Block by Block Towards IoT Applications

With Reactive Blocks and Java you can assemble applications for the Internet of Things from building blocks.

by Frank Alexander Kraemer

Reactive Blocks is a plugin for Eclipse, which allows to systematically create applications from building blocks. The building blocks can encapsulate Java code and have special interface descriptions, which goes beyond traditional interfaces. Therefore, blocks can be easily shared with other developers and be reused easily. The type of graphical representation makes it easy to specify and synchronize parallel processes. Because of these properties, Reactive Blocks is especially suitable for the future applications for the Internet of Things. The tool can also be used wherever systems must handle multiple tasks simultaneously.

The Magic of Building Blocks

Probably everyone has experienced that there is something magical about building blocks, and many have accomplished their first engineering tasks using building blocks. Also software is often constructed from building blocks, since programs can be created from libraries. But if one looks at the code of applications, little remains of the magic of building blocks: Already while creating applications it is easy to lose the overview. And if one wants to get familiar with code created by others, it gets even more difficult.

Reactive Blocks [1] is a tool that enables the development from building blocks. Blocks are graphical models, which can easily be connected together. In this way one can keep the overview of the application. Synchronizations and problems with concurrency can be described very well. Therefore, Reactive Blocks is excellent to meet the challenges of creating Internet of Things, IoT, applications: Such applications can be composed systematically from libraries.

If you are a typical programmer and feel uneasy about graphical tools, we can calm you down. From the beginning, we thought through how the graphical elements and Java can be developed together. Therefore, Reactive Blocks is integrated with the Java Tools in Eclipse, and the graphic description complements Java naturally.

The Internet of Things

The expectations for the IoT are huge. But no one really knows, how exactly IoT applications will turn out, and companies approach the market with different strategies. Mike Milinkovich argues in his blog post [2] that open source will play an important role. And, like in the usual Internet, standards will be pivotal to connect the things together. The Eclipse IoT working group implements some of these protocols and frameworks. But what about the applications? They need to ensure that everything fits nicely together. In that respect, IoT applications have quite special requirements and challenges.

Today's IoT systems can be roughly divided into three levels on which programs run:

- The smallest units, sensors and control modules, to collect data and control devices. These units only have a small microchip and are optimized for low power consumption and low manufacturing costs.
- Gateways are nodes in the system that have several tasks. They can do far more than only forward data. Equipped with more powerful processors, they can execute complex applications.
- Servers run typical back-end functions, such as the storage and analysis of data in large volumes, the integration to business processes or the remote control through web interfaces.

Gateways as Nodes

Once the smallest devices directly communicate over IP with any other device, the borders between these three levels will further blur. We can call this the "real" Internet of Things. Gateways, however, will not disappear: Even if the smallest units can communicate directly, they need a connection to the network, for example, via Bluetooth, Wifi or Zigbee. In addition, gateways become nodes on which applications can also maintain the local operation, if the network is down. Already today such gateways exist. For example as onboard units to accompany the transport of goods, for controlling a traffic junction, or as a gateway for home automation.

Challenges of IoT Applications

Most IoT applications share the same challenges. It is also interesting, that also relatively simple use cases in applications require more complex solutions than one would initially expect.

- Programs on gateways work without direct supervision and are administered remotely. This means that the programs must be robust. If one had to press a button on the device to reboot upon an error, it would quickly be an expensive affair. It also means that it must be possible to update each application remotely.
- During operation, the gateways often have the task to combine and process incoming data quickly, and manage connections between the network and hardware. There it is important to adhere to all protocols, and ensure that the communication works correctly.
- Even if parts of the network fail, applications on the gateway must still work correctly. Depending on the application, data can be cached and decisions can be made locally. To put it in a nutshell: Even if the connection to

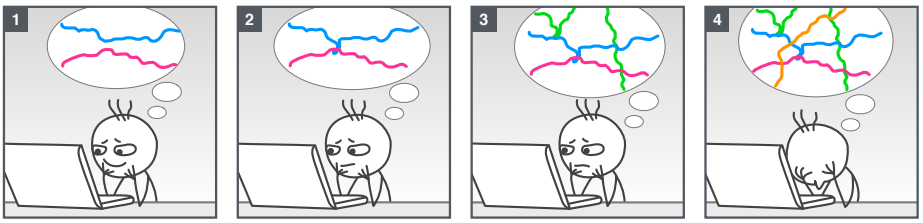


Figure 1: Programming concurrency is difficult. (1) Two parallel processes are easy to implement. (2) Synchronizing two parallel processes is okay for most programmers as well. With three processes, all gets much more complex and difficult. (4) At some point, any programmer will give up.

the Internet is down, you would still like to open your garage door that is controlled by your home gateway.

Such requirements are serious challenges when programming applications, since they involve a high degree of concurrency. Concurrency occurs when code must handle several things at the same time. For example, to communicate with hardware via Modbus, and simultaneously examine the incoming data for certain patterns, all while being prepared for new commands arriving from the server. This requires developers to care about difficult issues:

- Are all interfaces used properly?
- What happens if the network is temporarily interrupted? Will the application still work reasonably well?
- Are all combinations of errors taken into account?

Concurrency is Difficult

Experience shows that programs with a high degree of concurrency appear simple in the beginning, but quickly get more difficult. Our developer in Figure 1 begins with two concurrent processes that have initially not much to do with each other. In Java, this is easily accomplished with two threads.

Synchronizing two processes is also okay for most programmers. Once a third process comes into play, for example because an additional hardware component is added, everything gets more complicated. And at some point, any developer will give up. In cases where many processes need to be in sync one takes often for granted that the code contains subtle defects like race conditions. To search for them is difficult. To correct them is often extremely complicated and expensive. And if the program works correctly, it is very likely that new errors are added once applications have to be changed ever so slightly due to changing customer requirements.

As a consequence, programmers often try to reduce the degree of concurrency used in a program. This means that tasks that could actually be executed in parallel are instead carried out in series. This makes the processing of data slower by factors, and does not utilize the hardware optimally.

Java Interfaces Are Not Enough

Because applications in gateways must coordinate the behavior of various subsystems properly, it is important to combine the subsystems correctly. The standard interfaces written in Java describe which methods and which data types a subsystem can understand. If methods or types do not match, the compiler or the Java editor report errors even before one runs the system. That, however, is the easiest part. In many cases, it is critical that the methods of an interface are used in a specific order. A communication socket, for instance, must first be initialized before it can send messages.

Another aspect is the timing behavior of method calls. Once we do things concurrently, it is important to know which methods block the calling process. Unexpected blocking may lead to situations where we miss other events entirely or react to them much too late.

Formally, a Java interface says nothing about blocking methods or the necessary sequence of method calls. Though this can be described in the documentation, we know that documentation is not always maintained properly. Also, not all documentation is read at all times. The biggest disadvantage, however, is that developers do not get a direct warning from the compiler or editor about such errors, simply because the compiler or editor cannot read or understand the documentation. Such incorrect use of interfaces is often only found at run-time.

Reactive Blocks

To solve these challenges, Reactive Blocks defines special building blocks, as shown in Figure 2. For many tasks one can already find libraries of blocks that can be downloaded directly from within the tool. These building blocks are defined by a combination of graphical models and Java code. Graphical models are UML activities and state machines. The blocks can be connected together to form applications. One can of course also create own blocks, which can also share with others. In principle, everything that runs in Java can be encapsulated in a building block.

To handle concurrency properly, Reactive Blocks can analyze the applications simply with the push of a button. And when everything is right, the built-in code generator writes the necessary code that assembles the components. In this way, one can in a relatively short time create complete IoT applications. In the following, we will take a closer look at these features.

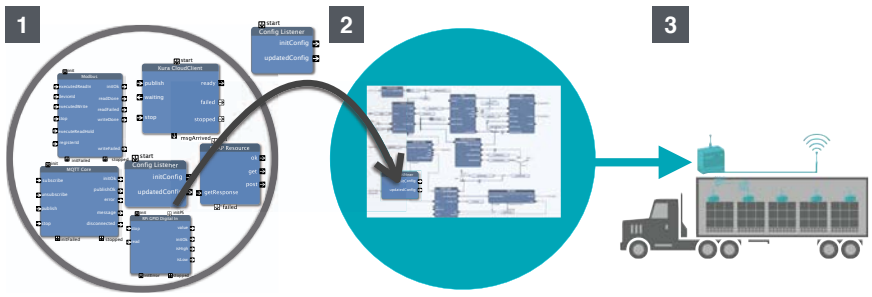


Figure 2: The coarse work flow with Reactive Blocks: Building Blocks from libraries are connected to complete applications. Everything is analyzed by the editor. If everything is correct, Java code that connects the blocks is connected automatically. The resulting application can then be used for example in an on-board unit or other gateways.

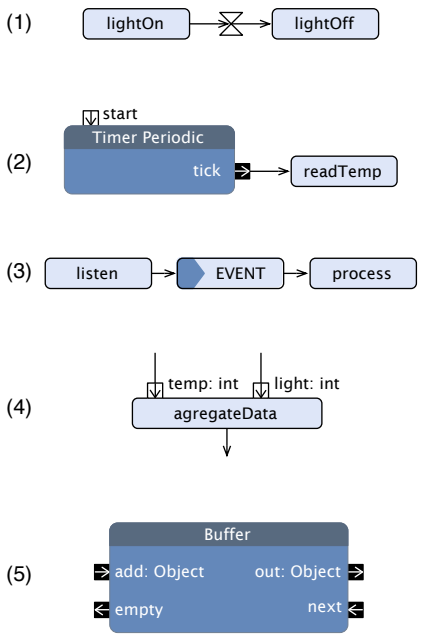
Graphically and Clearly

What is surprisingly difficult in program code is simple to express by graphics. The UML activities work like data flows. Figure 3 illustrates some of the graphic elements. The light blue elements refer to Java methods that can be edited directly in Eclipse.

- When two Java methods should be executed with a delay, one can set a timer between (1). The code for the timer and the invocation of the subsequent operations is then generated automatically. In addition to the basic timer, advanced timers are available in the form of special blocks, for example for periodic tasks (2).
- If certain methods block, their content is placed into a separate thread. To find out when the result is available, special events are used (3). Their output can then be linked directly with the subsequent logic. In pure code the necessary callbacks would make this look cluttered.
- Often one needs data from multiple sources in order to make a decision. To await the arrival of two data items one can use a joining operation (4). Again, the necessary code for this logic is generated.
- A useful block is also the buffer (5). It decouples two processes from each other so that they can work at their own pace. The first process, for example, could continuously receive new data points. The second process could send them to a server with an acknowledgement, which takes more or less time. The buffer uses internally a Java list of objects that the first project produces. In comparison to the simple Java list, however, the buffer knows about the state of the processes and triggers the second process at the right times.

This means that synchronizations become a lot easier. Another advantage is that you can often get an impression of how an application works at first glance . If you work in a team, you can simply project the block diagram on a wall and explain problems by pointing at them with your hands. During the implementation of customer projects, we have also found that even non-programmers can get a rough understanding of the application in this way. This helps to gain confidence and to communicate critical issues accurately.

Figure 3: The graphical data flows in Reactive Blocks connect Java methods with each other. The code is then generated automatically.



Interfaces for Building Blocks

To explain the sequence and the coarse temporal behavior of a block, each block has an extended interface description in the form of a state machine. This indicates which states a block has, and in which states you can use specific parameters. An example is the building block for the MQTT protocol [3] in Fig. 4. The state machine states that MQTT must be initialized first via init. This process takes some time, because the client must connect to the broker. This is described by the state 'connecting'. In this state, our program can do other things instead of being blocked by init.

Once the connection is established, we are informed about that via ready. And from that moment on (and not before), we can Send MQTT messages or receive them, via pins send or message, respectively.

One could of course say that such a behavior is obvious. However, there are much more complicated protocols and functions for which the behavior is not so obvious. Most notably, however, is that the tool understands now, how the MQTT block works. In this way, it can generate efficient code and also analyze the application automatically.

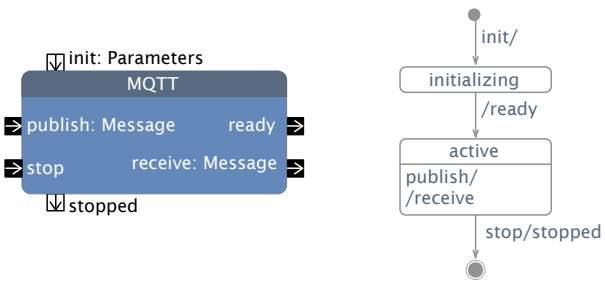


Figure 4: The block for MQTT. The advanced interface allows the tool to check whether the protocol is properly integrated into the application.

Libraries of Building Blocks

A building blocks kit is of course only as good as the individual building blocks that it offers. If you start with Reactive Blocks, you can choose from several libraries with ready-to-use blocks.

- The blocks for MQTT allow you to easily communicate through a broker. Depending on how you want to use MQTT, there are several blocks that allow this. In addition to the general block for MQTT there are also more specialized blocks that only publish or subscribe, for example. There is also a robust version that automatically buffers messages and reconnects upon network problems. Internally, the block uses the Paho library of Eclipse [4].
- The blocks for Kura make it easy to monitor and configure gateway applications remotely. The Config block receives the values that have been changed via the Kura web console via the parameter newConfig. The code generator then configures the completed application to be executed directly as Kura application. Kura is also an Eclipse IoT project [5].
- The blocks for CoAP [6], which internally use Californium [7] let applications communicate over CoAP. There are blocks for CoAP servers, CoAP clients, as well as predefined CoAP resources.
- A typical feature in IoT systems is to log data in the cloud. There are numerous services to do this, such as AirVantage from Sierra Wireless, ESF from Eurotech, or Xively. Conveniently, libraries for these services already exist.

In addition to these typical IoT protocols, there are plenty of other libraries, such as Modbus, serialization of data or for storage of events. Of course you can also create any custom blocks and libraries and share them with others. These can be blocks for new APIs or protocols, hardware, generally useful functions or special blocks for your own system.

Connecting Blocks Together

To create an application, you can drag-and drop blocks from the libraries and connect them in the editor. It takes, for instance, only a few blocks to build an application that records images and forwards them via MQTT. This allows you to quickly experiment with new technologies, try out new ideas and show customers a prototype within a short time. If you want to extend such a prototype, you don't need to start from scratch, but you can extend the prototype in several stages until you have the final application. You can do this by adding more blocks, or by refining blocks to suit your own needs. Reactive Blocks is thus suitable both for experimenting with prototypes but also for production quality.

Integration with Java and Eclipse

A challenge of graphical tools is often how they deal with code. In Reactive Blocks this is solved elegantly and practically: Blocks can point directly to Java methods. Double-clicking the graphic operations directly leads to the corresponding Java methods. The Java method can basically do anything that is possible in Java. And since all methods of a block are collected within a class, they can be edited directly with the Java Tools (JDT) in Eclipse. Programming works just as before and Java developers will feel right at home. The graphics are only used once larger modules (the blocks) are connected with each other, or to describe synchronizations as described above. In developing Reactive Blocks, we applied the principle that things that work great in code can still be programmed. Anything that has to do with concurrency, on the other side, is done graphically. This keeps the code clean and increases productivity.

Perfect Generated Code for Synchronization

Once an application is complete, it can be built. This is done using the builder of Reactive Blocks that creates a complete Java project from the structure of building blocks. Since the building blocks are precise mathematical structures, the builder can transform their logic into very effective code. Several parallel flows that in a manual implementation would need several threads can be efficiently mapped into a single one. The various synchronizations in Reactive Blocks (the joins, for example) are all processed event-driven. This approach creates code which can very effectively execute many events. And that's really the code no programmer even wants to write manually. To write such code is boring and so error prone that it's better to not even try.

Of course, the builder also generates all necessary boilerplate code that is needed for a complete application. Depending on which platform the application should run, there are several possibilities how the code can be deployed:

- As a standing-alone Java project, which can be run either directly or exported as an runnable jar file.
- As an OSGi bundle that can be executed directly in an OSGi framework.

Automatic Analysis

Only very few programmers analyze their applications in a formal way. Though powerful tools exist, they are often not used. One reason is that such tools are complicated to use. Another reason is that one first has to develop a simplified model that can then be analyzed mathematically. With only the code as source, this is a difficult task. As a result, only critical software is evaluated in that way, and often only partial.

In Reactive Blocks we have the analysis model already in the form of graphical diagrams. Therefore, the analysis works automatically, simply by pressing a button. In this way you can find out quickly whether the system can get stuck in a deadlock and whether you use all blocks according to their contract. If the analysis finds a problem, you can quickly see an animation of what is going wrong. The editor will show you step by step what happened, and why a particular situation is problematic. This then makes it much easier to solve complicated situations and synchronizations properly.

Complementary to Testing

Good developers test their programs. Also with Reactive Blocks, testing of the individual Java methods is till necessary. The tests ensure that the methods work as intended. Testing, however, is only as good as the test cases, and they have to be written first manually. And writing test cases for concurrent applications is especially difficult. The automatic analysis of Reactive Blocks can test concurrency-related issues without that one has to write test cases in the first place. So the testing can focus its effort on the individual Java methods, which of course is much easier.

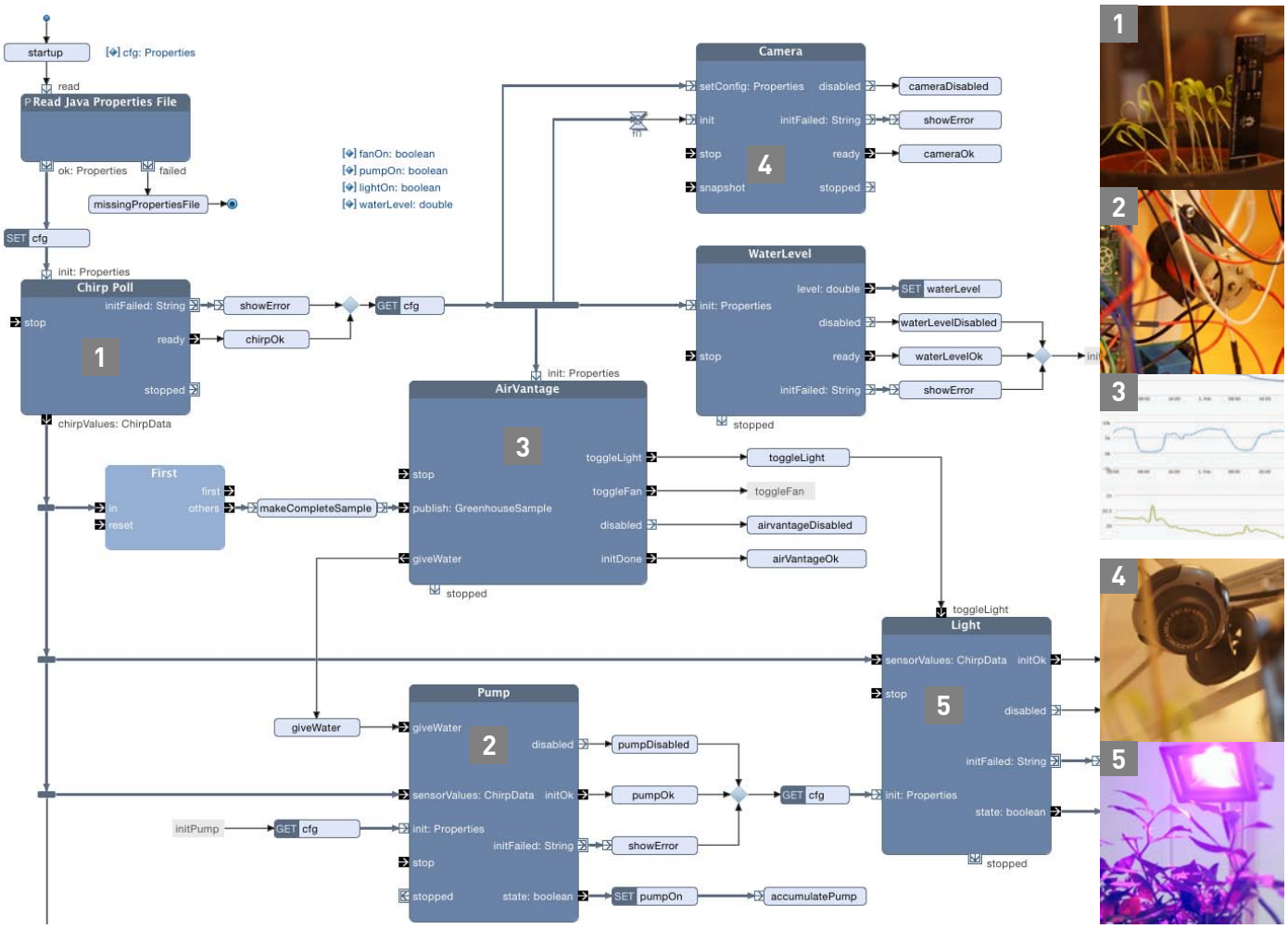


Figure 5: The application for the Raspberry Pi, which monitors and controls the greenhouse remotely. The application in Reactive Blocks and Eclipse is shown to the left. To the right images of what the building blocks represent.

A Greenhouse as an Example

As an example, we use a Raspberry Pi to control lighting, watering and ventilation of a greenhouse. The program sends all data and images into the network, so that plants can be monitored remotely. Figure 5 shows the program in Reactive Blocks. For each function there is a special block.

1. A sensor measures temperature, humidity and light intensity directly at the plant. The sensor communicates with the Raspberry Pi using the GPIO pins and an I2C bus.
2. The pump is activated when the soil is too dry. The block also monitors that the pump is not watering too much.
3. All the data is a sent via a block into the network (here to AirVantage from Sierra Wireless), so that data can can be monitored remotely.
4. A camera sends images via USB to the Raspberry Pi. From there they are loaded via MQTT to our website.
5. If it gets too dark, a special lamp is switched on.

Those who want to build a greenhouse on their own find instructions on our website [1].

Learn More

Reactive Blocks can be installed from the Eclipse Marketplace. With the free version one can create new applications without any limitations, as long as they are shared with other developers. More information, examples and tutorials are available at Bitreactive [1]. In addition to the instructions for the greenhouse, there is also the IoT trail, which introduces current IoT technology.

Links & Literature

1) Bitreactive http://bitreactive.com
2) The IoT will be built on open source http://blog.bosch-si.com/categories/technology/2014/10/the-iot-will-be-built-on-open-source/
3) MQTT protocol http://mqtt.org
4) Eclipse Paho http://www.eclipse.org/paho/
5) Eclipse Kura http://eclipse.org/kura/
6) CoAP http://coap.technology
7) Eclipse californium http://www.eclipse.org/californium/

bitreactive