Proxy and Interceptor Patterns

DEVELOPMENTOR
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Motivation

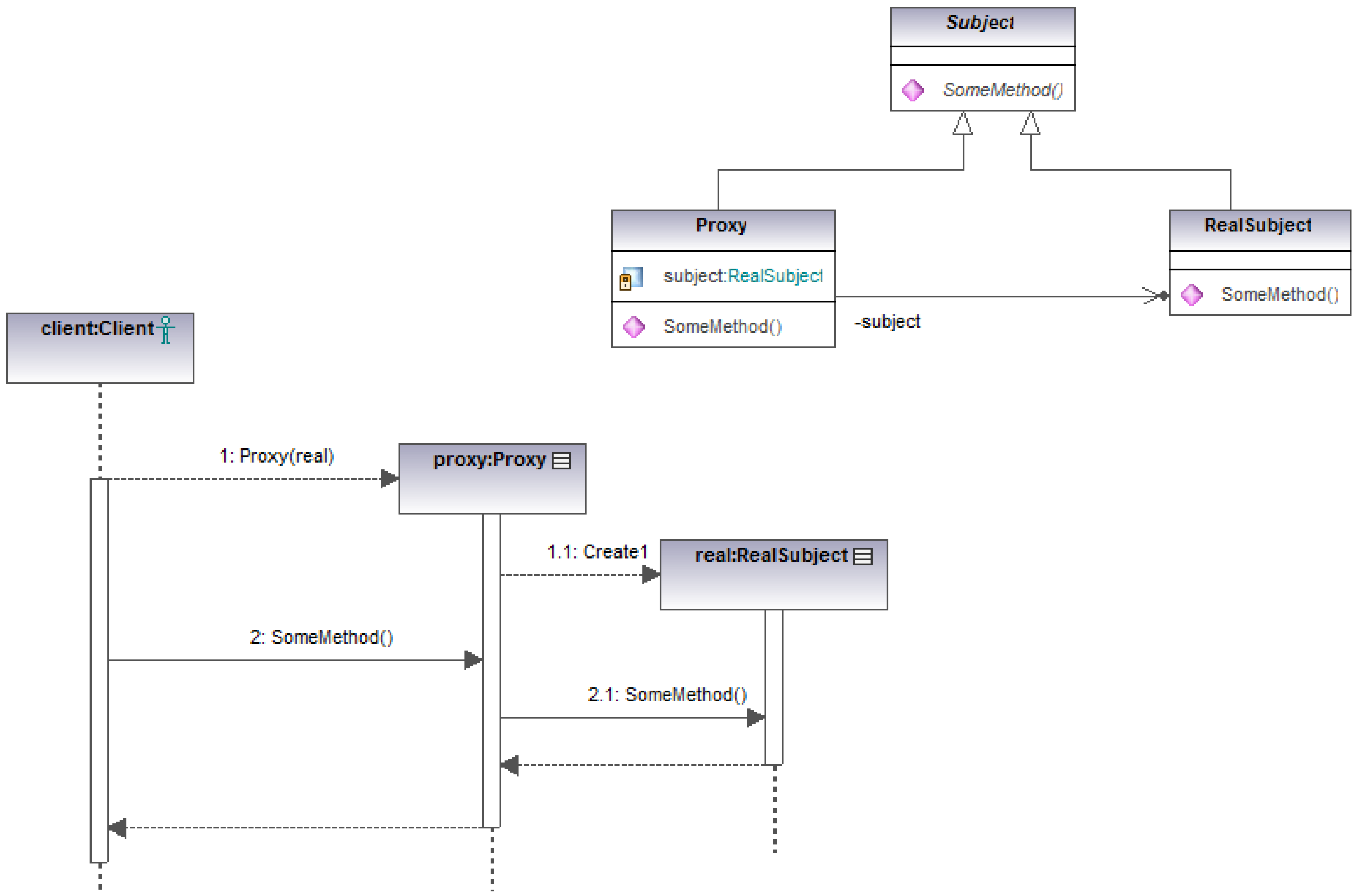
- Single threaded type now needs to live in a multi threaded environment
  - Control access so only one caller at a time
  - Or marshal all calls to a given thread
- The same invocation is made by lots of callers, producing the same response, this is hurting performance
  - Cache request/response pairs
- Object is actually remote
  - Need to convert the invocation into a serialized message ready for network transportation
- Object has methods only certain users can invoke
  - Need to check security policy prior to invoking

In essence we want to control access

Proxy Pattern

- Controlling access
  - Prior to the object functionality being invoked we wish to perform some additional steps
  - We don’t want to modify the existing code
    - “Closed for modification open for extension”
- To achieve this create a type that
  - Is compatible with the one we wish to control access too.
    - Create a common base type
  - Have this type wrap up the object being protected
  - Have the client work with an instance of this type, but in terms of the common base type

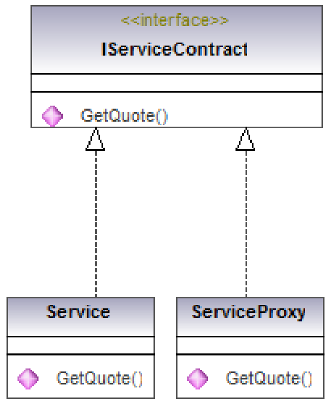
Proxy Pattern

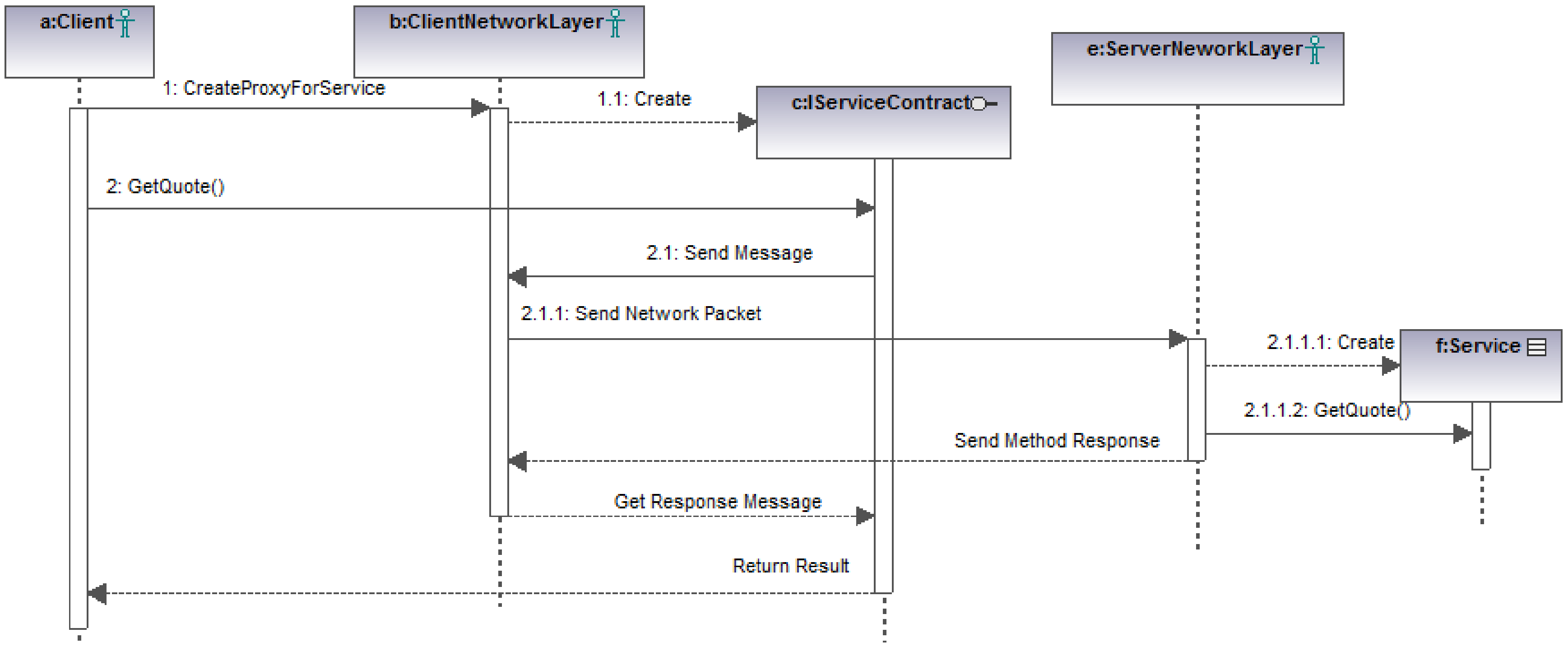


Cross App Domain boundaries

- Common use of proxy pattern is for invocations that cross app domain boundaries
  - RPC aka RMI
  - Web Services
- WCF utilises this pattern to produce client side proxies
  - Remote object only created on first call
  - Client continues to use in app domain programming model
    - Aspects of the implementation do leak

Proxies for Remote Invocation

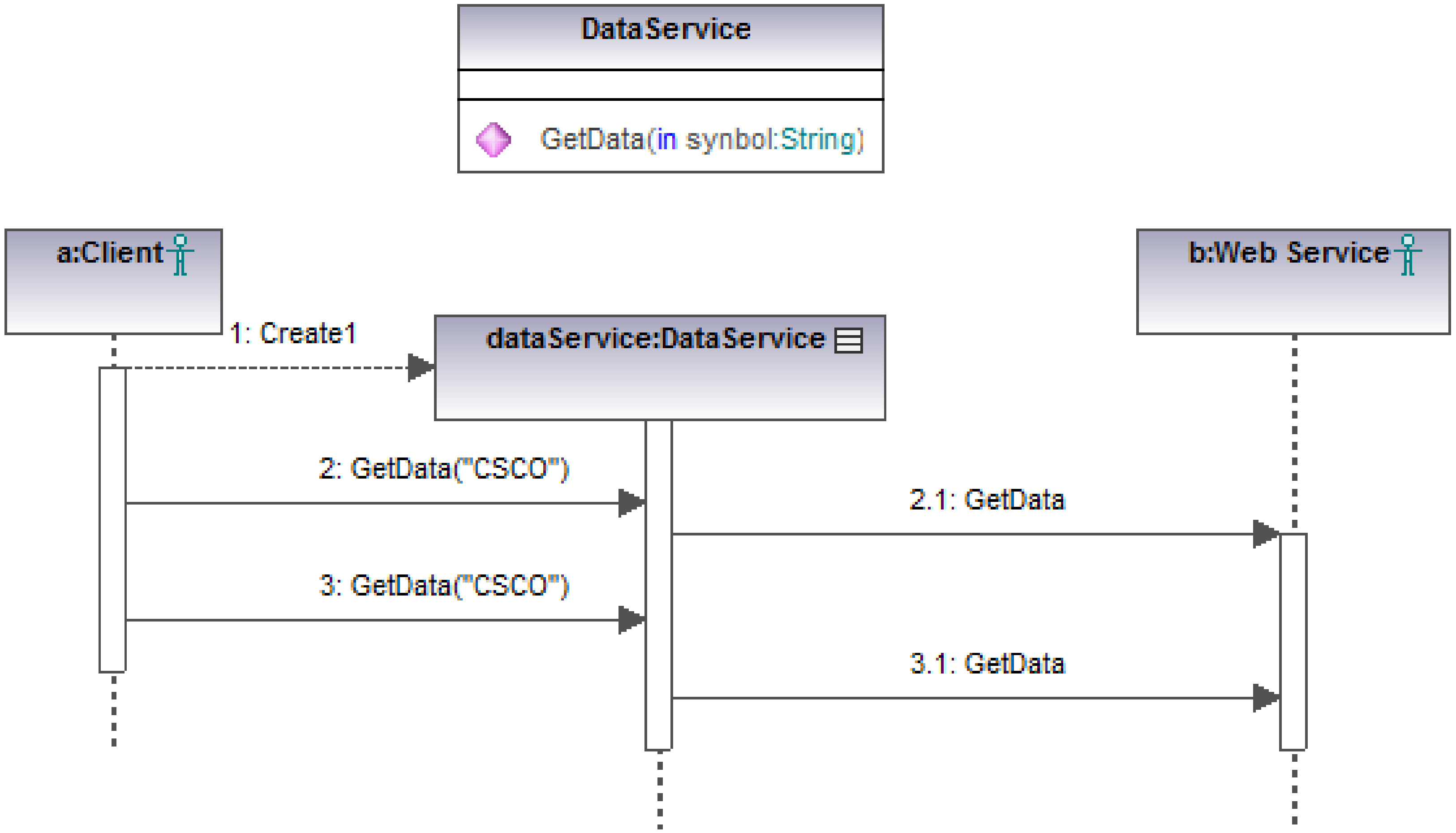




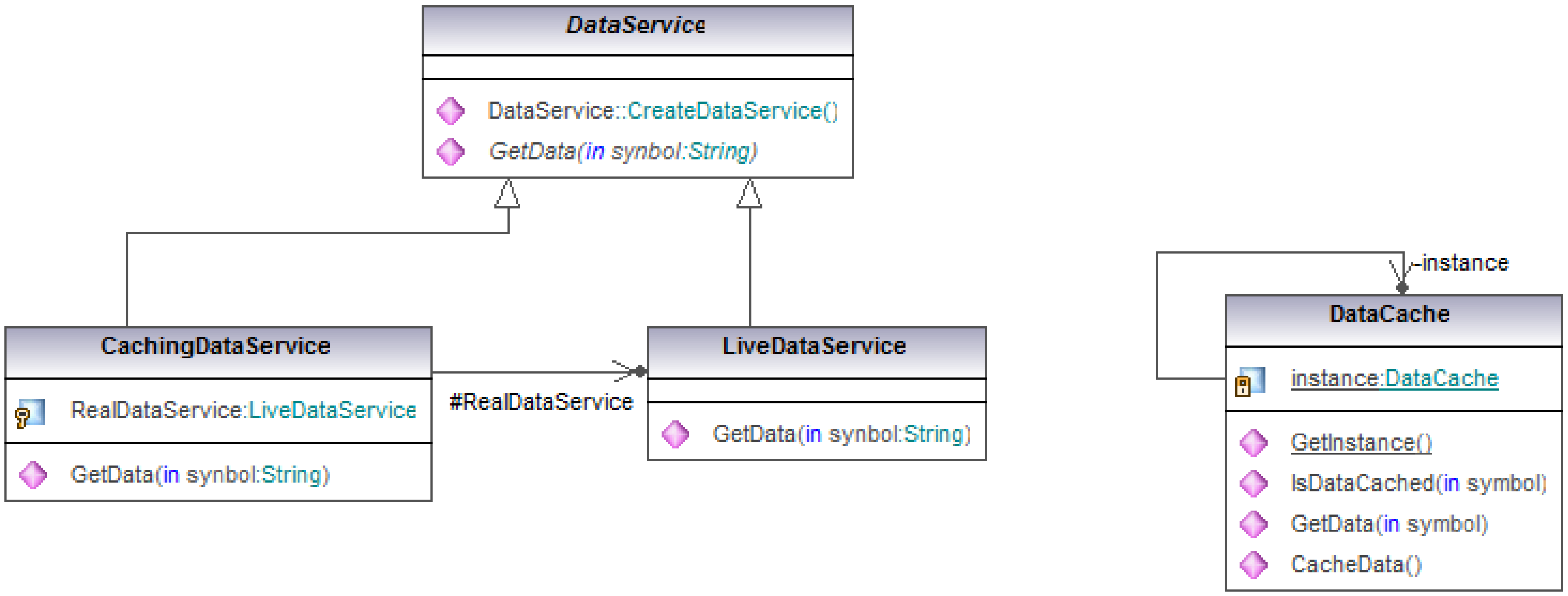
Proxies for Caching

- In multi user applications the same request can be made many times
  - Get the Stock price for CSCO
  - What’s on Channel X now
  - What’s the IP address of www.google.com
- It would therefore seem sensible for expensive operations to be cached
- Solution
  - Create a Caching proxy
  - The Proxy records requests and replays responses
    - If reoccurring request, return the cached result
    - Otherwise invoke the real subject, and cache the result

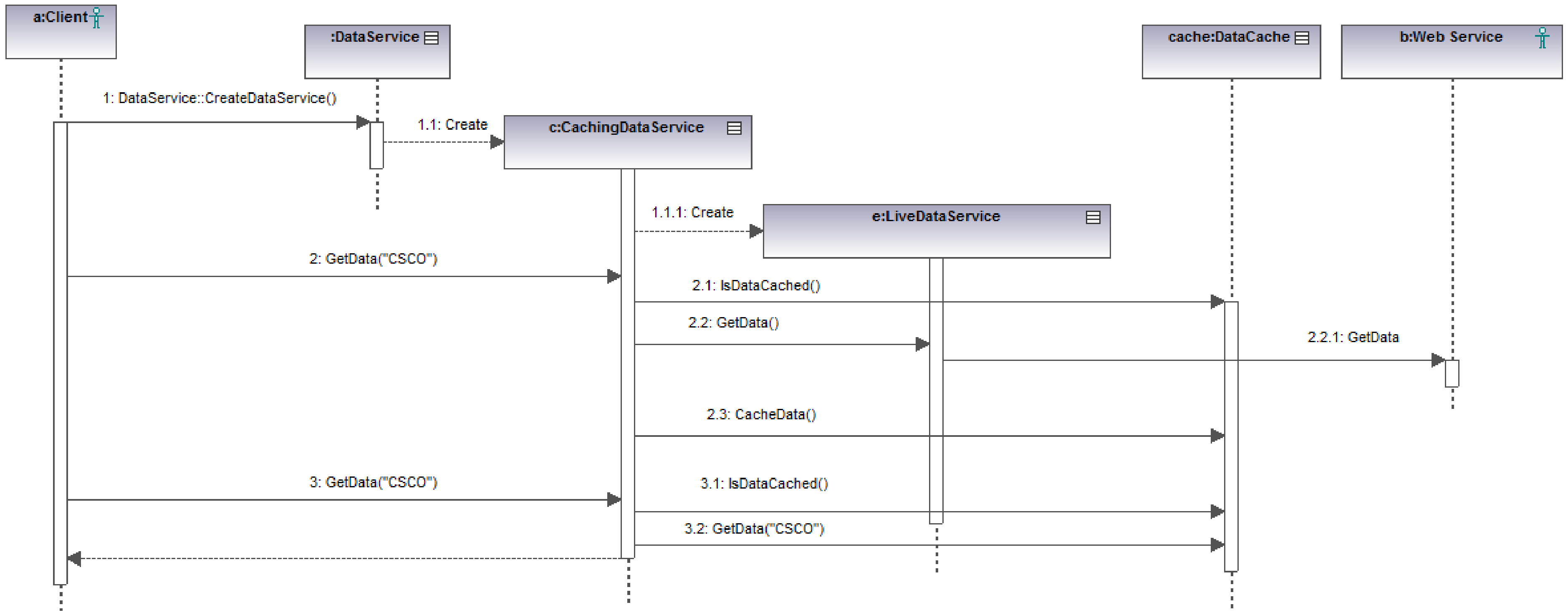
Non-Caching Data Service



Caching Data Service



Caching in action



Proxies for Synchronization

- Thread safety when not required increases overhead
- Types are often written not to be thread safe
- Synchronization proxies allow the addition of synchronization code at runtime when it is required
- Object can only be accessed by one thread at a time
  - Proxy method wraps each with a synchronization primitive similar to Java’s synchronized keyword
- Object can only be accessed on a given thread
  - Proxy method provides a means of marshalling the call on to the appropriate thread. Typically done via ThreadSynchronizationContext

Simple Synchronized Proxy

```
class SynchronizedAccount : Account {
   private Account account;
   private object sync = new object();

   public SynchronizedAccount(Account account){
     this.account = account;
   }

   public override void Credit(decimal amount){
     lock (sync)  {
       account.Credit(amount);
     }
   }
}
```

Synchronization Proxies Gotcha

- In the code below assume
  - queue is a Synchronized Wrapped version
  - You can’t de-queue an empty queue
  - Multiple threads are using this method
- Are there any problems with this code ?

```
void ProcessQueue(IQueue queue)
{
        if ( queue.Empty() == false )
        {
           object item = queue.Dequeue();
        }
}
```

Proxies for Security

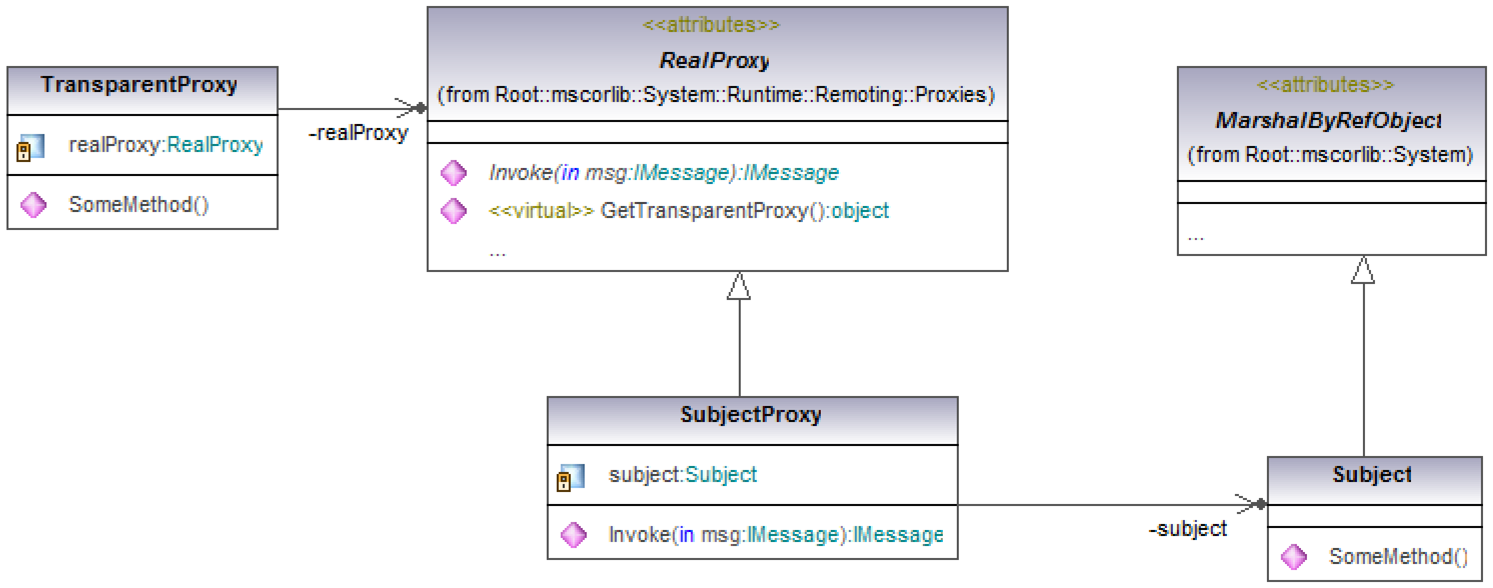
- Security always comes last ;-)
  - Not a problem if you implement with a proxy
- Separate logic and security
  - Allows both to evolve independently
  - Security rules can be clearly seen and understood
- Security Proxies can
  - Validate that the method can be invoked by the user
  - Provide Audit trail of method invocation
    - Who did what and when

Dynamic Proxies

- Building some proxy types can be tedious
  - RMI
  - Security
  - Synchronization
- Managed runtimes offer the ability to build a proxy at runtime
  - Write once and can be used for a range of types
- Dynamic Proxy has two parts
  - Transparent Proxy(TP)
    - Client makes method calls against TP
  - Real Proxy, Implementation of the proxy
    - Derives from RealProxy
    - Creates the TP
    - Handles method calls to TP through a single invoke method

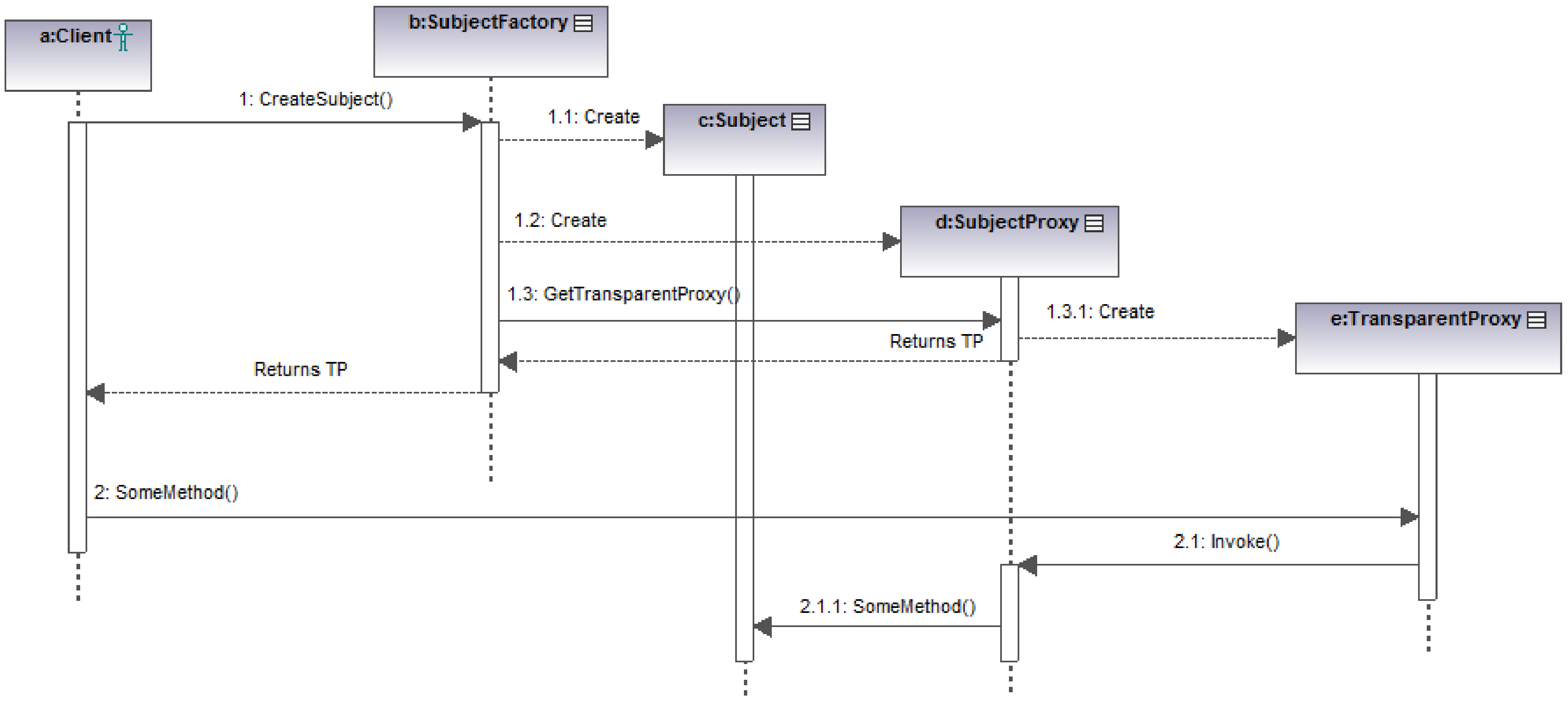
TP-RP and Subject

- Not all types can have a Transparent Proxy
  - Type must derive from MarshalByRefObject (MBRO)
  - Type then has JIT compiling inlining disabled for type



TransparentProxy
realProxy:RealProxy
SomeMethod()
-realProxy
<<attributes>>
RealProxy
(from Root::mscorlib::System::Runtime::Remoting::Proxies)
Invoke(in msg:IMessage):IMessage
<<virtual>> GetTransparentProxy():object
...
<<attributes>>
MarshalByRefObject
(from Root::mscorlib::System)
...
SubjectProxy
subject:Subject
Invoke(in msg:IMessage):IMessage
-subject
Subject
SomeMethod()

Dynamic Proxy in action



Dynamic Proxy, Coding Elements

```
class Subject : MarshalByRefObject {}

class MyProxy : RealProxy {
   private Subject subject;

    MyProxy(Subject subject) : base(typeof(Subject) ){
       this.subject = subject;
    }

    // All calls against TP routed to here
    public override IMessage Invoke(IMessage msg) {  }
}

...
MyProxy proxy = new MyProxy(subject);

Subject subject = (Subject)proxy.GetTransparentProxy();
```

Implementing the Proxy

```
class SubjectAuditProxy : RealProxy {
  private Subject realInstance;

  public SubjectAuditProxy(Subject instance) :
        base(typeof(Subject)) {
    realInstance = instance;
  }

  // One method call for every method on the subject
  public override IMessage Invoke(IMessage msg) {
   IMethodCallMessage methodCall = (IMethodCallMessage)msg;

   Console.Write("{0} on object {2} has invoked {1}",
                 Thread.CurrentPrincipal.Identity.Name,
                 methodCall.MethodName ,
                 realInstance );

   // Invoke the message against the realInstance
   return RemotingServices.ExecuteMessage(realInstance,
                  (IMethodCallMessage) msg);
   }
}
```

Single xxxProxy for all types

```
class AuditProxy<T> : RealProxy where T:MarshalByRefObject {
  private T realInstance;

  public AuditProxy(T instance) : base(typeof(T))
  {
     realInstance = instance;
  }

  public override IMessage Invoke(IMessage msg) {
   IMethodCallMessage methodCall = (IMethodCallMessage)msg;

   // Pre call code

   IMessage returnMsg = RemotingServices.ExecuteMessage(realInstance,
                                  (IMethodCallMessage) msg);
   // Post call code

    return returnMsg;
  }
}
```

Interceptors

- CLR based proxies limited use due to dependency on MBRO
- Prefer to use runtime code generation
  – NuGet “Castle Windsor Core, Dynamic Proxy”
- Framework dynamically builds proxy for a given interface
- Interceptors implement standard interface
  – Allows building general purpose interceptors
- Generated proxy keeps list of interceptors
- Calls to proxy, results in proxy calling first interceptor
  – Interceptors can
    - Modify call parameters before proceeding
    - Modify return parameters
    - Decide if to call other interceptors

Creating an Interceptor

- Interceptor implements Intercept method
  - IInvocation object contains call information
  - Calling Proceed will result in next interceptor being called or final target

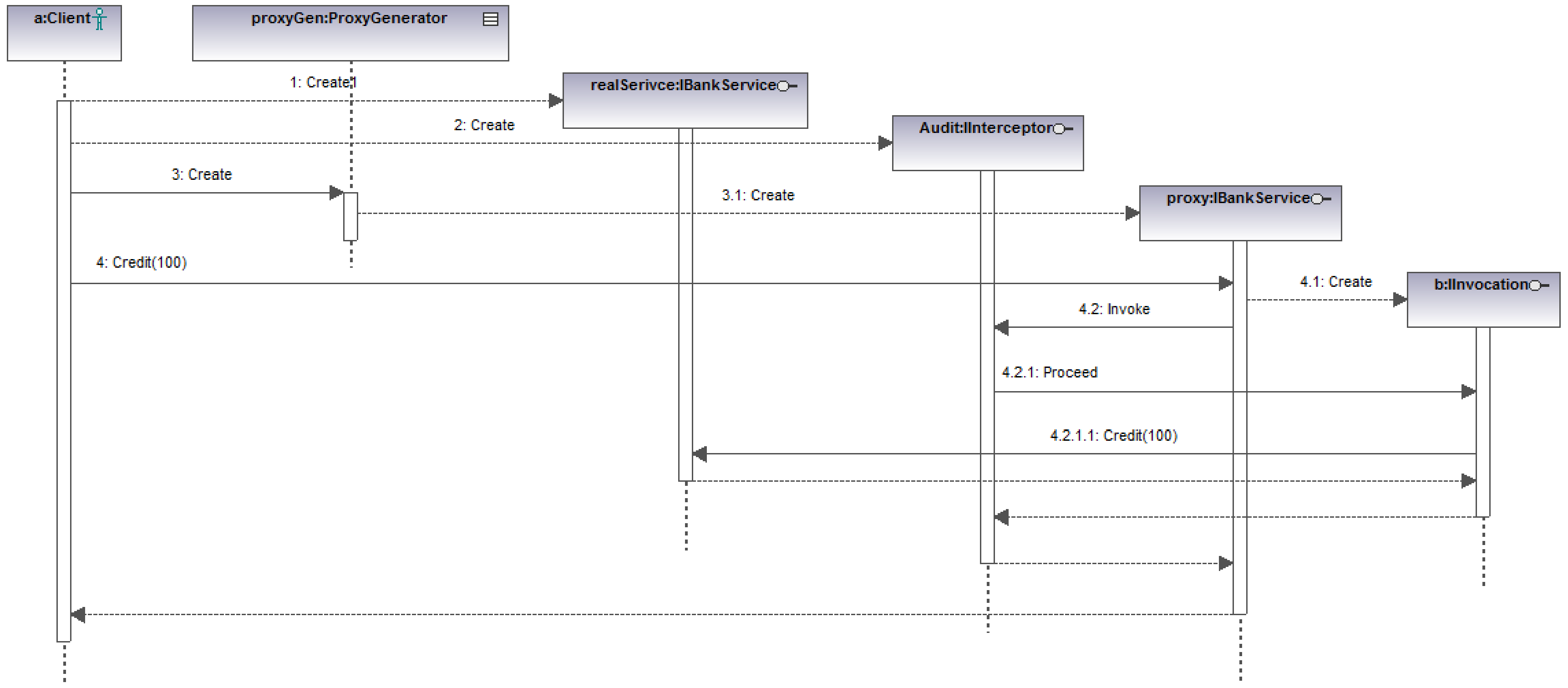
```
public class AuditInterceptor : IInterceptor {
  public void Intercept(IInvocation invocation)
  {
    Console.WriteLine("{0}({1})",invocation.Method.Name,
                                String.Join(",", invocation.Arguments));

    invocation.Proceed();

    if (invocation.ReturnValue != null){
     Console.WriteLine("Returns {0}" , invocation.ReturnValue);
    }
  }
}
```

Using the Interceptor

- Proxy Generator used to create general purpose proxy
  - Set of interceptors supplied
- Calls to proxy will result in calling the supplied interceptor auditInterceptor

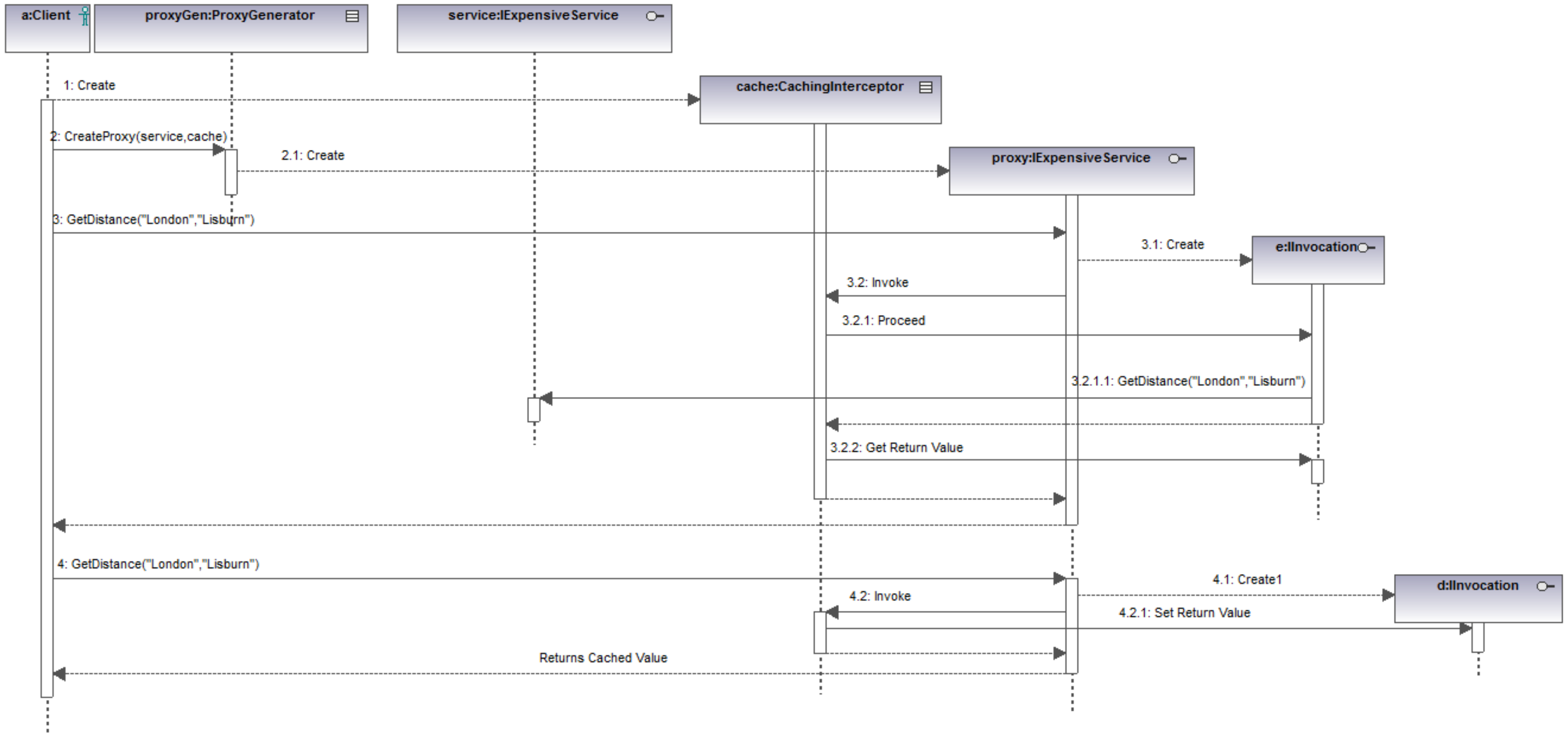
```
IBankAccount account = new Account();
AuditInterceptor auditInterceptor = new AuditInterceptor();

var proxyGenerator = new ProxyGenerator();

IBankAccount accountProxy = proxyGenerator
               .CreateInterfaceProxyWithTarget<IBankAccount>(
                                account, auditInterceptor);
accountProxy.Credit(100);

Console.WriteLine(accountProxy.Balance);
```

Castle Windsor Interceptor in action



Caching Interceptor

- Caching interceptor decides if to call final target if no cache value for current method/arguments



Summary

- If you need to control access to methods consider the Proxy Pattern
- For general purpose proxies consider a Dynamic Proxy
- Proxy Pattern is similar to decorator in structure but differs in intent
  - Proxy controls access
  - Decorator adds behaviour
- Dynamic proxies + Interceptors allow the building of general purpose proxies