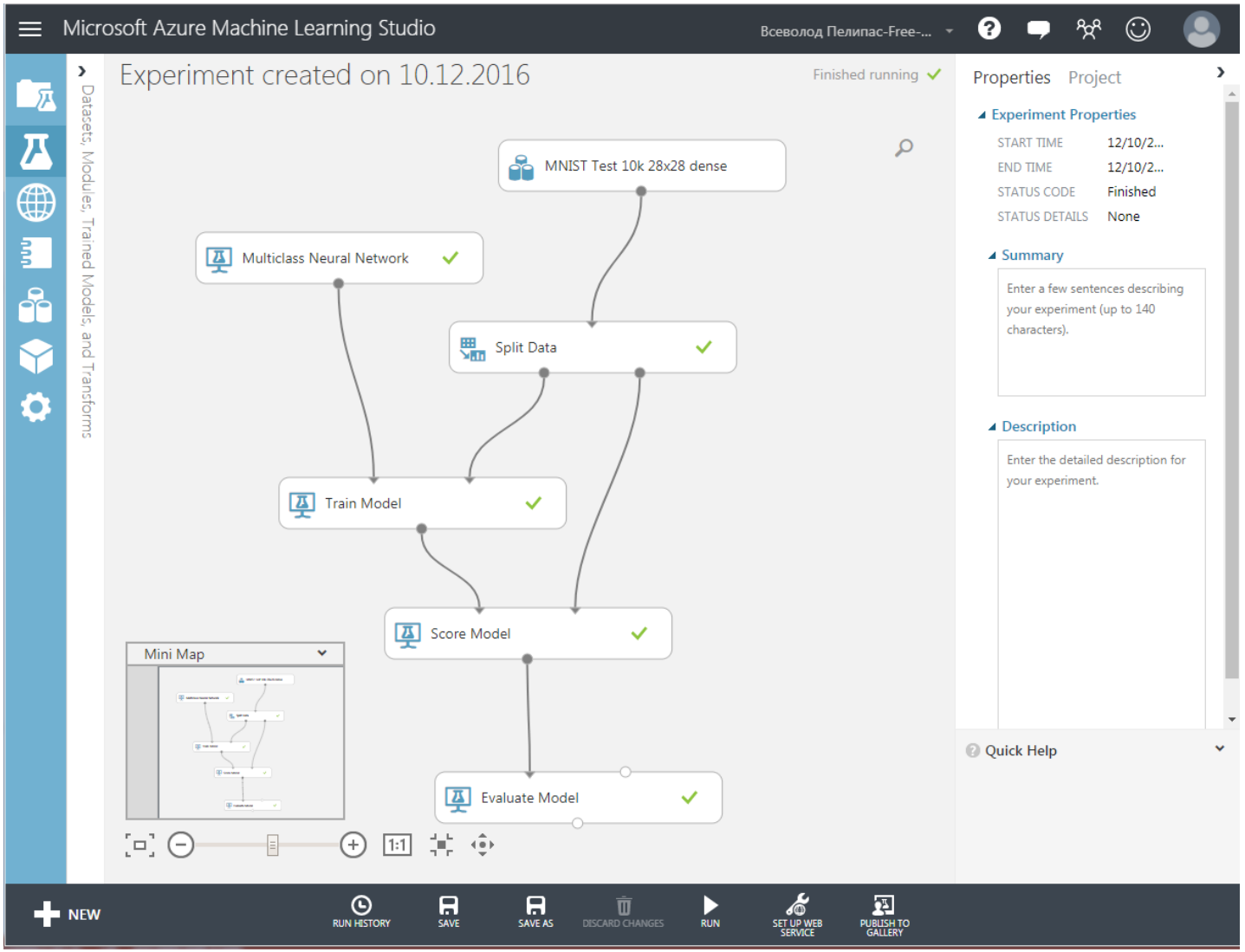
Парадигмы программирования

Реактивное программирование на примере Rx и его родословная

Программирование потоков данных

- ППД - подход к программированию, при котором программа моделируется в виде ориентированного графа потока данных между операциями *(подобного диаграмме потока данных)*.
- Структура:
  - Узлы, порты, дуги, токены, активация.
- Примеры:
  - Системы моделирования (Simulink, LabView)
  - Unix Pipes

ППД - Пример (Azure ML Studio)



ППД – не все так просто. Детали

- Push/pull дуги
- Join/split на дугах
- Mutable/immutable данные
- Stateful/stateless узлы
- Однотокенные/многотокенные дуги
- Sync/async активация
- Множественные порты
- Составные и рекурсивные узлы
- Обратные связи (циклы)

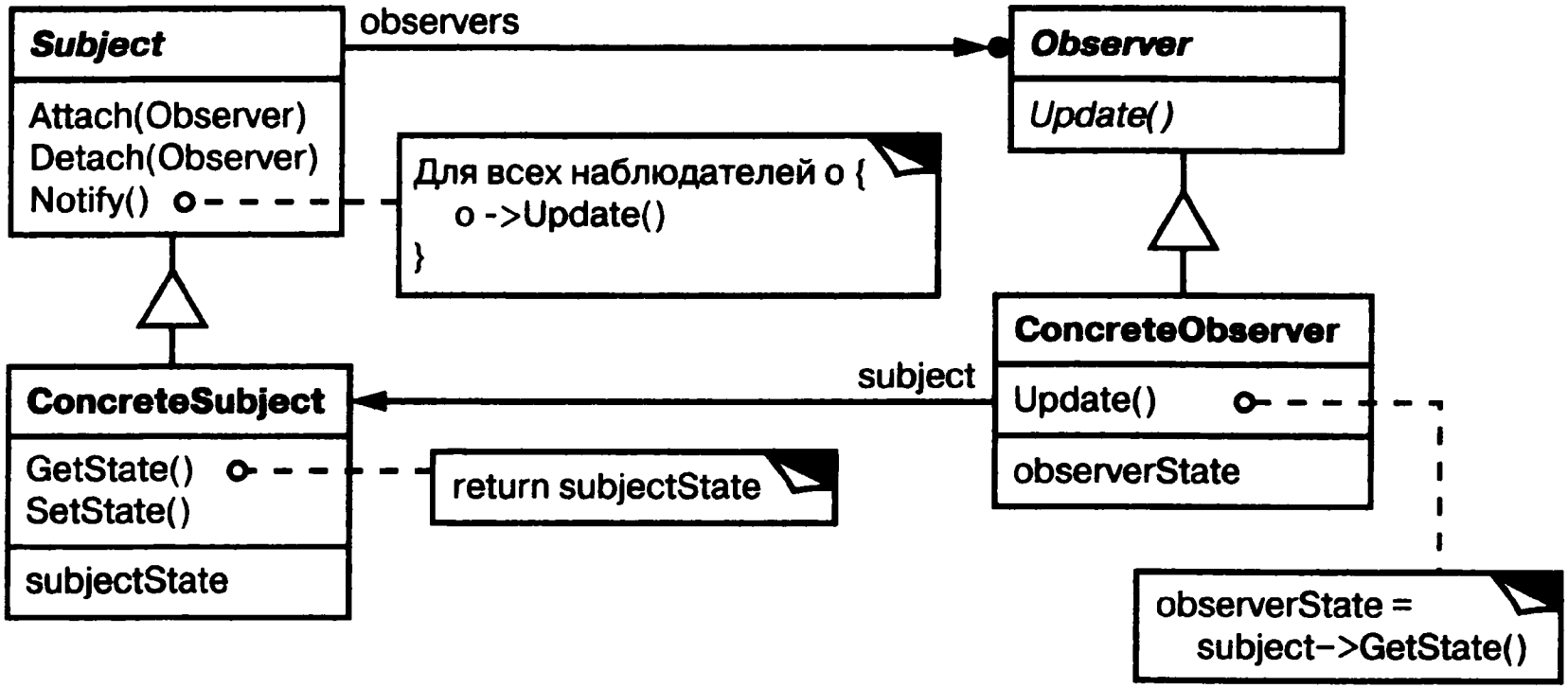
Unix Pipes – простейшая реализация ППД

- Концепция крайне проста:
  - На каждом узле 3 порта:
    - Входной stdin
    - Выходные stdout, stderr
  - Всегда stdin -> stdout
- Крайне полезная штука
  - cat example.log | grep Critical | awk {…достаем нужные данные}
  - ps aux | grep myProcess| grep -v grep | awk '{print $2}' | xargs kill

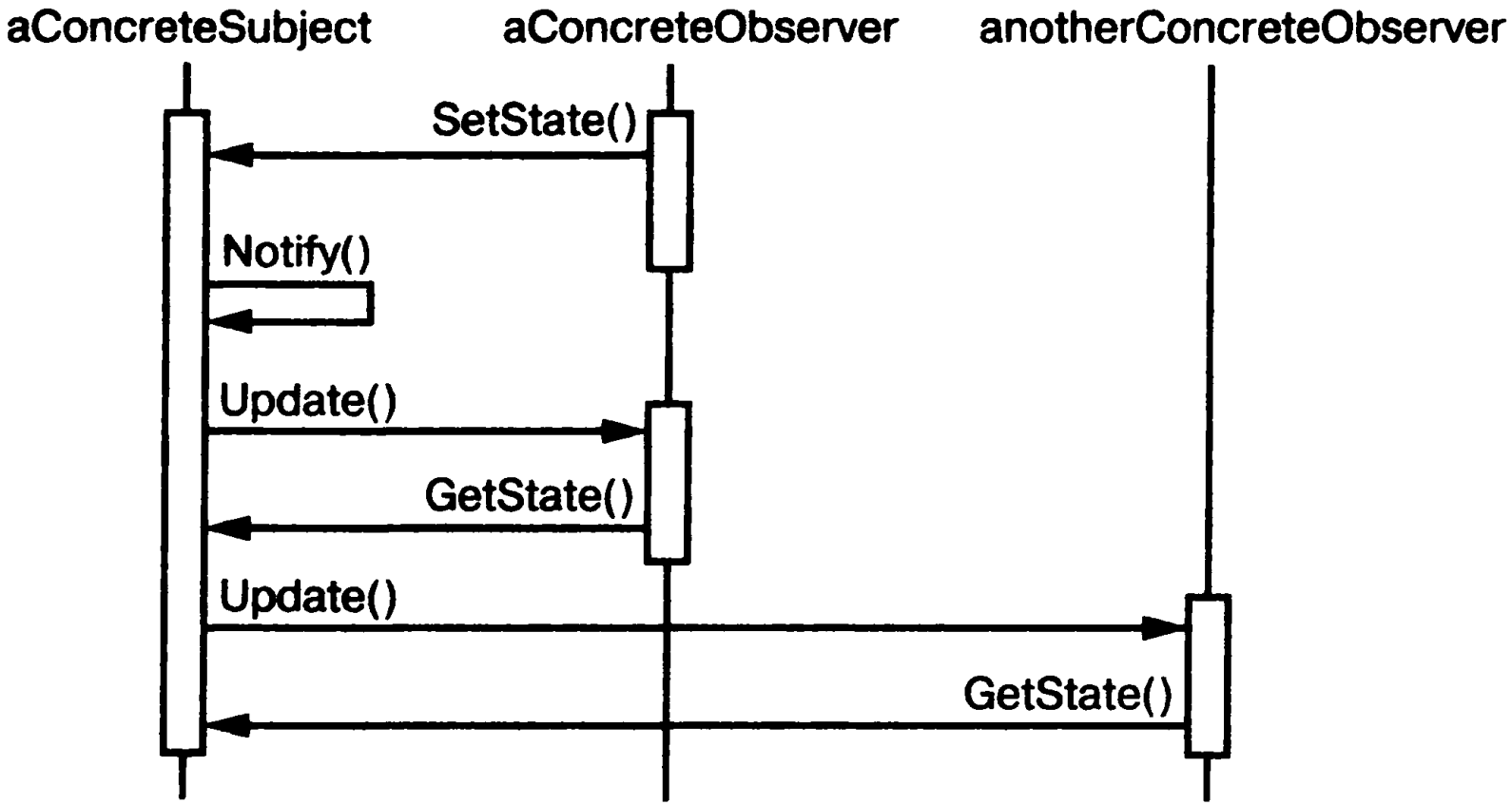
Событийно-ориентированное программирование

- СОП - парадигма программирования, в которой выполнение программы описывается как реакция на события
- События – любые:
  - действия пользователя (клавиатура, мышь);
  - сообщения других программ и потоков;
  - события операционной системы (например, поступление сетевого пакета).
- Основные преимущества
  - Слабая связность компонентов
  - Удобство распараллеливания вычислений

Шаблон «Наблюдатель»



Шаблон «Наблюдатель»



Применение СОП

- UI
  - Objective-C – полностью событийная модель взаимодействия объектов (модель Smalltalk)
    - [myObject doSomething: myParameter]
  - C#, JS, etc.
- Неблокирующие сервера (проблема 10 000 соединений)
  - Node.js, Twisted
  - Свои собственные реализации на уровне сокетов:
    - Input-Output Completion Ports (Win, Solaris) или select/poll/epoll/kqueue (*nix)
    - Автоматная модель

Реактивное программирование

- Реактивная парадигма
  - программа не действует сама по себе, она реагирует на изменение внешнего мира.
- Event Driven + Data Flow + LINQ = Rx
  - набор библиотек, которые позволяют работать с событиями и асинхронными вызовами в декларативном стиле (через Linq).
  - Многоплатформенность и многоязычность

Реактивные приложения

- Реагируют на события
  - Слабая связность компонентов
    - Отправитель и получатель могут быть реализованы, не оглядываясь на детали того, как события распространяются в системе
  - Удобство распараллеливания вычислений
    - Неблокирующее асинхронное взаимодействие позволяет эффективнее использовать ресурсы
- Реагируют на повышение нагрузки
  - Фокус на масштабируемость, конкурентный доступ к общедоступным ресурсам сводится к минимуму
- Реагируют на сбои
  - Строятся отказоустойчивые системы с возможностью восстанавливаться на всех уровнях
- Реагируют на пользователей
  - Гарантированное время отклика, не зависящее от нагрузки

Основные архитектурные идеи

- Push-модель взаимодействия
  - данные отправляются к своим потребителям, когда становятся доступными, вместо того чтобы впустую тратить ресурсы, постоянно запрашивая или ожидая данные.
- Неблокирующая асинхронная передача сообщений
  - Поток отправителя не блокируется в ожидании обработки сообщения получателем
- Минимизация общедоступного изменяемого состояния
  - Позволяет избегать конкурентного доступа и синхронизации
- Соблюдение этих принципов на всех слоях приложения
  - «Приложение должно быть реактивным сверху донизу», иначе масштабирование упрётся в слабое звено.

Масштабируемость, отказоустойчивость и отзывчивость

- Масштабируемое приложение - способное легко расшириться или модернизироваться
  - Событийно-ориентированная система, базирующаяся на асинхронной передаче сообщений, является основой масштабируемости.
  - Необходима единая среда передачи сообщений между узлами.
  - Serverless computing
- Отказоустойчивость является частью архитектуры
  - Слабая связность изолирует отказы в отдельных модулях
  - Отказ – такое же сообщение, и должен иметь свой обработчик
- Отзывчивое приложение – отвечающее быстро или реагирующее надлежащим образом.
  - Наблюдаемые модели позволяют другим системам получать события, когда их состояние изменяется.
  - Потоки событий позволяют избегать блокировок и позволяет преобразованиям и коммуникациям быть асинхронными и неблокирующими.

Основы Rx

- Push-коллекции – ключевой элемент.
  - Pull-коллекции (обычные) – MoveNext вызывается извне;
  - Push-коллекции – сами отдают полученные данные своим подписчикам;
    - Собственно коллекции (точнее, их итераторы)
    - События
    - Результаты асинхронных вызовов
- Концептуально закрывает «дыру» в таблице технологий получения данных:

	Одиночный элемент	Множество элементов
Синхронно	T getData()	IEnumerable<T> getData()
Асинхронно	Task<T> getData()	**Observable<T> getData()**

Аналогия с IEnumerable

```
//Синхронное получение набора данных
var list = new List<int>{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15};
var i = list.Skip(10)
            .Take(5)
            .Select(x => x + " transformed");
foreach (var item in i) Console.WriteLine(item);

//Асинхронное получение набора данных
var o = Observable.Interval(TimeSpan.FromSeconds(1))
                  .Skip(10)
                  .Take(5)
                  .Select(x => x + " transformed");
o.Subscribe(x => Console.WriteLine(x));

Console.ReadLine();
```

Rx: Интерфейсы

- IObservable<T> – наблюдаемый (аналог IEnumerable для pull-коллекций)
  - IDisposable Subscribe(IObserver<T> observer);
- IObserver<T> – наблюдатель
  - void OnNext(T value); // GoF Observer
  - void OnError(Exception error);
  - void OnCompleted();
- Классы-помощники Observable и Observer
  - НЕ реализуют интерфейсы, а содержат набор статических методов, облегчающих работу с ними.

Rx: пример с итератором

```
IObservable<int> o= Observable.Range(1, 5);
//(new List<int> { 1, 2, 3, 4, 5 }).ToObservable()

 var subscription  = o.Subscribe(
                x => Console.WriteLine(x),
                ex => Console.WriteLine("Oops!"),
                () => Console.WriteLine("Finished!")
                );


Console.ReadLine();
subscription.Dispose();
```

«Холодные» и «горячие» наблюдаемые объекты

- «Холодные» наблюдаемые:
  - Начинают выдавать данные только после подписки;
  - Повторяют выдаваемую последовательность для всех подписавшихся (выдают одну и ту же последовательность всем)
  - Пример – некие искусственные генераторы для мат. вычислений («для всех целых чисел от X до Y»)
    - Observable.Range(X, Y);
- «Горячие» (connectable) наблюдаемые:
  - Выдают данные постоянно, независимо от состояния подписки
  - Подписавшиеся получают части одной и той же последовательности, начиная с момента подписки.
  - Пример – потоки данных реального мира (например события от мышки, новостной поток, и т.п.)
    - Observable.Publish

Rx: работа с событиями .NET

- Observable.FromEventPattern позволяет создать экземпляр IObservable на базе объекта/класса и имени события.
  - Есть и другие полезные методы для работы с событиями, но пока не о них;
- Это «горячий» наблюдаемый объект
- Работа с событиями как с данными:
  - Фильтрация событий;
  - Композиция событий;
  - Манипуляция событиями (1st class citizenship):
    - Передача в параметрах и возвращение и функций;
    - Сохранение для последующего использования;
    - И т.д.;
  - Простой механизм отписывания от события через IDisposable (“using”)
- Управление временем обработки (Scheduling)
  - Асинхронная от источника обработка событий

Rx: Пример с координатами указателя мыши

```
var o = Observable.FromEventPattern(this, "MouseClick")

//опционально
.Where(x => ((MouseEventArgs)x.EventArgs).Button ==
MouseButtons.Right);


o.Subscribe(
    x => listBox1.Items.Add(
    $"Right click at
    X={((MouseEventArgs)x.EventArgs).X},
Y={((MouseEventArgs)x.EventArgs).Y}"
                    )
);
```

Rx: работа с асинхронными источниками данных

- Асинхронные источники данных = результаты работы асинхронных методов
  - «Не знаем точно когда, но когда-то данные будут готовы и с ними надо будет что-то делать»
- Поддерживаются два паттерна асинхронного программирования .NET
  - APM (Asynchronous Programming Model)
    - BeginX/EndX методы, AsyncCallback, IAsyncResult
    - Observable.FromAsyncPattern(BeginX, EndX)
  - TAP (Task-based Asynchronous Pattern)
    - Метод возвращает Task<TResult> вместо TResult
    - Observable.FromAsync(…) или Task.ToObservable (…)

Rx: примеры с асинхронными источниками данных

```
var o = Observable.FromAsync( () => Task.Run( () =>
                {
                    Thread.Sleep(5000);
                    return 12345;
                } ) );

            o.Subscribe(
                x => Console.WriteLine(x),
                ex => Console.WriteLine("Oops!"),
                () => Console.WriteLine("Finished!")
                );

            Console.ReadLine();
```

Rx: дополнительные инструменты LINQ

- Создание последовательностей
  - Generate, Defer, Range, Interval, Empty, Error, Never, etc.
- Объединение последовательностей
  - Concat последовательно
  - Merge параллельно
  - Catch при ошибке первой идет вторая («план Б»)
- Проекция
  - Select – обычная проекция
  - SelectMany – «сплющивает» результаты, когда возвращается IObservable, возвращающий другие IObservable (FlatMap).
- Фильтрация
  - Where
- И еще множество – функциональные, математические, временные, и т.п.

Тема (Subject)

- Subject<T> реализует IObservable<T> и IObserver<T> одновременно;
  - Одновременно и приемник, и передатчик
- Используется как прокси между группами подписчиков и источниками.
  - Собрать группу подписчиков и подписать их всех одновременно;
  - Дополнительная логика управления потоком, например, буферизация и временной сдвиг;
- «Кирпичик» для построения потока данных

Пример Subject

```
var o = Observable.Interval(TimeSpan.FromSeconds(1)).Take(5);

        var subj = new Subject<long>();

        var sub1 = subj.Subscribe(
            x => Console.WriteLine($"1st: {x}") );

        var sub2 = subj.Subscribe(
            x => Console.WriteLine($"2nd: {x}") );

        o.Subscribe(subj);

        Console.ReadLine();
```

Планировщик (Scheduler)

- Планировщик управляет публикацией нотификаций наблюдателям
  - Содержит внутри себя запланированные к исполнению задачи
  - Определяет контекст исполнения задач (тредпул, текущий поток, другой аппдомен)
  - Имеет внутренние часы (могут отличаться от реального времени)
- Встроенные
  - ImmediateScheduler
  - VirtualScheduler
  - CurrentThreadScheduler
  - NewThreadScheduler
  - TaskPoolScheduler
- Устанавливается с помощью оператора ObserveOn