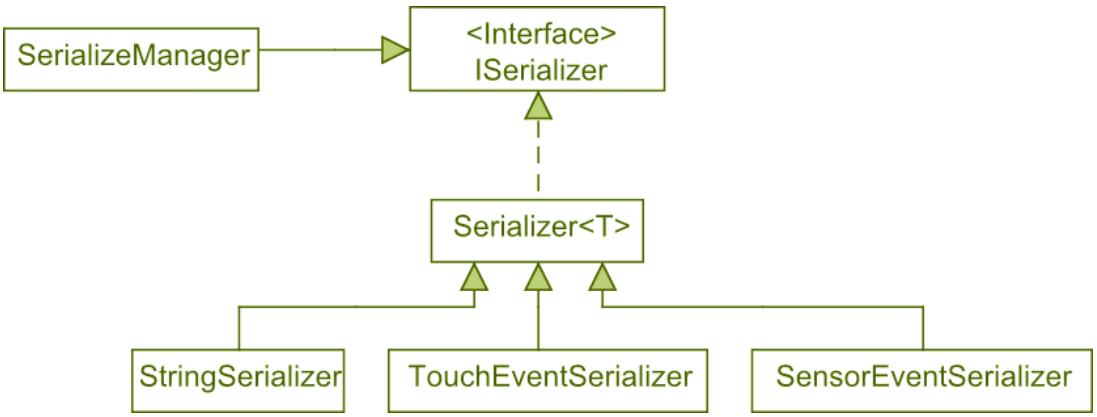
Table of Contents

Introduction

Spang is an API designed to facilitate the sending information from an android device to a computer. This can be any type of information, but the API provides an easy way to send touch, sensors, text, etc.

System architecture

Serialization architecture



Motivated larger design choices

Sensor refactoring 2012-10-10

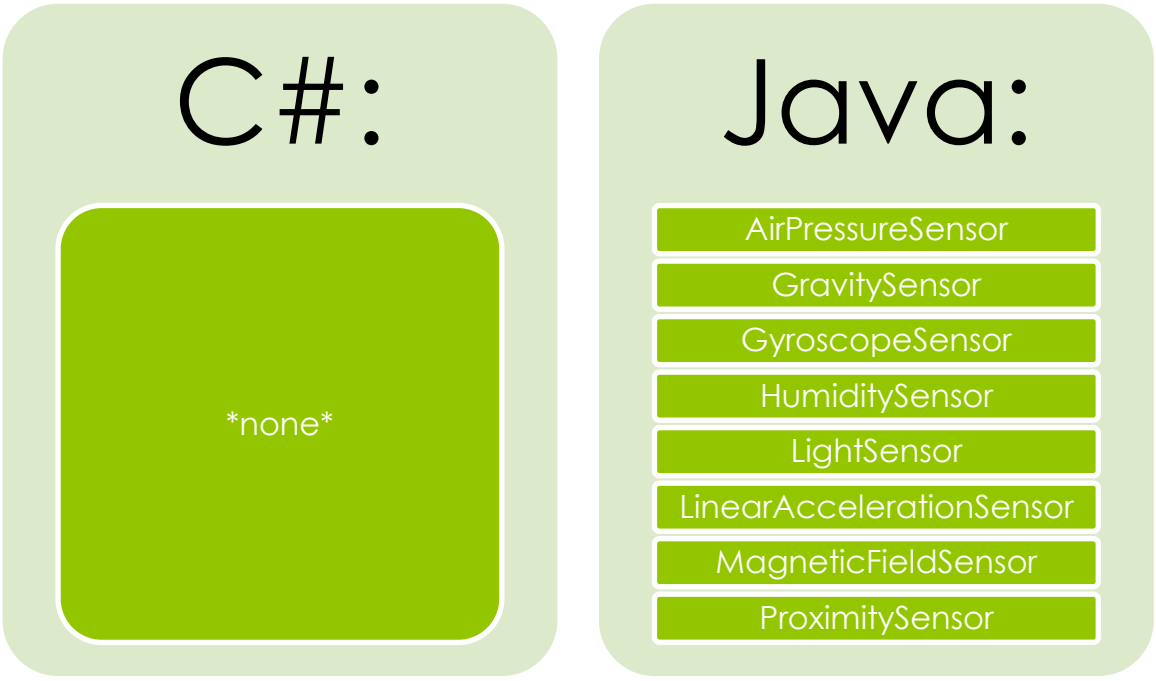
We refactored the way we implemented sensors in Spang. Instead of having one class (implementing ISensor) per sensor we decided to have a SpangSensor class (implementing ISensor) that can represent an arbitrary sensor using instance variables.

Because of this refactoring, a lot of lines of code were lost. It was no great loss though, since most of those lines were duplicated in each sensor class anyway.

The original reason for not designing this way in the first place is the fear for not being able to freely customize the methods for specific sensors, but this can still be achieved by creating a single customized sensor class (see OrientationSensor)

What changed?

The following classes where deleted:



And the following classes were added:

Instead of creating an entire new class in order to add a new sensor, one must simply create a new instance of a SpangSensor object in the SensorList builder class:

```
if(this.manager.getDefaultSensor(Sensor.TYPE_GRAVITY)!=null)
        sensorBindings.put(Sensor.TYPE_GRAVITY, new SpangSensor(manager,
    Sensor.TYPE_GRAVITY, (byte) resources.getInteger(R.integer.Gravity)));
```

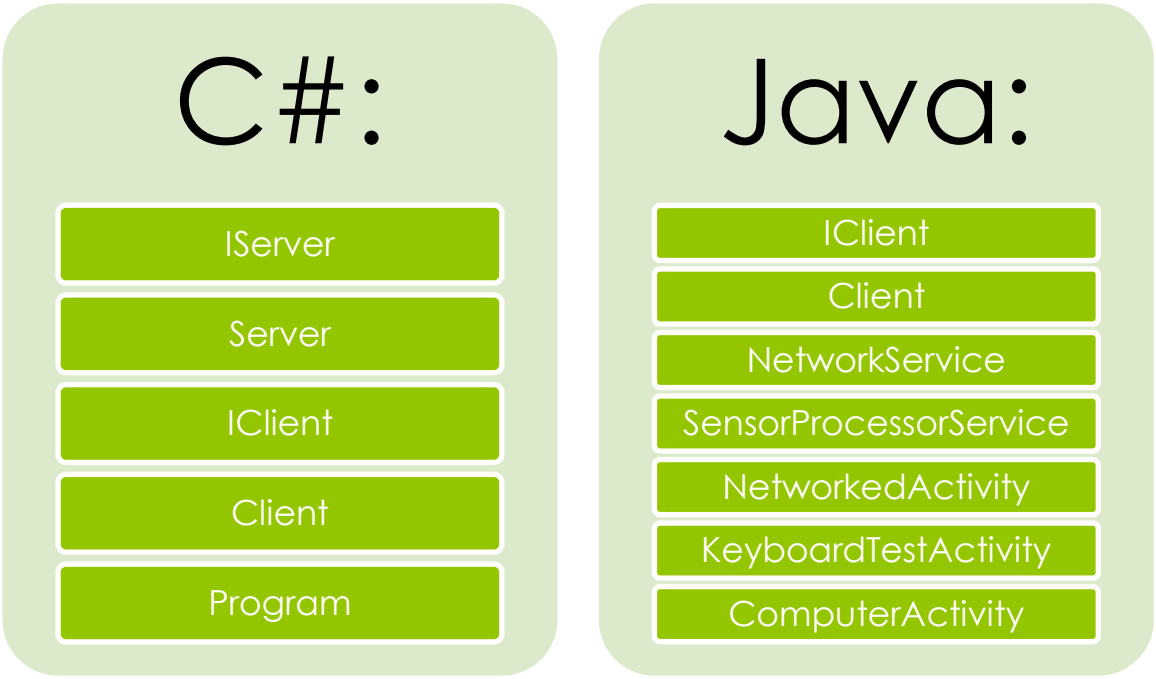
Note that the above code is still not as clean as one might which, but that it still is a lot cleaner than before this refactoring.

Network refactoring 2012-10-12.

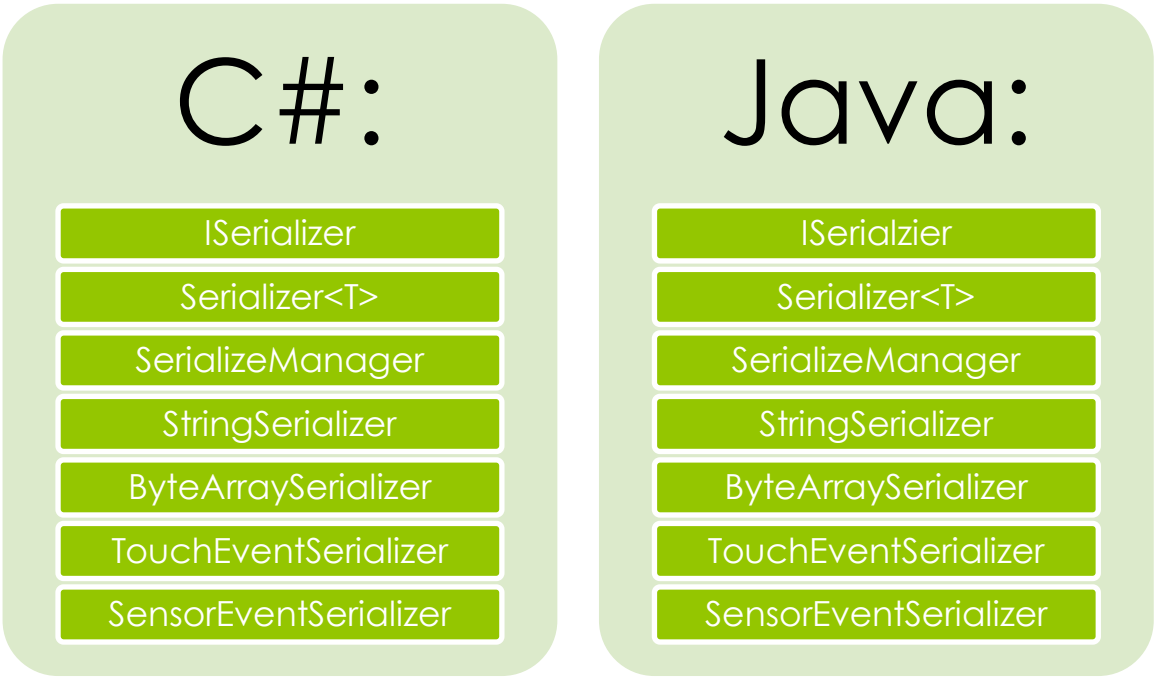
What changed?

A simple serialization API was introduced. This API serializes messages before they are sent over the connection. And deserializes received message. The following types were affected by the change.

The following classes where changed:



And the following classes were added:



Reason:

The end user had to interact with byte arrays as messages. That has worked so far in the project only through lots of trial and error.

Sending was not that hard (if you remembered to add the correct id before each message and pack as byte and not float etc.), but receiving was both error prone and annoying.

Furthermore, debugging was a problem since the code used for sending was scattered all over the place. The Serialization API solves most of these problems.

Before Refactoring

Usage: (End User)

State of code:	•Before refactoring
Code language:	•Java
Sending messages:	•Sending Touch Messages.
Context:	•We are in some class that has access to a NetworkService and Touch events.

```
public void sendTouchEvents(TouchEvent event) {
   //Packers help pack data into byte arrays.
   Packer packer = new Packer();

   //Packs the ID so that the reciving end knows that this is a touch event.
   packer.packByte((byte)R.integer.TouchID);
   //Packs the number of touches currently down on the android device.
   packer.packByte((byte)event.Touches.Length);

   for(int i = 0; i < event.Touches.Length;i++) {
     //Packs location and pressure for each touch.
     packer.packShort((short)event.Touches[i].getX());
     packer.packShort((short)event.Touches[i].getY());
     packer.packByte((byte)(event.Touches[i].getPressure() * 256.0f));
   }

   //Sends the packed data through the NetworkService.
   this.network.send(packer.getPackedData(), Protocol.Reliable);
                                          }
```

State of code:	•Before refactoring
Code language:	•Java
Receiving Messages:	•Receiving some arbitrary message.
Context:	•We are in a class that has access to an instance of IClient.

```
public void addListeners() {
    this.client.addRecivedListeners(new Action1<byte[]>() {
      public void onAction(byte[] message) {
        UnPacker unpacker = new UnPacker(message);
        while(unpacker.remaining() > 0) {
          //Gets the id of the message.
          int id = unpacker.unpackByte();
          if(id == TOUCH_ID) {
            //Decode the touch event.
            TouchEvent event = decodeTouchEvent(unpacker);
            //Use the event for something.
            this.updateTouchStateMachine(event);
          } else if(id == SOME_OTHER_MESSAGE_ID)
            //Decode some other message.
            SomeOtherMessage message = decodeSomeOtherMessage(unpacker);
            //Use the message somehow.
            this.useMessage(message);
          }
          //...More else if
        }
      }
    };
  }
```

Remarks: As can be seen receiving messages are annoying since you must both decode them and use them for something. The code above contains no sorts of error checking so if encounters unexpected messages if will just crash.

State of code:	•Before refactoring
Code language:	•Java
Adding a new type of message:	•Adding the message CoolMessage
Context:	•We are in some class that has access to a NetworkSerivce and CoolMessages

To add a new type of message, all that had to be done was to take a new Unique ID pack it before the message and then pack the contents of the message.

```
public void sendMyNewCoolMessage(CoolMessage message){
   Packer packer = new Packer();
   //Pack ID
   packer.packByte((byte)R.integer.cool_message);
   //Pack contents of message.
   packer.packInt(message.someIntVar);
   packer.packString(message.someStringVar);

   this.network.send(packer.getPackedMessage(), Protocol.Ordered);
}
```

Context:	•We are in the same class as the Receive Arbitrary message example.

```
public void addListeners() {
   this.client.addRecivedListeners(new Action1<byte[]>() {
     public void onAction(byte[] message) {
       UnPacker unpacker = new UnPacker(message);
       while(unpacker.remaining() > 0) {
         int id = unpacker.unpackByte();
         //...Recive alot of messages.
         } else if(id == COOL_MESSAGE_ID) {
           //Decode the cool message.
           CoolMessage message  = this.decodeCoolMessage(unpacker);
           //Do cool stuff with the message.
           this.doCoolStuff(message);
         }
       }
     }
   };
                                    }
```

Remarks: The largest problem with this approach is that new messages cannot even be added to code that you do not control and if you need to receive data in multiple locations the IDs the decode code and needs to be distributed to all the locations.

Another problem is that you need to know that the id is 1 byte long and the first item in a message. This is by far too low level for anyone getting started with the API.

After Refactoring

State of code:	•After refactoring
Code language:	•Java
Sending messages:	•Sending Touch Messages.
Context:	•We are in some class that has access to a NetworkService and Touch events.

```
public void sendTouchEvents(TouchEvent event) {
   //Sends the event.
   this.network.send(event, Protocol.Reliable);
}
```

State of code:	•After refactoring
Code language:	•Java
Receiving Messages:	•Receiving some arbitrary message.
Context:	•We are in a class that has access to an instance of IClient.

```
public void addListeners() {
   this.client.addRecivedListeners(new Action1<byte[]>() {
     public void onAction(Object message) {

        if(message instanceof TouchEvent)  {
          //Use the touch event for something.
          this.updateTouchStateMachine((TouchEvent)message);
        } else if(message instanceof SomeOtherMessage) {
          //Use the message somehow.
          this.useMessage((SomeOtherMessage)message);
        }

       }
     }
   };
}
```

Remarks: The Receiving code is a lot simpler to follow. We still use if else statements to know the type of the data, but we no longer need to decode them ourselves.

State of code:	•After refactoring
Code language:	•Java
Adding a new type of message:	•Adding the message CoolMessage
Context:	•We are in some class that has access to a NetworkSerivce and CoolMessages

First we create a class that can serialize and deserialize the message.

```
public class CoolMessageSerializer extends Serializer<CoolMessage>{

    public Class<?> getSerializableType() {
      return CoolMessage.class;
    }

    public void serializeInternal(Packer packer, CoolMessage message)
      {
      //Pack contents of message.
      packer.packInt(message.someIntVar);
      packer.packString(message.someStringVar);
           }

    public SensorEvent deserialize(UnPacker unpacker)
       {
            int id = unpacker.unpackInt();
      String string = unpacker.unpackString();

            return new CoolMessage(id, string);
    }
}
```

Context:

- We are in a class that has access to an instance of IClient or IServer

```
public void someMethod() {

   //If we are on the client side.
   this.client.registerSerializer(new SomeOtherSerializer());
     this.client.registerSerializer(new CoolMessageSerializer());

   //If we are on the server side.
   this.server.registerSerializer(new SomeOtherSerializer());
   this.server.registerSerializer(new CoolMessageSerializer());
}
```

Remarks: At the time of this writing the order in which the serializers are added is EXTREMLY IMPORTANT since the order they are added decides their ID. This is still subject to change.

Now the new message is ready to be received and send over the network. To use the new message simply add another else if statement in the receiving message listener checking for the CoolMessage type.

Touch handling refactoring

Instead of handling the users touch events in the Android application code; we decided to move the implementation to the PC server.

The idea was to simplify API usage for Android developers, and to provide a standard way of interpreting touch input on the PC side.

After the refactoring, the only thing an Android developer needs to write to control the mouse is a View sending all TouchEvents (in the onTouchEvent method) using a NetworkService. This makes the View code more focused on the actual visuals, instead of getting filled with hundreds of rows of confusing mouse control logic.

Another consequence of this refactoring is that a developer who wants to create a custom way of handling touches sent from an Android device now does this in the PC application.

We definitely prefer this approach, since the Android device should only be responsible for sending input (like a regular mouse sends input to a computer through an USB cable). The PC application then interprets and acts based on said input.