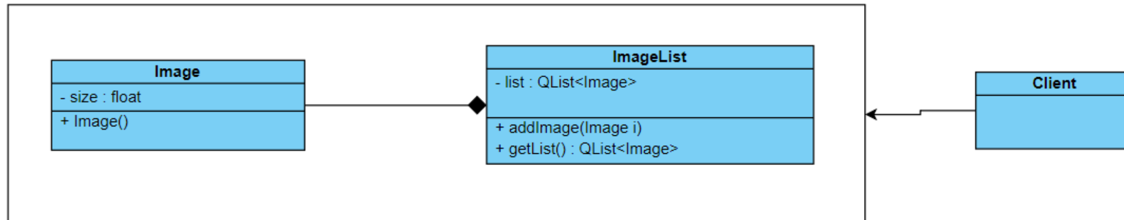
Quickly done by Edmund

If you notice any mistakes, please fix them ☺. If any, there aren't huge, maybe just typos and obvious. Someone taught you differently? Next time you see them, punch them in the face :D

Question 1

1.1

*Really not much to do here.



1.2

```
class Image : public QObject
{
	Q_OBJECT
	Q_PROPERTY(float size READ getSize WRITE setSize)

	public:
		Image();
		float getSize();
		void setSize(float f);
	private:
		float size;
};
```

*Optionally, you could also make ImageList a QObject and then include Q_OBJECT in the class.

1.3

a)

Input Kludge

b)

[1-9]\d+(\.\d+)?

c)

```
QValidator *validString = new QValidator(requiredFormat);
```

Question 2

2.1

```
QStringList InputOutput::toString()
{
	QStringList stringList;
	foreach(Image i, list)
	{
		const QMetaObject *meta = i.staticMetaObject(); //If not a
		                                                  pointer
		for(int i = meta->propertyOffset(); i < meta->propertyCount();
		  i++)
		{
			QMetaProperty metaProp = meta->property(i);
			QString propName = QString(metaProp.name());
			QString value = metaProp.read(&i).toString(); // OR: =
			 i.property(propName).toString();

			stringList.append(propName + ":" + value);
		}

		//For dyanmic properties
		QList<QByteArray> dynamicProps = i.dynamicPropertyNames();

		for(int i = 0; i < dynamicProps.size(); i++)
		{
			QString propName = QString(dynamicProps[i]);
			QString value = i.property(propName);

			stringList.append(propName + ":" + value);
		}

		return stringList.join(";");
	}
}
```

2.2

It's an object that carries information (meta-data) about another object. It is used in reflective programming (to allow objects to be inspected at runtime).

2.3.

A QMetaObject is created before runtime by the MOC (Meta Object Compiler). Since static objects exist during compile time, their information is carried (in QMetaProperties) by the QMetaObject. Dynamic properties are however not known by a QMetaObject because they are only created during runtime.

2.4

***1.**

```
QDomDocument doc;
doc.setContent(&file);
```

***2.**

```
root.tagName() == “ImageList”
```

***3.**
```
QDomNode = root.firstChild();
```

***4. – 5.**
```
QDomElement element = node.toElement();
```

***6.**
```
QString size = element.attribute(“size”);
```

***7.**
```
node = node.nextSibling();
```

Question 3

3.1.

QStandardItemModel.
If more properties are added, these can also be added to the model and the view will automatically show the new data.

3.2

QAbstractTableModel is an abstract class and cannot be instantiated.
The function to set a view’s model is setModel NOT setView used there.

3.3

A QStringLIstModel must be used with a QListView not independently.
A QListWidget (convenience class) can be used independently since it combines both the view and the model.

3.4.

In the MVC pattern, all three components automatically respond to a change of state in the other components. A view will automatically update if the data from the model changes. A model will also automatically update with any view changes. The controller automatically responds to any updates in the view and model.
These components can therefore be taken as observing each other and this follows the observer design pattern.
**Feel free to rephrase as seen fit.*

3.5

The C part represents the Controller. The controller in MVC is used to control data flow between the Model and the View.
In the Qt framework, a delegate is used to render items for editing on the view. A delegate can therefore be seen as playing the role of a controller.
**I just summarised. Read further.*

Question 4

**Here, ImageList is your originator while BackupImageList is the Memento.*

4.1.

```
class BackupImageList
{
      private:
            friend class ImageList;
            BackupImageList();
            QList<Image> getState();
            void setState(QList<Image> i);
            QList<Image> images;
};
```

4.2.

*Remember you need to introduce two functions in your originator: One to create the memento and another to restore.

```
BackImageList ImageList::createMemento()
{
      BackupImageList backup;
      backup.setState(list);

      return backup;
}
```

4.3.

The memento stores data (mementos) in memory using the Caretaker. The caretaker cannot write its objects to an external medium while the serializer can write data to an external destination e.g. a file.

**Read around – that's all I could think of now.*

Question 5

**Remember: The recommended approach is NOT to subclass QThread. Instead you should have a worker class and a client.*

5.1

```
class SearchImages : public QObject
{
	Q_OBJECT

	public:
		SearchImages(QList<Image> imageList);

	public slots:
		void searchImage(); //Must NOT return

	signals:
		void imageFound(Image);
		void finished(); //I'm used to always putting this

	private:
		QList<Image> images;
};
```

5.2.

**You can choose to read the output line by line OR wait for it to finish and read everything all at once. Question doesn't specify. Here is one way of doing it. Just be aware of all the necessary signals and functions of QProcess.*

**From my understanding of this question, this process needs to know the filename so it can use it to call that given function. I must be right!*

```
QProcess *process = new QProcess();
QStringList arguments;
arguments.append(filename); //assuming you have a filename variable
somewhere..

connect(process, SIGNAL(readyReadStandardOutput()), this,
SLOT(manage()));

process->start("imagesizes.exe", arguments);
```

5.3.

** I'd skip this one and leave the exam room early ☺*

** Google it!*

--- END --- and

Cheers,

Edmund Charumbira

edmund@vtutoronline.com