```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class CloseTab
    {
        public Guid Id;
        public decimal AmountPaid;
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class TabNotOpen : Exception
    {
    }

    public class DrinksNotOutstanding : Exception
    {
    }

    public class FoodNotOutstanding : Exception
    {
    }

    public class FoodNotPrepared : Exception
    {
    }

    public class MustPayEnough : Exception
    {
    }

    public class TabHasUnservedItems : Exception
    {
    }
}
```

Nov 29, 14 21:15 **MarkDrinksServed.cs** Page 1/1

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class MarkDrinksServed
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```

Nov 29, 14 21:15 **MarkFoodPrepared.cs** Page 1/1

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class MarkFoodPrepared
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class MarkFoodServed
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cafe.Tab
{
    public class OpenTab
    {
        public Guid Id;
        public int TableNumber { get; set; }
        public string Waiter { get; set; }
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Events.Cafe;

namespace Cafe.Tab
{
    public class PlaceOrder
    {
        public Guid Id;
        public List<OrderedItem> Items;
    }
}
```



```
ï»¿using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Edument.CQRS;
using Events.Cafe;

namespace Cafe.Tab
{
    public class TabAggregate : Aggregate,
        IHandleCommand<OpenTab>,
        IHandleCommand<PlaceOrder>,
        IHandleCommand<MarkDrinksServed>,
        IHandleCommand<MarkFoodPrepared>,
        IHandleCommand<MarkFoodServed>,
        IHandleCommand<CloseTab>,
        IApplyEvent<TabOpened>,
        IApplyEvent<DrinksOrdered>,
        IApplyEvent<FoodOrdered>,
        IApplyEvent<DrinksServed>,
        IApplyEvent<FoodPrepared>,
        IApplyEvent<FoodServed>,
        IApplyEvent<TabClosed>
    {
        private List<OrderedItem> outstandingDrinks = new List<OrderedItem>();
        private List<OrderedItem> outstandingFood = new List<OrderedItem>();
        private List<OrderedItem> preparedFood = new List<OrderedItem>();
        private bool open;
        private decimal servedItemsValue;

        public IEnumerable Handle(OpenTab c)
        {
            yield return new TabOpened
            {
                Id = c.Id,
                TableNumber = c.TableNumber,
                Waiter = c.Waiter
            };
        }

        public IEnumerable Handle(PlaceOrder c)
        {
            if (!open)
                throw new TabNotOpen();

            var drink = c.Items.Where(i => i.IsDrink).ToList();
            if (drink.Any())
                yield return new DrinksOrdered
                {
                    Id = c.Id,
                    Items = drink
                };

            var food = c.Items.Where(i => !i.IsDrink).ToList();
            if (food.Any())
                yield return new FoodOrdered
                {
                    Id = c.Id,
                    Items = food
                };
        }

        public IEnumerable Handle(MarkDrinksServed c)
        {
            if (!AreDrinksOutstanding(c.MenuNumbers))
                throw new DrinksNotOutstanding();

            yield return new DrinksServed
```

```
            {
                Id = c.Id,
                MenuNumbers = c.MenuNumbers
            };
        }

        public IEnumerable Handle(MarkFoodPrepared c)
        {
            if (!IsFoodOutstanding(c.MenuNumbers))
                throw new FoodNotOutstanding();

            yield return new FoodPrepared
            {
                Id = c.Id,
                MenuNumbers = c.MenuNumbers
            };
        }

        public IEnumerable Handle(MarkFoodServed c)
        {
            if (!IsFoodPrepared(c.MenuNumbers))
                throw new FoodNotPrepared();

            yield return new FoodServed
            {
                Id = c.Id,
                MenuNumbers = c.MenuNumbers
            };
        }

        public IEnumerable Handle(CloseTab c)
        {
            if (!open)
                throw new TabNotOpen();
            if (HasUnservedItems())
                throw new TabHasUnservedItems();
            if (c.AmountPaid < servedItemsValue)
                throw new MustPayEnough();

            yield return new TabClosed
            {
                Id = c.Id,
                AmountPaid = c.AmountPaid,
                OrderValue = servedItemsValue,
                TipValue = c.AmountPaid - servedItemsValue
            };
        }

        private bool AreDrinksOutstanding(List<int> menuNumbers)
        {
            return AreAllInList(want: menuNumbers, have: outstandingDrinks);
        }

        private bool IsFoodOutstanding(List<int> menuNumbers)
        {
            return AreAllInList(want: menuNumbers, have: outstandingFood);
        }

        private bool IsFoodPrepared(List<int> menuNumbers)
        {
            return AreAllInList(want: menuNumbers, have: preparedFood);
        }

        private static bool AreAllInList(List<int> want, List<OrderedItem> have)
        {
            var curHave = new List<int>(have.Select(i => i.MenuNumber));
            foreach (var num in want)
                if (curHave.Contains(num))
                    curHave.Remove(num);
```



```
                else
                    return false;
            return true;
        }

        public bool HasUnservedItems()
        {
            return outstandingDrinks.Any() || outstandingFood.Any() || preparedF
ood.Any();
        }

        public void Apply(TabOpened e)
        {
            open = true;
        }

        public void Apply(DrinksOrdered e)
        {
            outstandingDrinks.AddRange(e.Items);
        }

        public void Apply(FoodOrdered e)
        {
            outstandingFood.AddRange(e.Items);
        }

        public void Apply(DrinksServed e)
        {
            foreach (var num in e.MenuNumbers)
            {
                var item = outstandingDrinks.First(d => d.MenuNumber == num);
                outstandingDrinks.Remove(item);
                servedItemsValue += item.Price;
            }
        }

        public void Apply(FoodPrepared e)
        {
            foreach (var num in e.MenuNumbers)
            {
                var item = outstandingFood.First(f => f.MenuNumber == num);
                outstandingFood.Remove(item);
                preparedFood.Add(item);
            }
        }

        public void Apply(FoodServed e)
        {
            foreach (var num in e.MenuNumbers)
            {
                var item = preparedFood.First(f => f.MenuNumber == num);
                preparedFood.Remove(item);
                servedItemsValue += item.Price;
            }
        }

        public void Apply(TabClosed e)
        {
            open = false;
        }
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Events.Cafe;
using Edument.CQRS;

namespace CafeReadModels
{
    public class ChefTodoList : IChefTodoListQueries,
        ISubscribeTo<FoodOrdered>,
        ISubscribeTo<FoodPrepared>
    {
        public class TodoListItem
        {
            public int MenuNumber;
            public string Description;
        }

        public class TodoListGroup
        {
            public Guid Tab;
            public List<TodoListItem> Items;
        }

        private List<TodoListGroup> todoList = new List<TodoListGroup>();

        public List<TodoListGroup> GetTodoList()
        {
            lock (todoList)
                return (from grp in todoList
                        select new TodoListGroup
                        {
                            Tab = grp.Tab,
                            Items = new List<TodoListItem>(grp.Items)
                        }).ToList();
        }

        public void Handle(FoodOrdered e)
        {
            var group = new TodoListGroup
            {
                Tab = e.Id,
                Items = new List<TodoListItem>(
                    e.Items.Select(i => new TodoListItem
                    {
                        MenuNumber = i.MenuNumber,
                        Description = i.Description
                    }))
            };

            lock (todoList)
                todoList.Add(group);
        }

        public void Handle(FoodPrepared e)
        {
            lock (todoList)
            {
                var group = todoList.First(g => g.Tab == e.Id);

                foreach (var num in e.MenuNumbers)
                    group.Items.Remove(
                        group.Items.First(i => i.MenuNumber == num));

                if (group.Items.Count == 0)
                    todoList.Remove(group);
            }
        }
```



```
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;

namespace CafeReadModels
{
    public interface IChefTodoListQueries
    {
        List<ChefTodoList.TodoListGroup> GetTodoList();
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;

namespace CafeReadModels
{
    public interface IOpenTabQueries
    {
        List<int> ActiveTableNumbers();
        OpenTabs.TabInvoice InvoiceForTable(int table);
        Guid TabIdForTable(int table);
        OpenTabs.TabStatus TabForTable(int table);
        Dictionary<int, List<OpenTabs.TabItem>> TodoListForWaiter(string waiter)
;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Events.Cafe;
using Edument.CQRS;

namespace CafeReadModels
{
    public class OpenTabs : IOpenTabQueries,
        ISubscribeTo<TabOpened>,
        ISubscribeTo<DrinksOrdered>,
        ISubscribeTo<FoodOrdered>,
        ISubscribeTo<FoodPrepared>,
        ISubscribeTo<DrinksServed>,
        ISubscribeTo<FoodServed>,
        ISubscribeTo<TabClosed>
    {
        public class TabItem
        {
            public int MenuNumber;
            public string Description;
            public decimal Price;
        }

        public class TabStatus
        {
            public Guid TabId;
            public int TableNumber;
            public List<TabItem> ToServe;
            public List<TabItem> InPreparation;
            public List<TabItem> Served;
        }

        public class TabInvoice
        {
            public Guid TabId;
            public int TableNumber;
            public List<TabItem> Items;
            public decimal Total;
            public bool HasUnservedItems;
        }

        private class Tab
        {
            public int TableNumber;
            public string Waiter;
            public List<TabItem> ToServe;
            public List<TabItem> InPreparation;
            public List<TabItem> Served;
        }

        private Dictionary<Guid, Tab> todoByTab =
            new Dictionary<Guid,Tab>();

        public List<int> ActiveTableNumbers()
        {
            lock (todoByTab)
                return (from tab in todoByTab
                        select tab.Value.TableNumber
                       ).OrderBy(i => i).ToList();
        }

        public Dictionary<int, List<TabItem>> TodoListForWaiter(string waiter)
        {
            lock (todoByTab)
                return (from tab in todoByTab
                        where tab.Value.Waiter == waiter
                        select new
```



```
                        {
                            TableNumber = tab.Value.TableNumber,
                            ToServe = CopyItems(tab.Value, t => t.ToServe)
                        })
                        .Where(t => t.ToServe.Count > 0)
                        .ToDictionary(k => k.TableNumber, v => v.ToServe);
        }

        public Guid TabIdForTable(int table)
        {
            lock (todoByTab)
                return (from tab in todoByTab
                        where tab.Value.TableNumber == table
                        select tab.Key
                       ).First();
        }

        public TabStatus TabForTable(int table)
        {
            lock (todoByTab)
                return (from tab in todoByTab
                        where tab.Value.TableNumber == table
                        select new TabStatus
                        {
                            TabId = tab.Key,
                            TableNumber = tab.Value.TableNumber,
                            ToServe = CopyItems(tab.Value, t => t.ToServe),
                            InPreparation = CopyItems(tab.Value, t => t.InPrepar
ation),
                            Served = CopyItems(tab.Value, t => t.Served)
                        })
                        .First();
        }

        public TabInvoice InvoiceForTable(int table)
        {
            KeyValuePair<Guid, Tab> tab;
            lock (todoByTab)
                tab = todoByTab.First(t => t.Value.TableNumber == table);

            lock (tab.Value)
                return new TabInvoice
                {
                    TabId = tab.Key,
                    TableNumber = tab.Value.TableNumber,
                    Items = new List<TabItem>(tab.Value.Served),
                    Total = tab.Value.Served.Sum(i => i.Price),
                    HasUnservedItems = tab.Value.InPreparation.Any() || tab.Valu
e.ToServe.Any()
                };
        }

        private List<TabItem> CopyItems(Tab tableTodo, Func<Tab, List<TabItem>>
selector)
        {
            lock (tableTodo)
                return new List<TabItem>(selector(tableTodo));
        }

        public void Handle(TabOpened e)
        {
            lock (todoByTab)
                todoByTab.Add(e.Id, new Tab
                {
                    TableNumber = e.TableNumber,
                    Waiter = e.Waiter,
                    ToServe = new List<TabItem>(),
                    InPreparation = new List<TabItem>(),
                    Served = new List<TabItem>()
```

```
                });
        }

        public void Handle(DrinksOrdered e)
        {
            AddItems(e.Id,
                e.Items.Select(drink => new TabItem
                    {
                        MenuNumber = drink.MenuNumber,
                        Description = drink.Description,
                        Price = drink.Price
                    }),
                t => t.ToServe);
        }

        public void Handle(FoodOrdered e)
        {
            AddItems(e.Id,
                e.Items.Select(drink => new TabItem
                {
                    MenuNumber = drink.MenuNumber,
                    Description = drink.Description,
                    Price = drink.Price
                }),
                t => t.InPreparation);
        }

        public void Handle(FoodPrepared e)
        {
            MoveItems(e.Id, e.MenuNumbers, t => t.InPreparation, t => t.ToServe)
;
        }

        public void Handle(DrinksServed e)
        {
            MoveItems(e.Id, e.MenuNumbers, t => t.ToServe, t => t.Served);
        }

        public void Handle(FoodServed e)
        {
            MoveItems(e.Id, e.MenuNumbers, t => t.ToServe, t => t.Served);
        }

        public void Handle(TabClosed e)
        {
            lock (todoByTab)
                todoByTab.Remove(e.Id);
        }

        private Tab getTab(Guid id)
        {
            lock (todoByTab)
                return todoByTab[id];
        }

        private void AddItems(Guid tabId, IEnumerable<TabItem> newItems, Func<Ta
b, List<TabItem>> to)
        {
            var tab = getTab(tabId);
            lock (tab)
                to(tab).AddRange(newItems);
        }

        private void MoveItems(Guid tabId, List<int> menuNumbers,
            Func<Tab, List<TabItem>> from, Func<Tab, List<TabItem>> to)
        {
            var tab = getTab(tabId);
            lock (tab)
            {
```



```
                var fromList = from(tab);
                var toList = to(tab);
                foreach (var num in menuNumbers)
                {
                    var serveItem = fromList.First(f => f.MenuNumber == num);
                    fromList.Remove(serveItem);
                    toList.Add(serveItem);
                }
            }
        }
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace Edument.CQRS
{
    /// <summary>
    /// Aggregate base class, which factors out some common infrastructure that
    /// all aggregates have (ID and event application).
    /// </summary>
    public class Aggregate
    {
        /// <summary>
        /// The number of events loaded into this aggregate.
        /// </summary>
        public int EventsLoaded { get; private set; }

        /// <summary>
        /// The unique ID of the aggregate.
        /// </summary>
        public Guid Id { get; internal set; }

        /// <summary>
        /// Enuerates the supplied events and applies them in order to the aggre
gate.
        /// </summary>
        /// <param name="events"></param>
        public void ApplyEvents(IEnumerable events)
        {
            foreach (var e in events)
                GetType().GetMethod("ApplyOneEvent")
                    .MakeGenericMethod(e.GetType())
                    .Invoke(this, new object[] { e });
        }

        /// <summary>
        /// Applies a single event to the aggregate.
        /// </summary>
        /// <typeparam name="TEvent"></typeparam>
        /// <param name="ev"></param>
        public void ApplyOneEvent<TEvent>(TEvent ev)
        {
            var applier = this as IApplyEvent<TEvent>;
            if (applier == null)
                throw new InvalidOperationException(string.Format(
                    "Aggregate {0} does not know how to apply event {1}",
                    GetType().Name, ev.GetType().Name));
            applier.Apply(ev);
            EventsLoaded++;
        }
    }
}
```



```
ï»¿using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Xml.Serialization;
using NUnit.Framework;

namespace Edument.CQRS
{
    /// <summary>
    /// Provides infrastructure for a set of tests on a given aggregate.
    /// </summary>
    /// <typeparam name="TAggregate"></typeparam>
    public class BDDTest<TAggregate>
        where TAggregate : Aggregate, new()
    {
        private TAggregate sut;

    [SetUp]
        public void BDDTestSetup()
        {
            sut = new TAggregate();
        }

        protected void Test(IEnumerable given, Func<TAggregate, object> when, Ac
tion<object> then)
        {
            then(when(ApplyEvents(sut, given)));
        }

        protected IEnumerable Given(params object[] events)
        {
            return events;
        }

        protected Func<TAggregate, object> When<TCommand>(TCommand command)
        {
            return agg =>
            {
                try
                {
                    return DispatchCommand(command).Cast<object>().ToArray();
                }
                catch (Exception e)
                {
                    return e;
                }
            };
        }

        protected Action<object> Then(params object[] expectedEvents)
        {
            return got =>
            {
                var gotEvents = got as object[];
                if (gotEvents != null)
                {
                    if (gotEvents.Length == expectedEvents.Length)
                        for (var i = 0; i < gotEvents.Length; i++)
                            if (gotEvents[i].GetType() == expectedEvents[i].GetT
ype())
                                Assert.AreEqual(Serialize(expectedEvents[i]), Se
rialize(gotEvents[i]));
                            else
                                Assert.Fail(string.Format(
                                    "Incorrect event in results; expected a {0} but got a {1}",
```

```
                                    expectedEvents[i].GetType().Name, gotEvents[
i].GetType().Name));
                    else if (gotEvents.Length < expectedEvents.Length)
                        Assert.Fail(string.Format("Expected event(s) missing: {0}",
                            string.Join(",", EventDiff(expectedEvents, gotEvents
))));
                    else
                        Assert.Fail(string.Format("Unexpected event(s) emitted: {0}",
                            string.Join(",", EventDiff(gotEvents, expectedEvents
))));
                }
                else if (got is CommandHandlerNotDefiendException)
                    Assert.Fail((got as Exception).Message);
                else
                    Assert.Fail("Expected events, but got exception {0}",
                        got.GetType().Name);
            };
        }

        private string[] EventDiff(object[] a, object[] b)
        {
            var diff = a.Select(e => e.GetType().Name).ToList();
            foreach (var remove in b.Select(e => e.GetType().Name))
                diff.Remove(remove);
            return diff.ToArray();
        }

        protected Action<object> ThenFailWith<TException>()
        {
            return got =>
            {
                if (got is TException)
                    Assert.Pass("Got correct exception type");
                else if (got is CommandHandlerNotDefiendException)
                    Assert.Fail((got as Exception).Message);
                else if (got is Exception)
                    Assert.Fail(string.Format(
                        "Expected exception {0}, but got exception {1}",
                        typeof(TException).Name, got.GetType().Name));
                else
                    Assert.Fail(string.Format(
                        "Expected exception {0}, but got event result",
                        typeof(TException).Name));
            };
        }

        private IEnumerable DispatchCommand<TCommand>(TCommand c)
        {
            var handler = sut as IHandleCommand<TCommand>;
            if (handler == null)
                throw new CommandHandlerNotDefiendException(string.Format(
                    "Aggregate {0} does not yet handle command {1}",
                    sut.GetType().Name, c.GetType().Name));
            return handler.Handle(c);
        }

        private TAggregate ApplyEvents(TAggregate agg, IEnumerable events)
        {
            agg.ApplyEvents(events);
            return agg;
        }

        private string Serialize(object obj)
        {
            var ser = new XmlSerializer(obj.GetType());
            var ms = new MemoryStream();
            ser.Serialize(ms, obj);
            ms.Seek(0, SeekOrigin.Begin);
            return new StreamReader(ms).ReadToEnd();
```



```
        }

        private class CommandHandlerNotDefiendException : Exception
        {
            public CommandHandlerNotDefiendException(string msg) : base(msg) { }
        }
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Edument.CQRS
{
    /// <summary>
    /// Implemented by an aggregate once for each event type it can apply.
    /// </summary>
    /// <typeparam name="TEvent"></typeparam>
    public interface IApplyEvent<TEvent>
    {
        void Apply(TEvent e);
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace Edument.CQRS
{
    public interface IEventStore
    {
        IEnumerable LoadEventsFor<TAggregate>(Guid id);
        void SaveEventsFor<TAggregate>(Guid id, int eventsLoaded, ArrayList newE
vents);
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace Edument.CQRS
{
    public interface IHandleCommand<TCommand>
    {
        IEnumerable Handle(TCommand c);
    }
}
```



```
ï»¿using System;
using System.Collections;
using System.Collections.Concurrent;
using System.Threading;

namespace Edument.CQRS
{
    public class InMemoryEventStore : IEventStore
    {
        private class Stream
        {
            public ArrayList Events;
        }

        private ConcurrentDictionary<Guid, Stream> store =
            new ConcurrentDictionary<Guid, Stream>();

        public IEnumerable LoadEventsFor<TAggregate>(Guid id)
        {
            // Get the current event stream; note that we never mutate the
            // Events array so it's safe to return the real thing.
            Stream s;
            if (store.TryGetValue(id, out s))
                return s.Events;
            else
                return new ArrayList();
        }

        public void SaveEventsFor<TAggregate>(Guid aggregateId, int eventsLoaded
, ArrayList newEvents)
        {
            // Get or create stream.
            var s = store.GetOrAdd(aggregateId, _ => new Stream());

            // We'll use a lock-free algorithm for the update.
            while (true)
            {
                // Read the current event list.
                var eventList = s.Events;

                // Ensure no events persisted since us.
                var prevEvents = eventList == null ? 0 : eventList.Count;
                if (prevEvents != eventsLoaded)
                    throw new Exception("Concurrency conflict; cannot persist these events");

                // Create a new event list with existing ones plus our new
                // ones (making new important for lock free algorithm!)
                var newEventList = eventList == null
                    ? new ArrayList()
                    : (ArrayList)eventList.Clone();
                newEventList.AddRange(newEvents);

                // Try to put the new event list in place atomically.
                if (Interlocked.CompareExchange(ref s.Events, newEventList, even
tList) == eventList)
                    break;
            }
        }

        private Guid GetAggregateIdFromEvent(object e)
        {
            var idField = e.GetType().GetField("Id");
            if (idField == null)
                throw new Exception("Event type " + e.GetType().Name + " is missing an Id
field");
            return (Guid)idField.GetValue(e);
        }
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Edument.CQRS
{
    /// <summary>
    /// Implemented by anything that wishes to subscribe to an event emitted by
    /// an aggregate and successfully stored.
    /// </summary>
    /// <typeparam name="TEvent"></typeparam>
    public interface ISubscribeTo<TEvent>
    {
        void Handle(TEvent e);
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using System.Reflection;

namespace Edument.CQRS
{
    /// <summary>
    /// This implements a basic message dispatcher, driving the overall command
handling
    /// and event application/distribution process. It is suitable for a simple,
 single
    /// node application that can safely build its subscriber list at startup an
d keep
    /// it in memory. Depends on some kind of event storage mechanism.
    /// </summary>
    public class MessageDispatcher
    {
        private Dictionary<Type, Action<object>> commandHandlers =
            new Dictionary<Type, Action<object>>();
        private Dictionary<Type, List<Action<object>>> eventSubscribers =
            new Dictionary<Type, List<Action<object>>>();
        private IEventStore eventStore;

        /// <summary>
        /// Initializes a message dispatcher, which will use the specified event
 store
        /// implementation.
        /// </summary>
        /// <param name="es"></param>
        public MessageDispatcher(IEventStore es)
        {
            eventStore = es;
        }

        /// <summary>
        /// Tries to send the specified command to its handler. Throws an except
ion
        /// if there is no handler registered for the command.
        /// </summary>
        /// <typeparam name="TCommand"></typeparam>
        /// <param name="c"></param>
        public void SendCommand<TCommand>(TCommand c)
        {
            if (commandHandlers.ContainsKey(typeof(TCommand)))
                commandHandlers[typeof(TCommand)](c);
            else
                throw new Exception("No command handler registered for " + typeof(TComman
d).Name);
        }

        /// <summary>
        /// Publishes the specified event to all of its subscribers.
        /// </summary>
        /// <param name="e"></param>
        private void PublishEvent(object e)
        {
            var eventType = e.GetType();
            if (eventSubscribers.ContainsKey(eventType))
                foreach (var sub in eventSubscribers[eventType])
                    sub(e);
        }

        /// <summary>
        /// Registers an aggregate as being the handler for a particular
        /// command.
        /// </summary>
```

```
        /// <typeparam name="TAggregate"></typeparam>
        /// <param name="handler"></param>
        public void AddHandlerFor<TCommand, TAggregate>()
            where TAggregate : Aggregate, new()
        {
            if (commandHandlers.ContainsKey(typeof(TCommand)))
                throw new Exception("Command handler already registered for " + typeof(TCom
mand).Name);

            commandHandlers.Add(typeof(TCommand), c =>
                {
                    // Create an empty aggregate.
                    var agg = new TAggregate();

                    // Load the aggregate with events.
                    agg.Id = ((dynamic)c).Id;
                    agg.ApplyEvents(eventStore.LoadEventsFor<TAggregate>(agg.Id)
);

                    // With everything set up, we invoke the command handler, co
llecting the
                    // events that it produces.
                    var resultEvents = new ArrayList();
                    foreach (var e in (agg as IHandleCommand<TCommand>).Handle((
TCommand)c))
                        resultEvents.Add(e);

                    // Store the events in the event store.
                    if (resultEvents.Count > 0)
                        eventStore.SaveEventsFor<TAggregate>(agg.Id,
                            agg.EventsLoaded, resultEvents);

                    // Publish them to all subscribers.
                    foreach (var e in resultEvents)
                        PublishEvent(e);
                });
        }

        /// <summary>
        /// Adds an object that subscribes to the specified event, by virtue of
implementing
        /// the ISubscribeTo interface.
        /// </summary>
        /// <typeparam name="TEvent"></typeparam>
        /// <param name="subscriber"></param>
        public void AddSubscriberFor<TEvent>(ISubscribeTo<TEvent> subscriber)
        {
            if (!eventSubscribers.ContainsKey(typeof(TEvent)))
                eventSubscribers.Add(typeof(TEvent), new List<Action<object>>())
;
            eventSubscribers[typeof(TEvent)].Add(e =>
                subscriber.Handle((TEvent)e));
        }

        /// <summary>
        /// Looks thorugh the specified assembly for all public types that imple
ment
        /// the IHandleCommand or ISubscribeTo generic interfaces. Registers eac
h of
        /// the implementations as a command handler or event subscriber.
        /// </summary>
        /// <param name="ass"></param>
        public void ScanAssembly(Assembly ass)
        {
            // Scan for and register handlers.
            var handlers =
                from t in ass.GetTypes()
                from i in t.GetInterfaces()
                where i.IsGenericType
```



```
                where i.GetGenericTypeDefinition() == typeof(IHandleCommand<>)
                let args = i.GetGenericArguments()
                select new
                {
                    CommandType = args[0],
                    AggregateType = t
                };
            foreach (var h in handlers)
                this.GetType().GetMethod("AddHandlerFor")
                    .MakeGenericMethod(h.CommandType, h.AggregateType)
                    .Invoke(this, new object[] { });

            // Scan for and register subscribers.
            var subscriber =
                from t in ass.GetTypes()
                from i in t.GetInterfaces()
                where i.IsGenericType
                where i.GetGenericTypeDefinition() == typeof(ISubscribeTo<>)
                select new
                {
                    Type = t,
                    EventType = i.GetGenericArguments()[0]
                };
            foreach (var s in subscriber)
                this.GetType().GetMethod("AddSubscriberFor")
                    .MakeGenericMethod(s.EventType)
                    .Invoke(this, new object[] { CreateInstanceOf(s.Type) });
        }

        /// <summary>
        /// Looks at the specified object instance, examples what commands it ha
ndles
        /// or events it subscribes to, and registers it as a receiver/subscribe
r.
        /// </summary>
        /// <param name="instance"></param>
        public void ScanInstance(object instance)
        {
            // Scan for and register handlers.
            var handlers =
                from i in instance.GetType().GetInterfaces()
                where i.IsGenericType
                where i.GetGenericTypeDefinition() == typeof(IHandleCommand<>)
                let args = i.GetGenericArguments()
                select new
                {
                    CommandType = args[0],
                    AggregateType = instance.GetType()
                };
            foreach (var h in handlers)
                this.GetType().GetMethod("AddHandlerFor")
                    .MakeGenericMethod(h.CommandType, h.AggregateType)
                    .Invoke(this, new object[] { });

            // Scan for and register subscribers.
            var subscriber =
                from i in instance.GetType().GetInterfaces()
                where i.IsGenericType
                where i.GetGenericTypeDefinition() == typeof(ISubscribeTo<>)
                select i.GetGenericArguments()[0];
            foreach (var s in subscriber)
                this.GetType().GetMethod("AddSubscriberFor")
                    .MakeGenericMethod(s)
                    .Invoke(this, new object[] { instance });
        }

        /// <summary>
        /// Creates an instance of the specified type. If you are using some kin
d
```

```
        /// of DI container, and want to use it to create instances of the handl
er
        /// or subscriber, you can plug it in here.
        /// </summary>
        /// <param name="t"></param>
        /// <returns></returns>
        private object CreateInstanceOf(Type t)
        {
            return Activator.CreateInstance(t);
        }
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using System.Data.SqlClient;
using System.Data;
using System.Xml.Serialization;
using System.IO;

namespace Edument.CQRS
{
    /// <summary>
    /// This is a simple example implementation of an event store, using a SQL d
atabase
    /// to provide the storage. Tested and known to work with SQL Server.
    /// </summary>
    public class SqlEventStore : IEventStore
    {
        private string connectionString;

        public SqlEventStore(string connectionString)
        {
            this.connectionString = connectionString;
        }

        public IEnumerable LoadEventsFor<TAggregate>(Guid id)
        {
            using (var con = new SqlConnection(connectionString))
            {
                con.Open();
                using (var cmd = new SqlCommand())
                {
                    cmd.Connection = con;
                    cmd.CommandText = @"
      SELECT [Type], [Body]
      FROM [dbo].[Events]
      WHERE [AggregateId] = @AggregateId
      ORDER BY [SequenceNumber]";
                    cmd.CommandType = CommandType.Text;
                    cmd.Parameters.Add(new SqlParameter("@AggregateId", id));
                    using (var r = cmd.ExecuteReader())
                    {
                        while (r.Read())
                        {
                            yield return DeserializeEvent(r.GetString(0), r.GetS
tring(1));
                        }
                    }
                }
            }
        }

        private object DeserializeEvent(string typeName, string data)
        {
            var ser = new XmlSerializer(Type.GetType(typeName));
            var ms = new MemoryStream(Encoding.UTF8.GetBytes(data));
            ms.Seek(0, SeekOrigin.Begin);
            return ser.Deserialize(ms);
        }

        public void SaveEventsFor<TAggregate>(Guid aggregateId, int eventsLoaded
, ArrayList newEvents)
        {
            using (var cmd = new SqlCommand())
            {
                // Query prelude.
                var queryText = new StringBuilder(512);
                queryText.AppendLine("BEGIN TRANSACTION;");
```

```
                queryText.AppendLine(
                    @"IF NOT EXISTS(SELECT * FROM [dbo].[Aggregates] WHERE [Id] = @AggregateId)
            INSERT INTO [dbo].[Aggregates] ([Id], [Type]) VALUES (@AggregateId, @AggregateType);");
                cmd.Parameters.AddWithValue("AggregateId", aggregateId);
                cmd.Parameters.AddWithValue("AggregateType", typeof(TAggregate).AssemblyQualifiedName);

                // Add saving of the events.
                cmd.Parameters.AddWithValue("CommitDateTime", DateTime.UtcNow);
                for (int i = 0; i < newEvents.Count; i++)
                {
                    var e = newEvents[i];
                    queryText.AppendFormat(
                        @"INSERT INTO [dbo].[Events] ([AggregateId], [SequenceNumber], [Type], [Body], [Timestamp])
            VALUES(@AggregateId, {0}, @Type{1}, @Body{1}, @CommitDateTime);",
                            eventsLoaded + i, i);
                    cmd.Parameters.AddWithValue("Type" + i.ToString(), e.GetType().AssemblyQualifiedName);
                    cmd.Parameters.AddWithValue("Body" + i.ToString(), SerializeEvent(e));
                }

                // Add commit.
                queryText.Append("COMMIT;");

                // Execute the update.
                using (var con = new SqlConnection(connectionString))
                {
                    con.Open();
                    cmd.Connection = con;
                    cmd.CommandText = queryText.ToString();
                    cmd.CommandType = CommandType.Text;
                    cmd.ExecuteNonQuery();
                }
            }
        }

        private string SerializeEvent(object obj)
        {
            var ser = new XmlSerializer(obj.GetType());
            var ms = new MemoryStream();
            ser.Serialize(ms, obj);
            ms.Seek(0, SeekOrigin.Begin);
            return new StreamReader(ms).ReadToEnd();
        }
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class DrinksOrdered
    {
        public Guid Id;
        public List<OrderedItem> Items;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class DrinksServed
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class FoodOrdered
    {
        public Guid Id;
        public List<OrderedItem> Items;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class FoodPrepared
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class FoodServed
    {
        public Guid Id;
        public List<int> MenuNumbers;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class OrderedItem
    {
        public int MenuNumber;
        public string Description;
        public bool IsDrink;
        public decimal Price;
    }
}
```



```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class TabClosed
    {
        public Guid Id;
        public decimal AmountPaid;
        public decimal OrderValue;
        public decimal TipValue;
    }
}
```

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Events.Cafe
{
    public class TabOpened
    {
        public Guid Id;
        public int TableNumber;
        public string Waiter;
    }
}
```