Instituto Superior Técnico,
Universidade Técnica de Lisboa, Portugal

ANACOM

Técnicas de Refactorização e Teste

GrupoA-0211

64714, Alexandre Quitério, alexandre.quiterio@ist.utl.pt
66978, Emídio Silva, rafaelsilva@ist.utl.pt
67013, João Oliveira, joao.delgado.oliveira@ist.utl.pt
67020, Joaquim Guerra, joaquimguerra@ist.utl.pt
67036, Miguel Neves, miguel.neves@ist.utl.pt
67074, Rodrigo Bruno, rodrigo.bruno@ist.utl.pt

15 de Maio de 2012

Introdução

Este relatório pretende explicitar as técnicas utilizadas pelo nosso grupo para refatorização e teste do projeto. Começaremos por explicar como implementámos testes de unidade no início do desenvolvimento e explicaremos de seguida como fomos testando as funcionalidades do projeto de Sistemas Distribuídos. Por fim, iremos referir as técnicas de refatorização que utilizámos ao longo da elaboração do projeto.

Testes de unidade dos serviços da ANACOM

A partir da segunda parte do projeto de Engenharia de Software, começámos a implementar testes de unidade para os serviços sobre o domínio da ANACOM. Implementámos uma superclasse de teste de serviço responsável por implementar as operações necessárias a todos os testes tais como inicialização da base de dados e limpeza da mesma. Além disso, a superclasse também ficou responsável por implementar operações como registo de operadoras e telemóveis, permitindo a não duplicação de código nas subclasses.

Testes de segurança

Para a parte da segurança, decidimos criar um novo projeto para a autoridade de certificação. Isto permitiu-nos ter um código totalmente independente do desenvolvido até então, logo, conseguimos criar uma autoridade de certificação que teoricamente poderia ser utilizada por qualquer entidade interessada nos seus serviços.

Decidimos também utilizar uma estrutura de diretórios,semelhante à utilizada no projeto porque entendemos que se adequava também a esta situação. Além da estrutura, decidimos utilizar também a Fenix Framework para mantermos uma base de dados persistente de certificados revogados.

Tendo uma estrutura semelhante ao projeto ANACOM, decidimos também começar por testar da mesma forma, isto é, verificando os serviços sobre o domínio com testes de unidade e aplicando os princípios já descritos anteriormente.

Uma vez que a implementação dos requisitos de segurança implica a criação de handlers, decidimos testar esta componente antes de a implementar no projeto ANACOM. Isto permitiu-nos ter um cliente de teste bastante mais simples do que o que teríamos no projeto principal e, logo, menos propício a erros. O cliente faz apenas chamadas à autoridade de certificação e, uma vez que o procedimento de comunicação com esta é igual ao procedimento de comunicação entre servidores de aplicação e apresentação, isso significa que se tivermos a interação a funcionar no cliente de teste, deveremos ter a comunicação a funcionar no projeto principal.

Partimos desse princípio, testámos tudo e resolvemos os erros e, assim, quando foi necessário colocar esse código no programa principal, conseguimos um mínimo de problemas de sincronização.

Testes de Replicação de Pacotes e Tolerância a Faltas

Para testar a replicação de pacotes foi criada uma classe de testes que utiliza jUnit e jMock. Esta classe testa se a classe ReplicatedCommunication está a replicar os pedidos devidamente.

Para simular as situações de teste foram utilizadas várias classes "dummy" que derivam das classes que são realmente utilizadas no projeto (por exemplo a classe AnacomApplicationServerPortType). Esta técnica foi utilizada para permitir alterar o comportamento normal destas classes para o desejado. É de notar que esta técnica foi necessária (e não apenas o jMock) porque é necessário testar a comunicação entre diversos objetos, neste caso os Handlers de callback utilizados na comunicação assíncrona, AnacomApplicationServerPortType (a implementação assíncrona do servidor) e ReplicatedCommunication. Devido a este facto não basta contar as chamadas e comparar argumentos e/ou retornos mas também foi necessário incluir algum comportamento.

Foi utilizado o jMock para permitir avaliar as chamadas feitas ao AnacomApplicationServerPortType. Apesar de ter sido criada uma classe "dummy", foi criado um "mock object" a partir da interface de comunicação assíncrona. Este objeto permitiu analisar o número e quais as chamadas assíncronas que foram lançadas e permitiu avaliar a sua correção. Foi testada ainda a situação de retorno de exceções de comunicação.

Para além da replicação de pacotes foram feitos testes individuais sobre as diferentes classes usadas para implementar o protocolo de tolerância a faltas, como o ByzantineFailureTolerator e o GenericComparator por exemplo. Foram testados maioritariamente os diferentes casos possíveis de respostas dentro do modelo de faltas considerado no enunciado de Sistemas de Distribuídos.

Refactorização de Código

Um típico problema durante as primeiras fases de implementação é ligarmos mais à funcionalidade e não termos tanta preocupação com a legibilidade do código produzido.

Esta abordagem, apesar de não ser aconselhada, acaba sempre por ocorrer em partenum projeto com alguma magnitude. A solução seguida para diminuir o tamanho das funções é acompanhar a abordagem de código auto-comentado, i.e, sempre que tivermos um conjunto de ações semanticamente ligadas, estas devem ser extrapoladas do corpo da função principal e encapsuladas numa função que por poucas palavras explique quais as ações da chamada à mesma. Ao seguir esta abordagem, é mais fácil ler o código e dividir as complexidades de raciocínio por camadas.

Uma outra técnica utilizada para refatorização de código foi o TemplateMethod. No contexto do domínio da ANACOM, os estados do Phone (ex: MakingCallState) têm funções que se comportam de maneira igual para diferentes tipos de comunicações.

Uma forma de não ter várias funções para vários tipos de comunicação é obrigar as subclasses de MakingCallState a criar os tipos de comunicação que implementam. Desta forma, como toda a lógica da função é implementada pela superclasse, e só o tipo de comunicação é contextualizado pela subclasse, melhorando significativamente a flexibilidade do código atual e para futuros acréscimos.

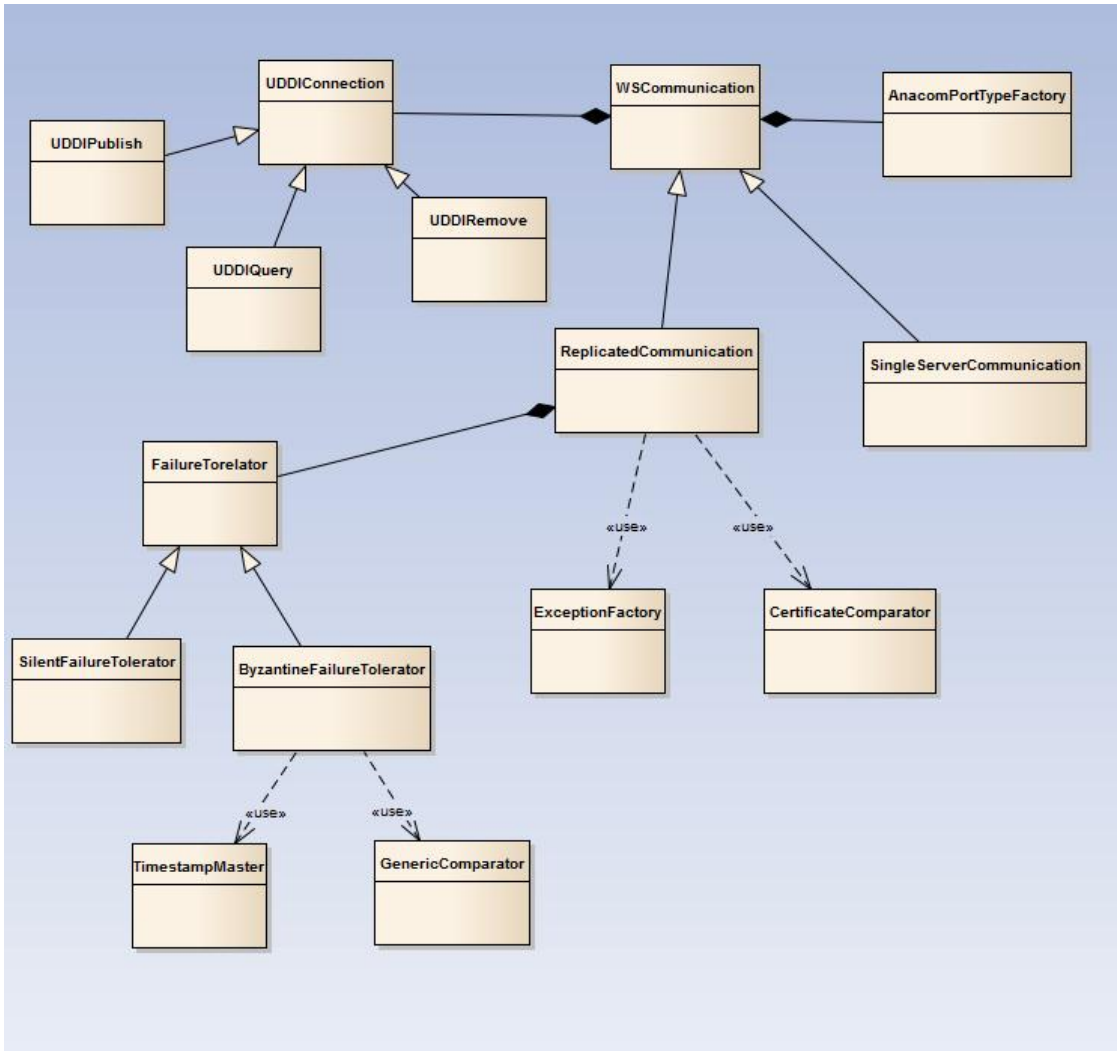
Para garantir que todo o comportamento da classe Phone é tratado nas classes concretas de PhoneState, toda a lógica de verificação de estado válido ou inválido, de adicionar chamadas recebidas, efetuadas ou atualização da útlima comunicação é implementada pelos estados de Phone. Desta forma

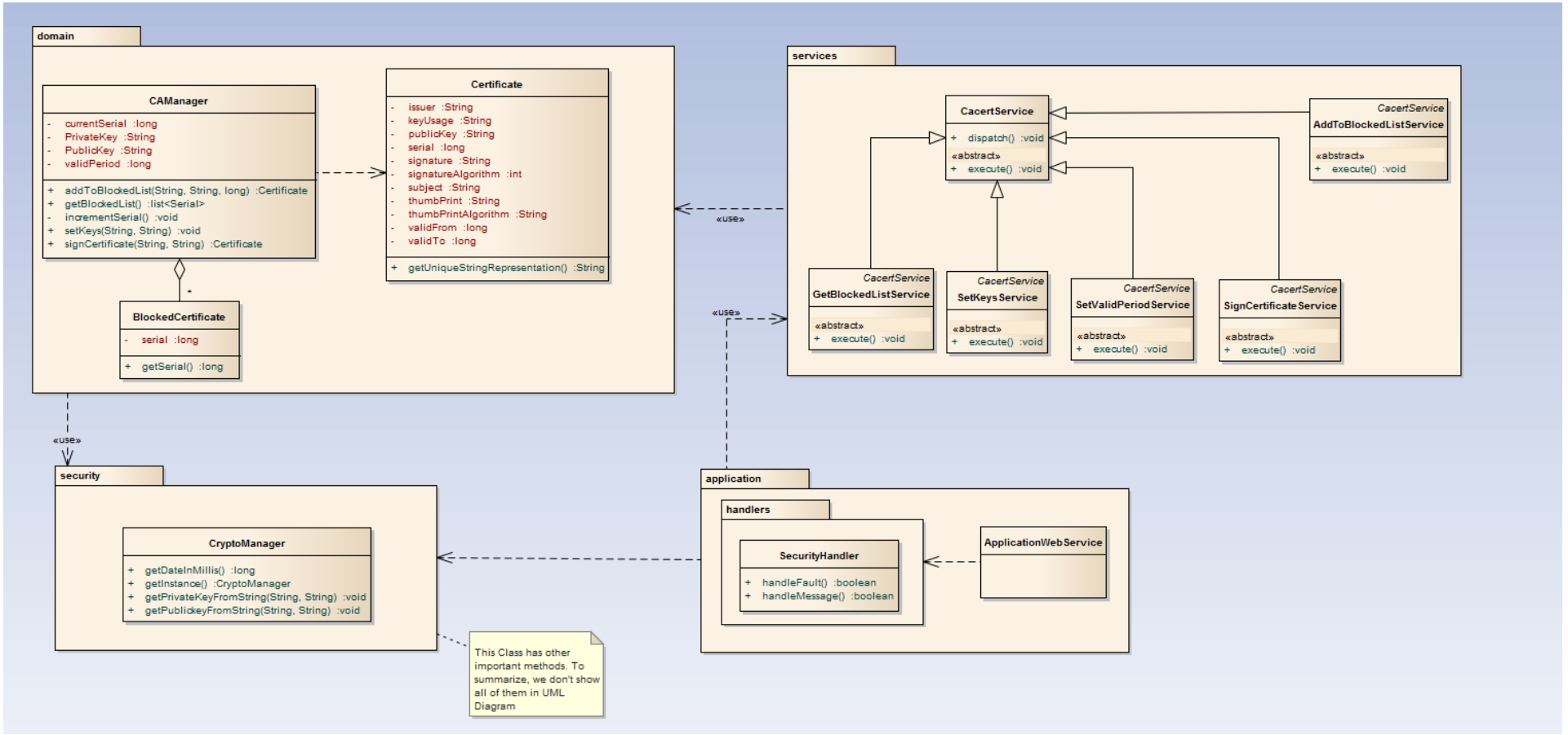
garantimos que o phone não sabe qual a lógica dos seus estados e como tal, subclasses de Estados mais genéricos podem implementar métodos

Durante o progresso da nossa implementação optámos por utilizar o método sub-string como solução para obter o prefixo de um dado telemóvel. Esta decisão inicial veio a mostrar-se péssima como realização isto porque caso o prefixo do telemóvel fosse alterado teríamos de mudar código desde o GWT até ao domínio.A solução utilizada para garantir a modularização e independência do projeto face ao prefixo é garantir que esta representação só é criada num ponto do projeto. Essa representação está definida do lado da Network.

De modo a separar a representação externa de estados, tipos de telemóvel e comunicações, optámos por utilizar classes singleton que nos permitem representar universalmente um dado tipo. Este tipo de representações são importantes por exemplo para evitar ambiguidades de representação quando o cliente tenta mudar o estado do seu telemóvel. Um dos passos intermédios é a construção de um DTO que tem um campo String que representa o tipo state. A questão deste problema está em como é que o service sabe qual é a representação externa dos estados do domínio a partir do GWT. Estas representações externas permitem-nos uniformizar a representação dos tipos, quer do lado do servidor de apresentação, quer do lado dos serviços e do servidor de aplicação.

Anexos


UDDIConnection
WSCommunication
AnacomPortTypeFactory
UDDIPublish
UDDIRemove
UDDIQuery
ReplicatedCommunication
SingleServerCommunication
FailureTorelator
«use»
«use»
ExceptionFactory
CertificateComparator
SilentFailureTolerator
ByzantineFailureTolerator
«use»
«use»
TimestampMaster
GenericComparator


domain
CAManager
- currentSerial :long
- PrivateKey :String
- PublicKey :String
- validPeriod :long
+ addToBlockedList(String, String, long) :Certificate
+ getBlockedList() :list<Serial>
- incrementSerial() :void
+ setKeys(String, String) :void
+ signCertificate(String, String) :Certificate
Certificate
- issuer :String
- keyUsage :String
- publicKey :String
- serial :long
- signature :String
- signatureAlgorithm :int
- subject :String
- thumbPrint :String
- thumbPrintAlgorithm :String
- validFrom :long
- validTo :long
+ getUniqueStringRepresentation() :String
BlockedCertificate
- serial :long
+ getSerial() :long
«use»
services
CacertService
+ dispatch() :void
«abstract»
+ execute() :void
CacertService
AddToBlockedListService
«abstract»
+ execute() :void
CacertService
GetBlockedListService
«abstract»
+ execute() :void
CacertService
SetKeysService
«abstract»
+ execute() :void
CacertService
SetValidPeriodService
«abstract»
+ execute() :void
CacertService
SignCertificateService
«abstract»
+ execute() :void
«use»
«use»
security
CryptoManager
+ getDateInMillis() :long
+ getInstance() :CryptoManager
+ getPrivateKeyFromString(String, String) :void
+ getPublickeyFromString(String, String) :void
This Class has other important methods. To summarize, we don't show all of them in UML Diagram
application
handlers
SecurityHandler
+ handleFault() :boolean
+ handleMessage() :boolean
ApplicationWebService