Message-Bus im Eigenbau

So kommt die Nachricht an

clean code developer

Im vorangegangenen Heft haben Sie einen Fahrkartenautomaten erstellt. Jetzt kommt noch ein Bus dazu. Und zwar einer, der Nachrichten transportiert.

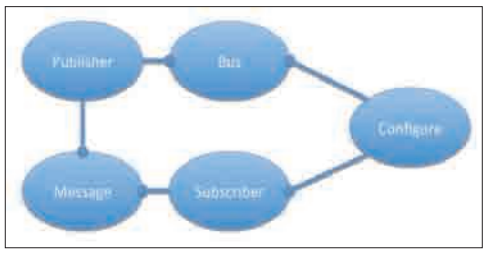
dnpCode: A1109DojoLoesung

Stefan Lieser ist Softwareentwickler aus Leidenschaft. Nach seinem Informatikstudium mit Schwerpunkt auf Softwaretechnik hat er sich intensiv mit Patterns und Principles auseinandergesetzt. Er arbeitet als Berater und Trainer, hält zahlreiche Vorträge und hat gemeinsam mit Ralf Westphal die Initiative Clean Code Developer ins Leben gerufen. Sie erreichen ihn unter stefan@lieser-online.de oder lieser-online.de/blog.

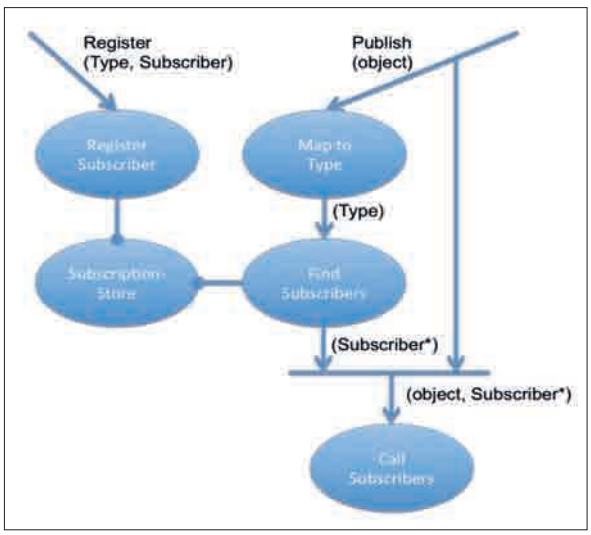
Ein Message-Bus hat die Aufgabe, Nachrichten von Sendern zu Empfängern zu übertragen. Der Sender wird im Allgemeinen als Publisher, der Empfänger als Subscriber bezeichnet. Die Nachrichten werden beim Transport vom Publisher zum Subscriber nicht unmittelbar übertragen, sondern dazwischen sitzt der Message-Bus als Übertragungsinstanz. Das hat den Vorteil, dass Publisher und Subscriber keine direkte Abhängigkeit zueinander haben. Der Einsatz eines Message-Busses kann also die Entkopplung von Funktionseinheiten fördern.

Natürlich sind Publisher und Subscriber vom Typ der Nachrichten abhängig. Schließlich muss der Empfänger mit der Nachricht des Senders etwas anfangen können. Folglich müssen sich beide auf einen gemeinsamen Typ verständigen. Abbildung 1 zeigt die Abhängigkeiten von Publisher, Subscriber und Bus.

- Der Publisher ist vom Bus abhängig, da er Nachrichten an diesen übergeben muss. Ferner ist er abhängig vom Nachrichtentyp.
- Der Subscriber ist lediglich vom Nachrichtentyp abhängig.
- Damit der Bus weiß, an welche Subscriber er Nachrichten übermitteln soll, muss er die Subscriber kennen. Dafür muss eine Funktionseinheit während der Konfigurationsphase einer Anwendung sorgen. Diese Funktionseinheit, in der Abbildung mit *Configure* bezeichnet, ist vom Bus und den Subscribern abhängig.



[Abb. 1] Abhängigkeiten von Publisher, Subscriber und Bus.



[Abb. 2] Der Bus als Platine.

In der Praxis sind von den Nachrichtentypen unter Umständen viele Funktionseinheiten abhängig. Änderungen am Nachrichtentyp betreffen mindestens Publisher und Subscriber. Verwenden mehrere Publisher und/oder Subscriber den Nachrichtentyp, sind im Ergebnis viele Funktionseinheiten von diesem Typ abhängig. Daraus folgt, dass von Änderungen am Nachrichtentyp potenziell viele abhängige Funktionseinheiten betroffen sind.

Damit dies keine negativen Auswirkungen hat, sollten die Nachrichtentypen so einfach wie möglich gehalten werden. Sind nämlich die Nachrichtentypen reine Datenklassen ohne Logik, bleiben die Auswirkungen einer Änderung gering. Sobald die Nachrichtentypen jedoch Logik enthalten, steigt das Risiko, dass Änderungen an der Logik Konsequenzen für die verschiedenen davon abhängigen Funktionseinheiten nach sich ziehen.

API-Entwurf

Nach diesen ersten Vorüberlegungen habe ich begonnen, ein paar Tests zu schreiben, um das API des Busses zu entwerfen. Um zu prüfen, ob das so entworfene API auch funktioniert, habe ich den Bus in einer minimalen Form als Spike implementiert. Dabei habe ich zunächst auf ein Modell verzichtet und den Bus direkt in einer Klasse *Bus* implementiert. Als Ergebnis wusste ich, dass das API prinzipiell funktioniert. Listing 1 zeigt den ersten Test.

Die Implementation zu den Tests zeigt Listing 2. Die Registrierung der Subscriber erfolgt mithilfe eines Dictionarys. Als Schlüssel für das Dictionary verwende ich den Nachrichtentyp. Da es für einen Typ mehrere Subscriber geben kann, ist der Wertetyp des Dictionarys eine Liste von *object*. Jedes Objekt repräsentiert hier eine Instanz eines Subscribers. Beim Ausliefern der Nachrichten werden die Subscriber auf den generischen Typ *ISubscriberOf<T>* gecastet. Hier zeigen sich bereits die Herausforderungen im Umgang mit generischen Typen. Die Spike-Version des Busses findet sich auf der dem Heft beiliegenden DVD in der Datei *BusSpike.cs*.

Nachdem auf diese Weise das API feststand, habe ich die Implementation des Busses modelliert. Dabei bin ich zunächst von einem minimalen Funktionsumfang ausgegangen:

- Subscriber können im Bus registriert werden.
- Veröffentlichte Nachrichten werden vom Bus zu den Subscribern übertragen.

Listing 3 zeigt die Interfaces. Natürlich verwende ich für die Modellierung wieder Flow-Design und setze den Entwurf mit Event-Based Components (EBCs) um. Beim API des Busses handelt es sich um ein „klassisches API". Damit meine ich, dass das API aus Methoden aufgebaut ist und selbst keine Event-Based Component nutzt. Das hindert mich aber natürlich nicht daran, innerhalb der Implementation des Busses EBCs zu verwenden.

Shared State

Die beiden Funktionen *Register* und *Publish* müssen auf demselben Zustand arbeiten: *Register* registriert einen neuen Subscriber, während *Publish* aus der Liste aller Subscriber diejenigen ermitteln muss, denen die Nachricht zugestellt werden muss. Diesen *Shared State* habe ich daher in eine eigene Funktionseinheit *SubscriptionStore* ausgelagert.

Für das Feature habe ich eine Platine erstellt, in der die beiden Funktionen *Register* und *Publish* jeweils durch einen Flow abgebildet sind. Aufgrund des gemeinsam genutzten Zustands erscheint mir das naheliegend. Ferner sind nur wenige Funktionseinheiten erforderlich, sodass die Platine überschaubar bleibt. Abbildung 2 zeigt das Modell.

Für die Registrierung eines Subscribers ist eine Funktionseinheit erforderlich, die den Subscriber im *SubscriptionStore* registriert. Der *SubscriptionStore* ist im Wesentlichen ein Dictionary, daher habe ich ihn mit einem gewöhnlichen API realisiert. Die Funktionseinheit *Register_Subscriber* stellt das API des *SubscriptionStore* quasi in den Flow.

Der *Publish*-Flow beginnt mit einem *object* auf dem eingehenden Datenfluss. Das Objekt soll an alle registrierten Subscriber verteilt werden. Dazu müssen zunächst die Subscriber ermittelt werden. Weil hierzu nur der Typ des Objekts erforderlich ist, sorgt der Mapper-Baustein *Map_to_Type* für die Transformation. Aus dem Typ kann die Funktionseinheit *Find_Subscribers* mithilfe des *SubscriptionStore* die Subscriber

Listing 1

Das Bus-API mit Tests entwerfen.

```
[TestFixture]
public class BusTests {
    private Bus bus;
    private MySubscriber mySubscriber;

    [SetUp]
    public void Setup() {
        bus = new Bus();
        mySubscriber = new MySubscriber();
    }
    [Test]
    public void Message_wird_vom_Publisher_zum_Subscriber_übertragen() {
        bus.Register(mySubscriber);
        bus.Publish("Hi");
        Assert.That(mySubscriber.LastReceivedMessage, Is.EqualTo("Hi"));
    }
}
public class MySubscriber : ISubscriberOf<string> {
    public string LastReceivedMessage;
    public void Receive(string message) {
        LastReceivedMessage = message;
    }
}
public interface ISubscriberOf<T> {
    void Receive(T message);
}
```

Listing 2

Eine Spike-Implementation.

```
public class Bus : IBus {
    internal readonly IDictionary<Type, IList<object>> allSubscribers =
    new Dictionary<Type, IList<object>>();
    public void Register<TMessage>(ISubscriberOf<TMessage> subscriber) {
        if (!allSubscribers.ContainsKey(typeof(TMessage))) {
            allSubscribers.Add(typeof(TMessage), new List<object>());
        }
        allSubscribers[typeof(TMessage)].Add(subscriber);
    }
    public void Publish<TMessage>(TMessage message) {
        var subscriberList = allSubscribers[typeof(TMessage)];
        foreach (ISubscriberOf<TMessage> subscriber in subscriberList) {
            subscriber.Receive(message);
        }
    }
}
```

Listing 3

Interface-Design.

```
public interface IBus {
    void Register<TMessage>(ISubscriberOf<TMessage> subscriber);
    void Publish<TMessage>(TMessage message);
}
public interface ISubscriberOf<in T> {
    void Receive(T message);
}
```

ermitteln, die für den Typ registriert sind. *Find_Subscribers* hat daher eine Abhängigkeit zum *SubscriptionStore*.

Sobald die Subscriber ermittelt sind, muss ihnen nur noch die Nachricht zugestellt werden. Dafür ist die Funktionseinheit *Call_Subscribers* zuständig. Da zum Aufrufen der Subscriber zusätzlich die zuzustellende Nachricht benötigt wird, sorgt ein *Join*-Baustein für das Zusammenführen zu einem *Tuple*. Der *Join*-Baustein stammt wieder einmal aus dem Open-Source-Projekt ebclang [1].

Listing 4

BusBoard.ebc.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<board name="BusBoard">
  <using namespace="ebcpatterns.infrastructure" />
  <using namespace="ebcpatterns.flow" />
  <using namespace="messagebus.boards" />
  <wire from="this.Publish" type="object" to="Map_to_type" />
  <wire from="this.Publish" type="object" to="(Join{object,Subscriber*})join.Input1" />
  <wire from="Map_to_type" type="Type" to="Find_subscribers" />
  <wire from="Find_subscribers" type="Subscriber*" to="(Join{object,Subscriber*})join.Input2" />
  <wire from="(Join{object,Subscriber*})join.Output" type="Tuple{object,Subscriber*}"
        to="Call_subscribers" />
  <wire from="this.Subscribe" type="Tuple{Type,Subscriber}" to="Register_subscriber" />
  <dependency from="Find_subscribers" to="SubscriptionStore" />
  <dependency from="Register_subscriber" to="SubscriptionStore" />
</board>
```

Listing 5

Subscriber arbeiten auf dem Typ object.

```
public class Subscriber {
    private Action<object> receive;
    public void SetSubscriber<TMessage>(ISubscriberOf<TMessage> subscriber) {
        receive = x => subscriber.Receive((TMessage)x);
    }
    public void Receive(object message) {
        receive(message);
    }
}
```

Listing 6

Der SubscriptionStore liefert passende Subscriber.

```
public class SubscriptionStore {
    private readonly IDictionary<Type, IList<Subscriber>> subscriptions =
        new Dictionary<Type, IList<Subscriber>>();
    public IEnumerable<Subscriber> FindSubscribers(Type type) {
        IList<Subscriber> subscribers;
        if(!subscriptions.TryGetValue(type, out subscribers)) {
            yield break;
        }
        foreach (var subscriber in subscribers) {
            yield return subscriber;
        }
    }
    public void Register(Type messageType, Subscriber subscriber) {
        if (!subscriptions.ContainsKey(messageType)) {
            subscriptions.Add(messageType, new List<Subscriber>());
        }
        subscriptions[messageType].Add(subscriber);
    }
}
```

.ebc.xml

Nachdem ich das Modell auf Papier skizziert hatte, habe ich es in eine *.ebc.xml*-Datei übersetzt. Auch dabei verwende ich wieder das Tooling aus dem ebclang-Projekt. Die Datei *BusBoard.ebc.xml* wird per Pre-Build-Step in C#-Code übersetzt. Dabei wird der Code für die Platine vollständig erzeugt. Ferner werden für alle Funktionseinheiten Interfaces generiert. Somit musste ich nach dem Erstellen der *.ebc.xml*-Datei lediglich die Bausteine implementieren. Listing 4 zeigt die Datei *BusBoard.ebc.xml*. Neben der Tatsache, dass die Interfaces für alle Bauteile generiert werden, hat die Verwendung der *.ebc.xml*-Datei den Vorteil, dass die Verdrahtung der Platine nun visualisiert werden kann. Abbildung 3 zeigt das visualisierte Modell. Der große Vorzug dieser Darstellung der *.ebc.xml*-Datei ist, dass man so schnell visuell überprüfen kann, ob man das Modell korrekt in die *.ebc.xml*-Datei übertragen hat. Natürlich lässt sich die Verdrahtung der Bausteine auch direkt in C# implementieren. Allerdings ist es dann nicht mehr so einfach, daraus das Modell abzulesen, von einer Visualisierung ganz zu schweigen. Die Implementation der einzelnen Bausteine war damit im Prinzip sehr einfach.

Generics

Die einzige Herausforderung ergab sich, wie erwartet, im Umgang mit generischen Typen. Die Schnittstelle des Busses möchte ich gern mithilfe generischer Typen zur Kompilierzeit überprüfbar machen. Daher ist das Interface *ISubscriberOf<T>* generisch, wobei der Typ *T* den Nachrichtentyp angibt. Das Problem beginnt jedoch, sobald man überlegt, wie man die Subscriber in ein Dictionary einfügt: Der Wertetyp des Dictionarys ist eine Liste. Doch diese Liste kann keine sogenannten *Open Generic Types* aufnehmen, sie muss von einem konkreten Typ sein. Die folgende Definition wäre syntaktisch falsch:

```
IList<ISubscriberOf<>>
```

Versuchte man hier, ebenfalls einen generischen Typparameter zu verwenden, stünde man vor dem Problem, dass das Dictionary am Ende Subscriber zu unterschiedlichen Nachrichtentypen aufnehmen muss. Es gäbe also nicht einen konkreten Typ, mit dem der generische Typ geschlossen werden könnte.

Listing 7

Den SubscriptionStore in den Flow einbinden.

```
public class Register_subscriber : IRegister_subscriber {
    private SubscriptionStore subscriptionStore;
    public void Inject(SubscriptionStore independent) {
        subscriptionStore = independent;
    }
    public void Process(Tuple<Type, Subscriber> message) {
        subscriptionStore.Register(message.Item1, message.Item2);
    }
}
```

Aus diesem Grund, und auch um die generischen Typen aus dem Modell herauszuhalten, habe ich die Klasse *Subscriber* eingeführt. Im Flow spielt es keine Rolle, auf welchen Nachrichtentyp sich ein Subscriber konkret registriert hat. Die Klasse *Subscriber* ist daher nicht generisch, sondern arbeitet mit Nachrichten vom Typ *object*. An den notwendigen Stellen werden die Nachrichten dann einfach auf den passenden Typ gecastet. Dass der Cast funktioniert, ist dadurch sichergestellt, dass der *SubscriptionStore* nur passende Subscriber liefert. Listing 5 zeigt die Implementation der Klasse *Subscriber*.

Die Klasse enthält eine *Action<object>*, die in der *Receive*-Methode aufgerufen wird, um eine Nachricht an den Subscriber zuzustellen. Damit die Nachricht auf den passenden Typ gecastet werden kann, verwende ich zum Setzen des Subscribers eine Methode statt des Konstruktors, denn eine Methode kann einen generischen Typparameter tragen. Ich hatte zuerst versucht, das Zuweisen der Action im Konstruktor vorzunehmen. Da dieser jedoch keinen generischen Typparameter erhalten kann, schied dies als Lösung aus. Ferner wollte ich, wie bereits beschrieben, die Klasse *Subscriber* selbst nicht generisch machen, weil sie im Modell verwendet wird und weil dies das Problem nur verlagert hätte.

Die Methode *SetSubscriber* erhält einen Subscriber vom Typ *ISubscriberOf<TMessage>* und erstellt dann eine Action, in der die Nachricht von *object* nach *TMessage* gecastet wird. Dadurch kann die Action in der *Receive*-Methode einfach aufgerufen werden. Der *SubscriptionStore* ist dafür verantwortlich, nur solche Subscriber zu liefern, die zum Nachrichtentyp passen, siehe Listing 6.

Beim Registrieren eines Subscribers wird geprüft, ob der zu registrierende Nachrichtentyp schon im Dictionary enthalten ist. Dies ist der Fall, wenn zuvor schon ein anderer Subscriber auf den Nachrichtentyp registriert wurde. Beim ersten Registrieren eines Subscribers für den Typ wird ein neuer Eintrag im Dictionary erzeugt und mit einer leeren Liste initialisiert. Anschließend wird der Subscriber in die Liste eingefügt.

Beim Ermitteln der Subscriber muss zunächst geprüft werden, ob zu dem Nachrichtentyp überhaupt ein Eintrag existiert. Ist dies nicht der Fall, wird durch *yield break* eine leere Aufzählung geliefert. Anderenfalls wird über die Liste der Subscriber iteriert und jeder mit *yield return* an den Aufrufer übergeben.

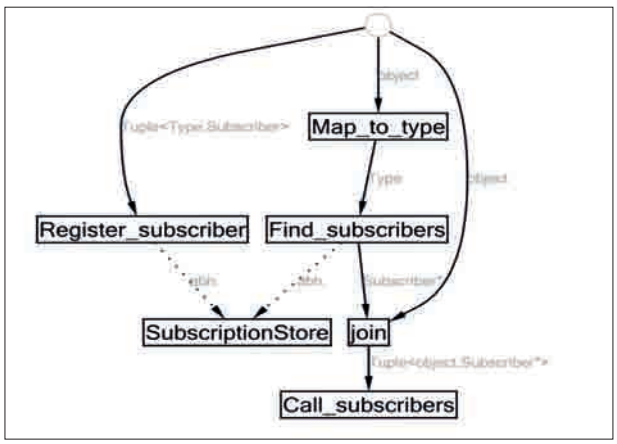
Injection von Abhängigkeiten

Die Funktionseinheit *Register_Subscriber* dient dazu, den *SubscriptionStore* in den Flow einzubinden. Sie hat daher nach außen ein Flow-Interface und übersetzt dieses nach innen in Aufrufe eines normalen APIs. Die Abhängigkeit zum *SubscriptionStore* wird über die *Inject*-Methode in die Funktionseinheit hineingereicht, siehe Listing 7.

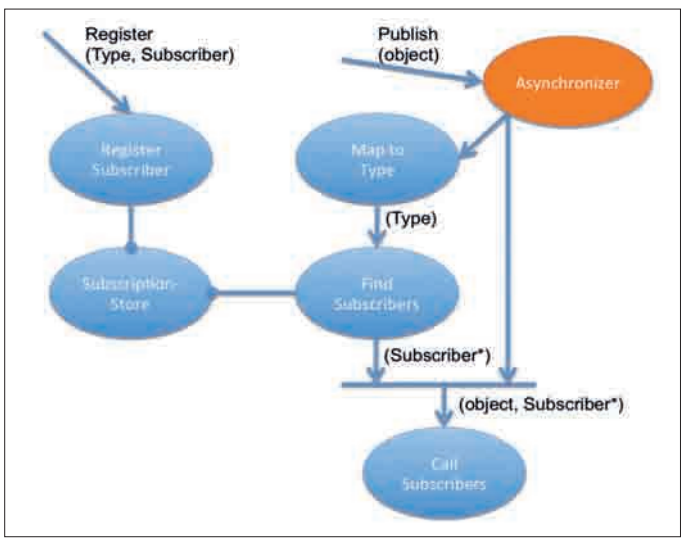
Dass *Register_Subscriber* die *Inject*-Methode implementieren muss, liegt daran, dass in *BusBoard.ebc.xml* mit dem *dependency*-Element eine Abhängigkeit definiert ist. Die folgende Zeile führt dazu, dass das generierte Interface *IRegister_subscriber* von *IDependsOn<SubscriptionStore>* erbt:

```
<dependency from="Register_subscriber"
to="SubscriptionStore" />
```

Dieses Interface ist ebenfalls im ebclang-Projekt definiert. Der Mapper *Map_to_type* ist trivial, siehe Listing 8.



[Abb. 3] Das Modell aus Sicht der .ebc.xml-Datei.



[Abb. 4] Message-Bus mit asynchroner Verarbeitung.

Listing 8

Typ bestimmen.

```
public class Map_to_type : IMap_to_type
{
    public event Action<Type> Result;
    public void Process(object message) {
        Result(message.GetType());
    }
}
```

In der Methode *Process* löse ich den *Result*-Event aus, ohne vorher zu prüfen, ob er möglicherweise *null* ist. Dies ist nicht etwa Schludrigkeit, sondern pure Absicht. Der *Result*-Event muss in der Platine verdrahtet werden. Anderenfalls ist etwas faul, denn es ergibt keinen Sinn, die *Process*-Methode aufzurufen, ohne das Ergebnis im *Result*-Event abzuholen. Insofern ist es gut, wenn die Funktionseinheit bei fehlerhafter Verdrahtung „Bummsdi“ macht.

Nach dem Ermitteln des Typs steht im Flow die Funktionseinheit *Find_subscribers*. Diese sieht *Register_subscriber* zum Verwechseln ähnlich. Das liegt daran, dass auch sie nur die Flow-Schnittstelle zum *SubscriptionStore* darstellt, siehe Listing 9. Die Abhängigkeit ist ebenfalls in der Datei *BusBoard.ebc.xml* ausgedrückt, sodass auch hier eine *Inject*-Methode zu implementieren ist. Zuletzt kommt, nach dem *Join*, noch das Aufrufen der Subscriber an die Reihe. Dafür ist *Call_subscribers* zuständig, siehe Listing 10.

Abschließend müssen alle Funktionseinheiten samt Platine instanziert werden. Dies geschieht im Bus. Da der Bus selbst keine Logik enthält, sondern lediglich ein Wrapper um die Platine ist, halte ich es für legitim, dass der Bus die Platine selbst instanziert, siehe Listing 11. Eine Konstruktorinjektion wäre natürlich möglich, doch ich teste den Bus ohnehin in mehreren Integrationstests. Insofern wäre durch die Injektion nichts gewonnen worden.

Asynchrone Verarbeitung

Die Implementation des Message-Busses ist damit, bis auf das Gefummel mit den generischen Typen, leicht von der Hand gegangen. Daher wage ich mich noch etwas weiter vor. Schließlich möchte ich auch wissen, ob das Modell tatsächlich gut evolvierbar ist. Die nächste Anforderung lautet daher: Die Nachrichten sollen asynchron verarbeitet werden.

Listing 9

Subscriber finden.

```
public class Find_subscribers : IFind_subscribers
{
    private SubscriptionStore subscriptionStore;
    public void Inject(SubscriptionStore independent) {
        subscriptionStore = independent;
    }
    public event Action<IEnumerable<Subscriber>> Result;
    public void Process(Type type) {
        Result(subscriptionStore.FindSubscribers(type));
    }
}
```

Listing 10

Subscriber aufrufen.

```
public class Call_subscribers : ICall_subscribers
{
    public void Process(Tuple<object, IEnumerable<Subscriber>> message) {
        foreach (var subscriber in message.Item2) {
            subscriber.Receive(message.Item1);
        }
    }
}
```

Listing 11

Alle Funktionseinheiten instanzieren.

```
public class Bus : IBus
{
    private BusBoard busBoard;
    public Bus() {
        var find_subscribers = new Find_subscribers();
        var register_subscriber = new Register_subscriber();

        busBoard = new BusBoard(
            new Map_to_type(),
            find_subscribers,
            new Join<object, IEnumerable<Subscriber>>(),
            new Call_subscribers(),
            register_subscriber);

        var subscriptionStore = new SubscriptionStore();
        find_subscribers.Inject(subscriptionStore);
        register_subscriber.Inject(subscriptionStore);
    }
    public void Register<TMessage>(ISubscriberOf<TMessage> subscriber) {
        var sub = new Subscriber();
        sub.SetSubscriber(subscriber);
        busBoard.Subscribe(new Tuple<Type, Subscriber>(typeof(TMessage), sub));
    }
    public void Publish<TMessage>(TMessage message) {
        busBoard.Publish(message);
    }
}
```

Das bedeutet, der *Publish*-Aufruf soll sofort zum Aufrufer zurückkehren und die Nachricht soll auf einem neuen Thread an die Subscriber zugestellt werden. Ob Flow-

Design die Evolvierbarkeit der Modelle gut unterstützt, wird sich nun also wieder einmal zeigen müssen.

Abbildung 4 zeigt das veränderte Modell. Es ist lediglich ein Baustein hinzugekommen, der *Asynchronizer*. Bei ihm handelt es sich wieder um einen Standardbaustein aus dem ebclang-Projekt. Seine Aufgabe besteht darin, den eingehenden Datenfluss unverändert am Ausgang wieder auszugeben.

Allerdings erfolgt die Ausgabe am Ausgang auf einem neuen Thread. Dadurch kehrt die *Process*-Methode des *Asynchronizers* wieder zum Aufrufer zurück, bevor die Daten am *Result*-Ausgang anliegen.

Für die asynchrone Arbeitsweise des Busses waren nur zwei Änderungen erforderlich:

- In der Datei *BusBoard.ebc.xml* musste der *Asynchronizer* ergänzt werden.
- Beim Instanzieren der Platine musste eine *Asynchronizer*-Instanz in die Platine hineingereicht werden.

Allerdings sind nun einige Tests kaputt gegangen. Das liegt daran, dass der Test bereits weiterläuft, während der Bus noch damit beschäftigt ist, die Nachrichtenzustellung auf einem anderen Thread abzuarbeiten. So kann es passieren, dass zu dem Zeitpunkt, an dem der Testrunner die Assertions ausführt, das Ergebnis noch nicht eingetroffen ist.

Eine schnelle, aber auch suboptimale Lösung besteht darin, hinter die *Publish*-Aufrufe ein *Thread.Sleep* einzufügen. Dadurch wird der Test-Thread kurzzeitig angehalten, sodass der Bus-Thread aufholen kann. Die notwendige Dauer des *Thread.Sleep* ist allerdings nur schwer einzuschätzen. Auf einem stark ausgelasteten Continuous-Integration-Server müssen die Verzögerungen sicher länger ausfallen als auf einer Entwicklermaschine.

Das Problem des Multithreadings in den Tests wird eleganter gelöst, indem man einen *AutoResetEvent* verwendet, siehe Listing 12. Der *AutoResetEvent* wird im Test-Setup instanziert und in die Subscriber-Attrappe *MySubscriber* hineingereicht. Erhält der Subscriber die Nachricht, setzt er den Event mit *autoResetEvent.Set()*. Im Test wird nach dem Publish-Aufruf der Thread angehalten, indem auf den *AutoResetEvent* gewartet wird. Die Zeile

```
Assert.That(autoResetEvent1.WaitOne(500),
            Is.True);
```

sorgt dafür, dass der laufende Thread maximal 500 ms angehalten wird. Tritt der Event in dieser Zeit ein, liefert die *WaitOne*-Methode *true* zurück. In dem Fall ist aus Sicht unseres Tests alles in Ordnung. Wird der Timeout erreicht, gibt *WaitOne false* zurück und der Test schlägt fehl. Der große Vorteil gegenüber einem *Thread.Sleep* liegt nun darin, dass der Test-Thread nur genau so lange angehalten wird, wie es für die korrekte Ausführung der parallelen Verarbeitungsschritte nötig ist. Dennoch sorgt der Timeout dafür, dass der Test nicht ewig läuft.

Listing 12

Tests für Multithreading absichern.

```
[TestFixture]
public class BusTests
{
    private Bus bus;
    private MySubscriber mySubscriber1;
    private AutoResetEvent autoResetEvent1;

    [SetUp]
    public void Setup() {
        bus = new Bus();
        autoResetEvent1 = new AutoResetEvent(false);
        mySubscriber1 = new MySubscriber(autoResetEvent1);
    }

    [Test]
    public void Message_wird_vom_Publisher_zum_Subscriber_übertragen() {
        bus.Register(mySubscriber1);
        bus.Publish("Hi");

        Assert.That(autoResetEvent1.WaitOne(500), Is.True);

        Assert.That(mySubscriber1.LastReceivedMessage, Is.EqualTo("Hi"));
    }
}

public class MySubscriber : ISubscriberOf<string>
{
    private readonly AutoResetEvent autoResetEvent;
    public string LastReceivedMessage;
    public int NumberOfReceivedMessages;

    public MySubscriber(AutoResetEvent autoResetEvent) {
        this.autoResetEvent = autoResetEvent;
    }

    public void Receive(string message) {
        NumberOfReceivedMessages++;
        LastReceivedMessage = message;
        autoResetEvent.Set();
    }
}
```

Fazit

Die Modellierung des Message-Busses mit Flow-Design hat gut funktioniert. Im direkten Vergleich mit der Spike-Implementation mag die Implementation mittels Event-Based Components anfangs vielleicht etwas „over-engineered“ gewirkt haben. Spätestens beim Ergänzen der Asynchronizität dürfte der Vorteil aber auf der Hand liegen: Flow-Designs und Umsetzungen mit Event-Based Components evolvieren sehr gut.

In der Praxis ist es bei Softwareprojekten so, dass die Probleme nicht am Anfang der Entwicklung auftreten, sondern erst später, wenn bereits einiges an Code vorhanden ist. Insofern tut man gut daran, sich auf diese Phase vorzubereiten und Modell sowie Implementation so zu wählen, dass Änderungen und Ergänzungen leicht möglich sind. Dafür mag der Aufwand anfangs manchmal etwas höher erscheinen, doch zahlt sich diese Vorsorge schnell aus. **[ml]**

[1] Event-Based Components Tooling, http://ebclang.codeplex.com