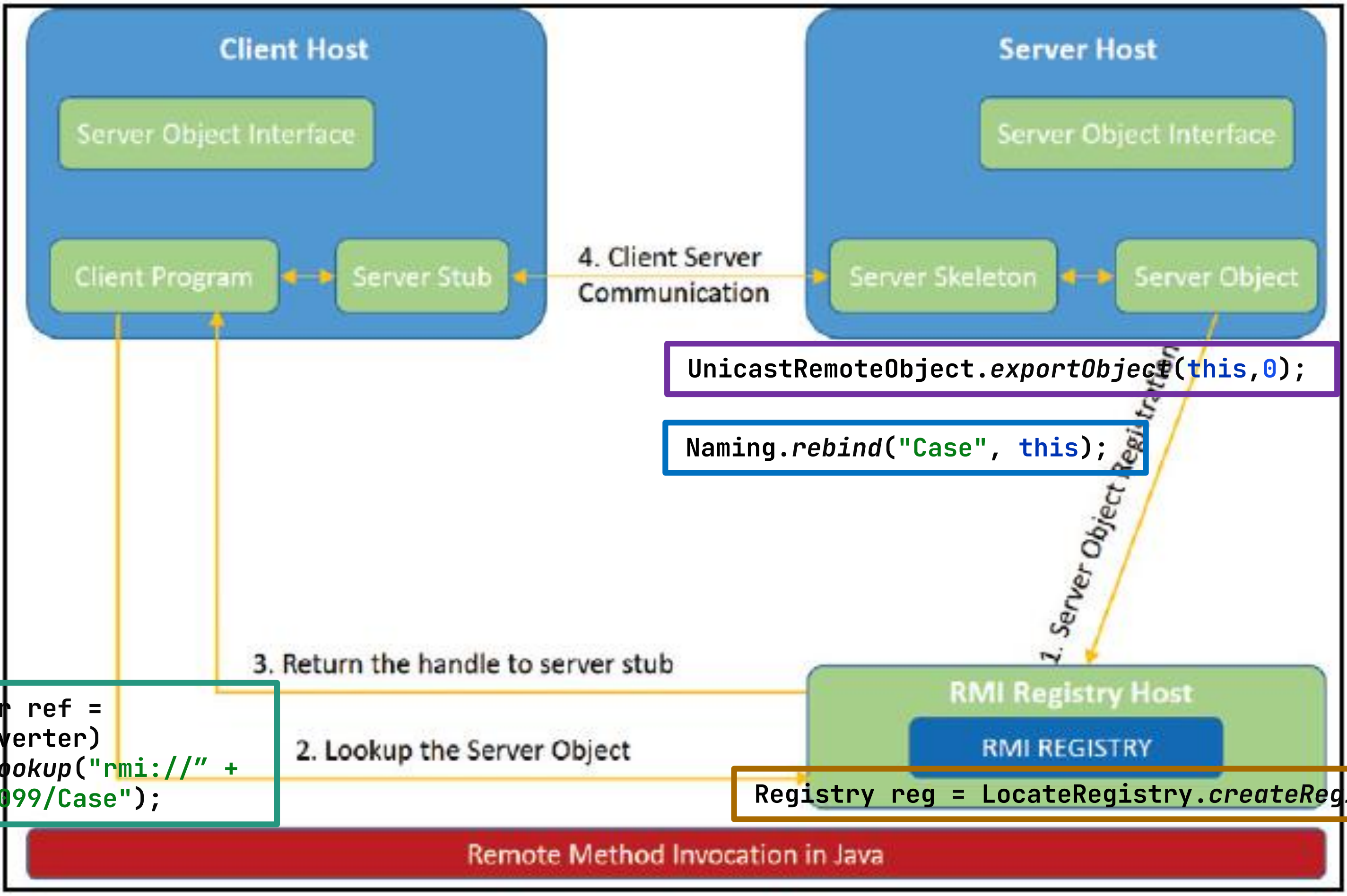
Life is great
VIA University College

Software Development with UML and Java 2

RMI communication



Client Host
Server Object Interface
Client Program
Server Stub
4. Client Server Communication
Server Host
Server Object Interface
Server Skeleton
Server Object
1. Server Object Registration
3. Return the handle to server stub
2. Lookup the Server Object
RMI Registry Host
RMI REGISTRY
Remote Method Invocation in Java

```
UnicastRemoteObject.exportObject(this,0);
```

```
Naming.rebind("Case", this);
```

```
TextConverter ref =
    (TextConverter)
    Naming.lookup("rmi://" +
"localhost:1099/Case");
```

```
Registry reg = LocateRegistry.createRegistry(1099);
```

RmiServer (in parts) – Registry

```
public class RmiServer extends UnicastRemoteObject
                       implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      ServerInterface rmiServer = new RmiServer();

      Naming.rebind("Case", rmiServer);
      System.out.println("Starting server...");
   }
   public RmiServer() throws RemoteException
   {
      super();
   }
```

RmiServer (in parts) – Registry

```
public class RmiServer extends UnicastRemoteObject
                       implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      ServerInterface rmiServer = new RmiServer();

      reg.rebind("Case", rmiServer);
      System.out.println("Starting server...");
   }
   public RmiServer() throws RemoteException
   {
      super();
   }
```

Start the Registry (in its own try-catch block)

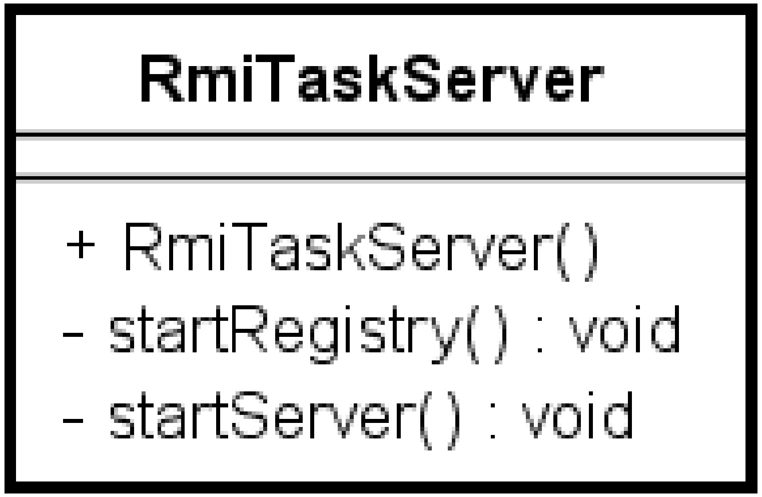
- If registry is already started:

```
java.rmi.server.ExportException: Port already in use: 1099;
nested exception is:
    java.net.BindException: Address already in use: JVM_Bind
    ...
```

```
try
{
   Registry reg = LocateRegistry.createRegistry(1099);
   System.out.println("Registry started...");
}
catch (java.rmi.server.ExportException ex)
{
  // already started
  System.out.println("Registry already started?"
                   + " Error: " + ex.getMessage());
}
```

RmiTaskServer with private methods

```
public RmiTaskServer()
{
    // ...
    startRegistry();
    startServer();
}
```



Create and publish the remote object

1) Create:
   a) Create an object of the class that implements the remote interface (in a main method or in another class), or
   b) Use `this` if publishing is done in the class implementing the remote interface
2) Publish:
   a) The runtime need to create a TCP server socket and start waiting for connecting clients. `UnicastRemoteObject` is used for this purpose
      I. Either extending `UnicastRemoteObject`, or
      II. Calling static method `exportObject` in `UnicastRemoteObject`
   b) Upload the stub to the registry and bind it to a name/string

RmiServer (in parts) - main method

```
public class RmiServer extends UnicastRemoteObject
                          implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      ServerInterface rmiServer = new RmiServer();

      Naming.rebind("Case", rmiServer);
      System.out.println("Starting server...");
   }
   public RmiServer() throws RemoteException
   {
      super();
   }
```

Upload the stub to registry and bind it to a name/string

Publish the object (start listening for clients)

Create an object of the class implementing the remote interface

RmiServer (in parts) - main method

```
public class RmiServer implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      ServerInterface rmiServer = new RmiServer();

      UnicastRemoteObject.exportObject(rmiServer, 0);

      Naming.rebind("Case", rmiServer);
      System.out.println("Starting server...")
   }
   public RmiServer()
   {
   }
```

Upload the stub to registry and bind it to a name/string

Publish the object (start listening for clients)

Create an object of the class implementing the remote interface

RmiServer (in parts) - main method

```
public class RmiServer implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      ServerInterface rmiServer = new RmiServer();

      ServerInterface stub = (ServerInterface)
             UnicastRemoteObject.exportObject(rmiServer, 0);

      Naming.rebind("Case", stub);
      System.out.println("Starting server...");
   }
   public RmiServer()
   {
   }
```

Upload the stub to registry and bind it to a name/string

Publish the object (start listening for clients)

Create an object of the class implementing the remote interface

RmiServer (in parts) – constructor

```
public class RmiServer implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      RmiServer server = new RmiServer();
   }
   public RmiServer() throws RemoteException
   {
      ServerInterface stub = (ServerInterface)UnicastRemoteObject
            .exportObject(this, 0);

      Naming.rebind("Case", this);
      System.out.println("Starting server...");    }
   }
```

Upload the stub to registry and bind it to a name/string

Publish the object (start listening for clients)

An object of the class implementing the remote interface

RmiServer (in parts) – constructor

```
public class RmiServer implements ServerInterface
{
   public static void main(String[] args) throws RemoteException
   {
      Registry reg = LocateRegistry.createRegistry(1099);
      RmiServer server = new RmiServer();
   }
   public RmiServer() throws RemoteException
   {
      ServerInterface stub = (ServerInterface)UnicastRemoteObject
            .exportObject(this, 0);

      Naming.rebind("Case", stub);
      System.out.println("Starting server...");    }
   }
```

Upload the stub to registry and bind it to a name/string

Publish the object (start listening for clients)

A stub to the object of the class implementing the remote interface

Create and publish the remote object

- Method 1
  - The “server” `extends UnicastRemoteObject`
- Method 2
  - The “server” calls method

```
UnicastRemoteObject.exportObject(server, 0);
```

Publish remote object – extending

```
public class RmiTaskServer extends UnicastRemoteObject
                                implements RemoteTaskList
{
   // ...

   public RmiTaskServer() throws RemoteException
   {
      // ...
      super();
      Naming.rebind("tasks", this);
   }

   // ...
```

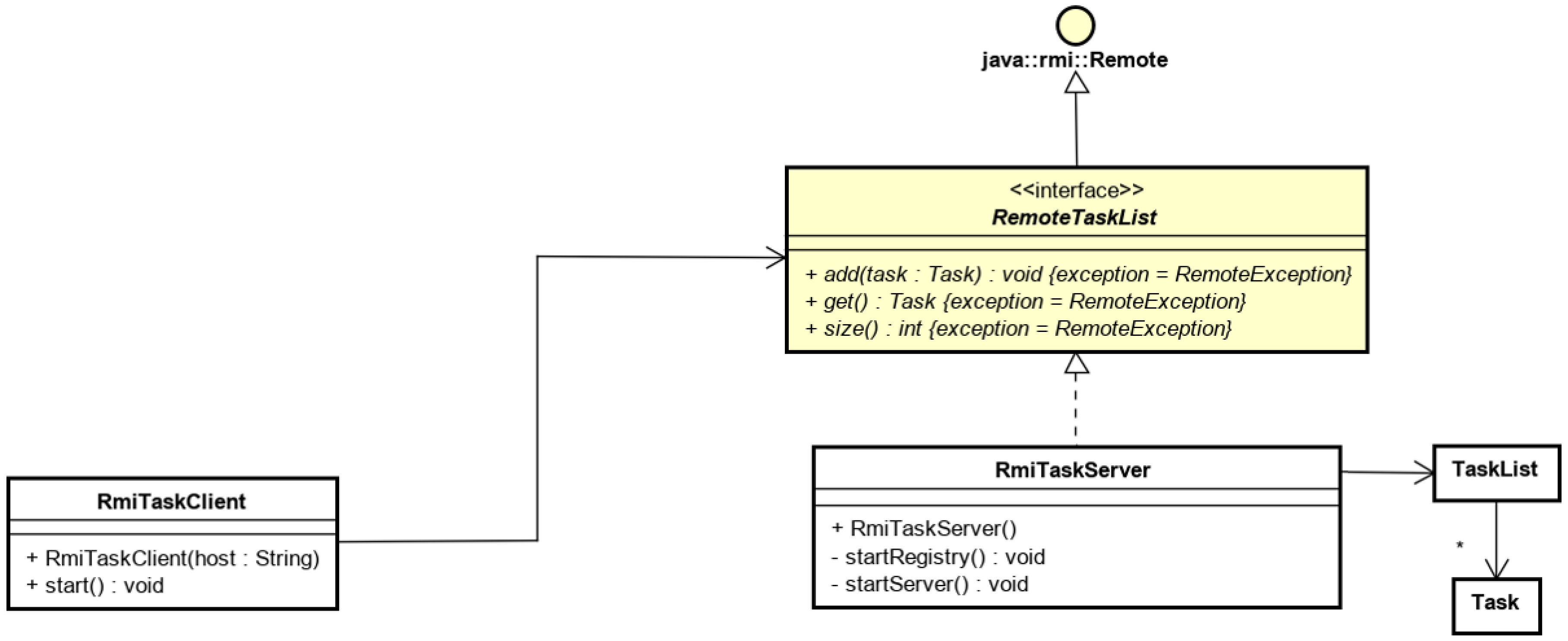
Publish remote object – without extending

```
public class RmiTaskServer implements RemoteTaskList
{
   // ...

   public RmiTaskServer() throws RemoteException, ...
   {
      // ...
      RemoteTaskList stub = (RemoteTaskList)
            UnicastRemoteObject.exportObject(this, 0);
      Naming.rebind("tasks", stub);
   }

   // ...
```

Publish remote object – without extending

```
public class RmiTaskServer implements RemoteTaskList
{
   // ...

   public RmiTaskServer() throws RemoteException, ...
   {
      // ...
      UnicastRemoteObject.exportObject(this, 0);
      Naming.rebind("tasks", this);
   }

   // ...
```

What about security?


java::rmi::Remote
<<interface>>
RemoteTaskList
+ add(task : Task) : void {exception = RemoteException}
+ get() : Task {exception = RemoteException}
+ size() : int {exception = RemoteException}
RmiTaskClient
+ RmiTaskClient(host : String)
+ start() : void
RmiTaskServer
+ RmiTaskServer()
- startRegistry() : void
- startServer() : void
TaskList
*
Task

Security – main method

```
public class Client
{
  public static void main(String[] args) throws Exception
  {
      if (System.getSecurityManager() == null)
      {
         System.setSecurityManager(new SecurityManager());
      }
      RmiTaskClient client = new RmiTaskClient();
      client.start();
  }
}
```

```
java.security.AccessControlException: access denied
("java.net.SocketPermission" "127.0.0.1:1099" "connect,resolve")
       at
java.base/java.security.AccessControlContext.checkPermission(AccessCo
ntrolContext.java:472)
...
```

Security

StartClient.bat

```
java  -Djava.security.policy=rmi.policy Client

pause
```

rmi.policy

```
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

all.policy

```
grant {
   permission java.security.AllPermission;
};
```

Security – and dynamic download

StartClient.bat

```
java  -Djava.rmi.server.codebase=http://ict-engineering.dk/class/
      -Djava.security.policy=rmi.policy Client
pause
```

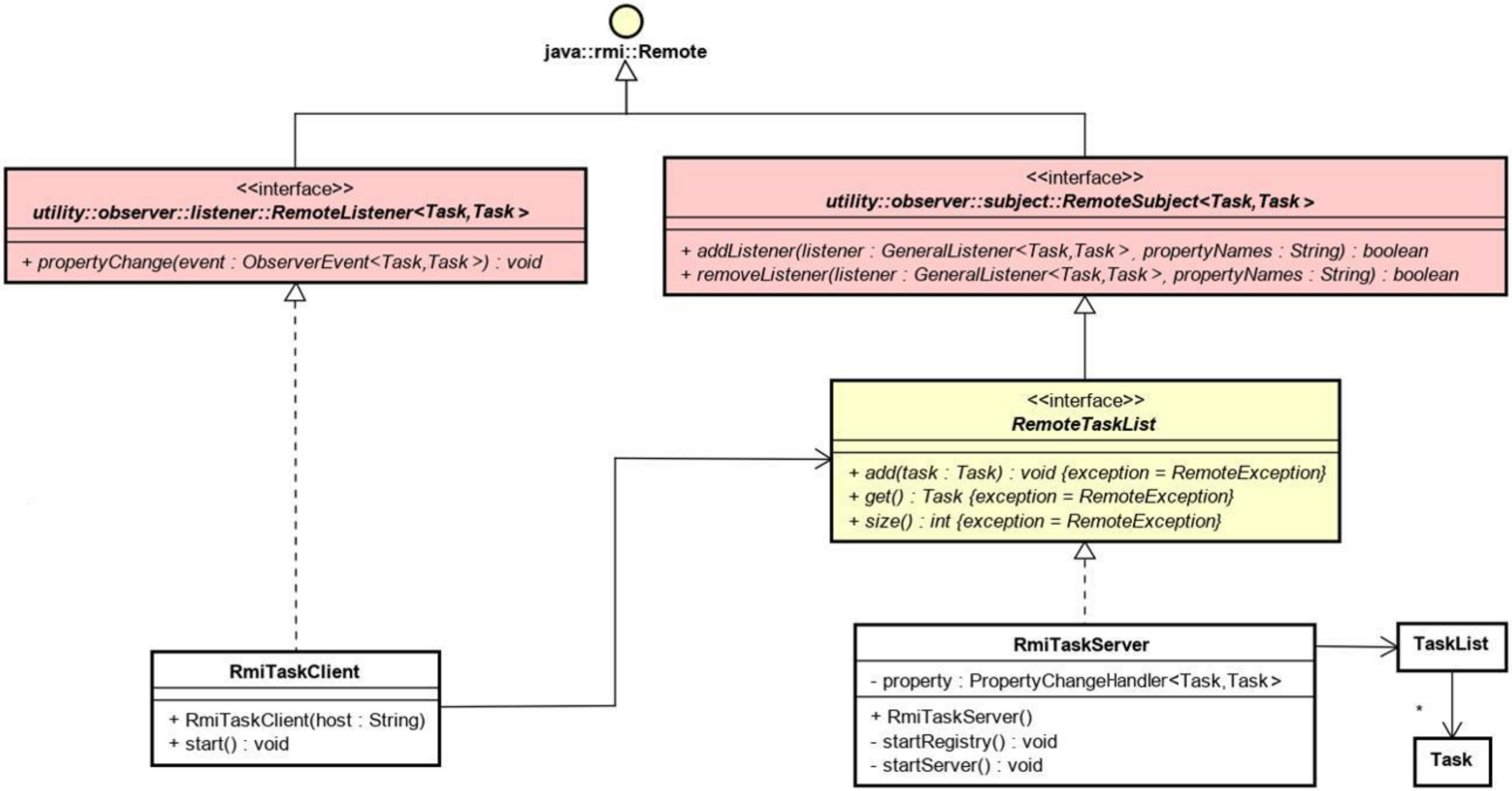
rmi.policy

```
grant {
   permission java.net.SocketPermission "*:1024-65535", "connect,accept";
   permission java.net.SocketPermission "*:80", "connect";
};
```

all.policy

```
grant {
  permission java.security.AllPermission;
};
```

What about Observer? (MyObserver-1.4.jar)



java::rmi::Remote
<<interface>>
utility::observer::listener::RemoteListener<Task,Task >
+ propertyChange(event : ObserverEvent<Task,Task >) : void
<<interface>>
utility::observer::subject::RemoteSubject<Task,Task >
+ addListener(listener : GeneralListener<Task,Task >, propertyNames : String) : boolean
+ removeListener(listener : GeneralListener<Task,Task >, propertyNames : String) : boolean
<<interface>>
RemoteTaskList
+ add(task : Task) : void {exception = RemoteException}
+ get() : Task {exception = RemoteException}
+ size() : int {exception = RemoteException}
RmiTaskClient
+ RmiTaskClient(host : String)
+ start() : void
RmiTaskServer
- property : PropertyChangeHandler<Task,Task >
+ RmiTaskServer()
- startRegistry() : void
- startServer() : void
TaskList
*
Task

Interface

```
package model;

import utility.observer.subject.RemoteSubject;

import java.rmi.RemoteException;

public interface RemoteTaskList extends RemoteSubject<Task, Task>
{
  void add(Task task) throws RemoteException;
  Task get() throws RemoteException;
  int size() throws RemoteException;
}
```

Server side

```
public class RmiTaskServer implements RemoteTaskList
{
  private TaskList model;
  private PropertyChangeHandler<Task, Task> property;

  public RmiTaskServer()
  {
    this.property = new PropertyChangeHandler<>(this, true);
    //...
    UnicastRemoteObject.exportObject(this, 0);
    Naming.rebind("TaskList", this);
  }
  @Override public void add(Task task)
  {
    model.add(task);
    property.firePropertyChange("ADD", null, task);
  }
  @Override
  public boolean addListener(GeneralListener<Task,Task> listener,
      String... propertyNames) throws RemoteException
  {
    return property.addListener(listener, propertyNames);
  }
  //...
}
```

Client side

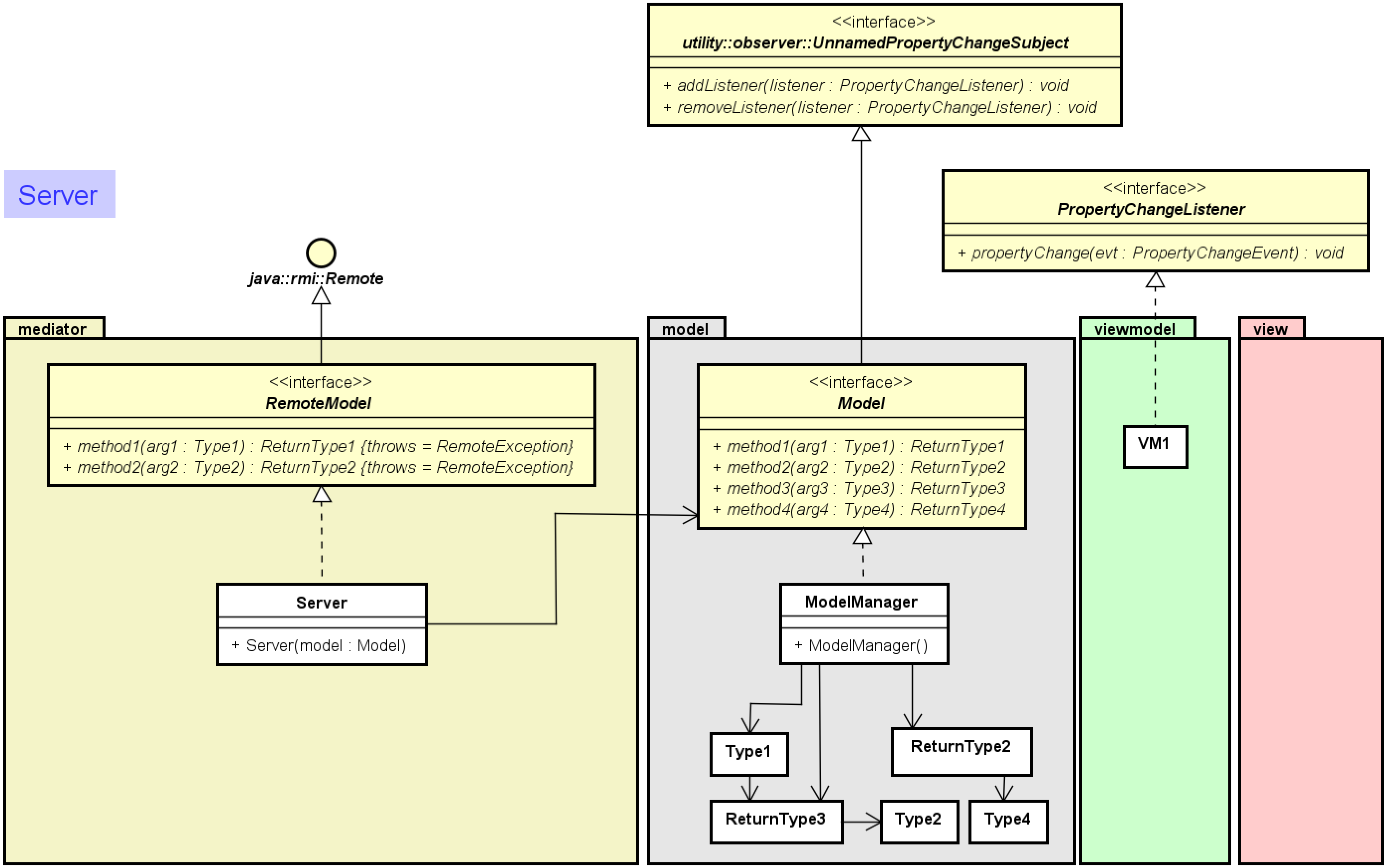
```java
public class RmiTaskClient implements RemoteListener<Task, Task>
{
  private RemoteTaskList remoteModel;

  public RmiTaskClient(String host) throws Exception
  {
     remoteModel = (RemoteTaskList)Naming.lookup("rmi://" + host
                                        + ":1099/TaskList");
     UnicastRemoteObject.exportObject(this, 0);
     remoteModel.addListener(this);
  }
  //...

  @Override
  public void propertyChange(ObserverEvent<Task, Task> event)
  {
    System.out.println("Server added: " + event.getValue1());
  }
}
```

What about MVVM?

- Original (local) model do not depend on client/server technology
  - Socket classes could be replaced by RMI interface and class
  - Client ModelManager delegates to another class to make a registry lookup, export object if remote listener and to handle possible RemoteExceptions
- Local and remote model could be different
  - Server may have methods unavailable by clients
- No methods have to handle RemoteException's if called locally
- Mediator for Sockets could be replaced by a Mediator for RMI

MVVM + Mediator (Server)

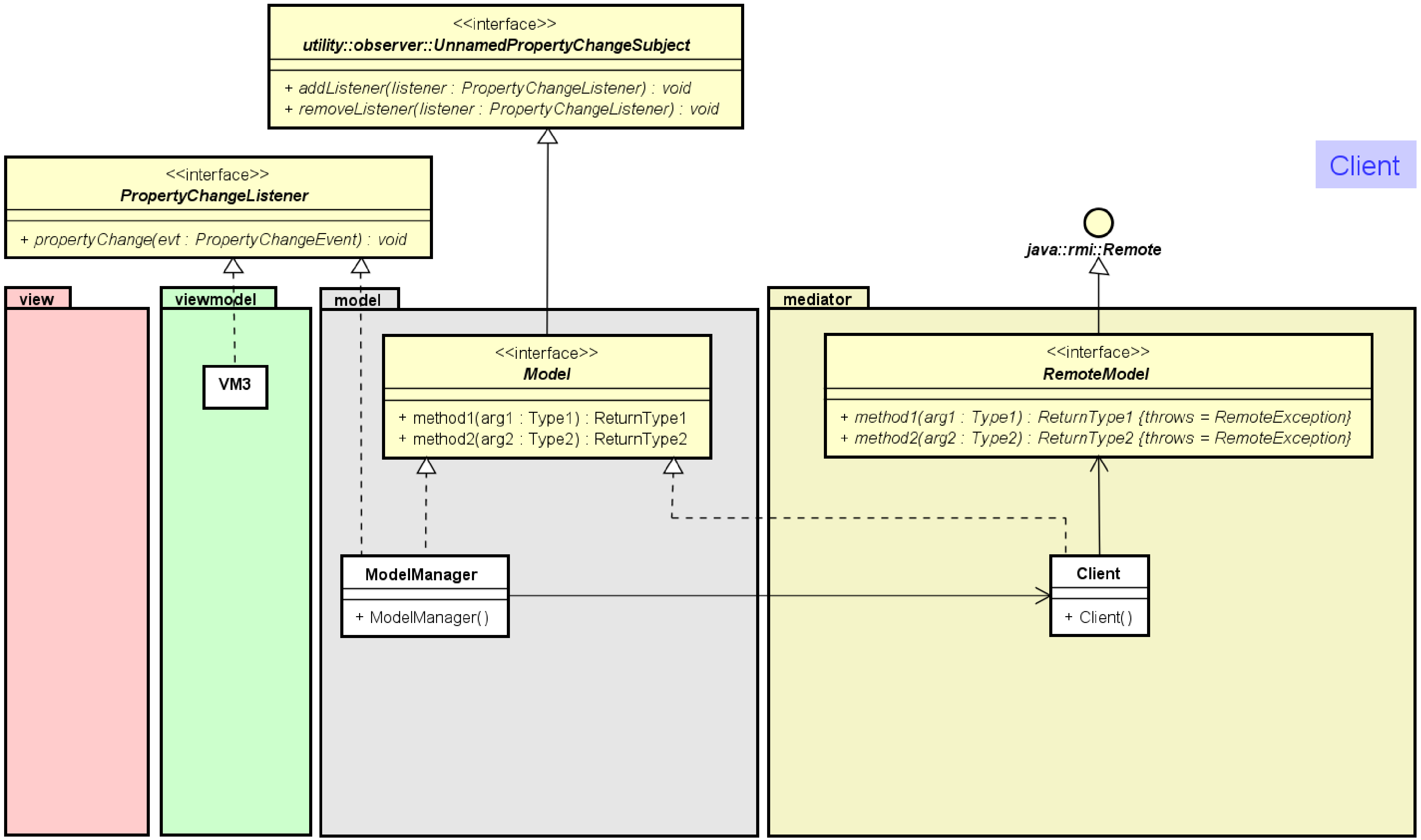
Server



<<interface>>
utility::observer::UnnamedPropertyChangeSubject
+ addListener(listener : PropertyChangeListener) : void
+ removeListener(listener : PropertyChangeListener) : void
<<interface>>
PropertyChangeListener
+ propertyChange(evt : PropertyChangeEvent) : void
java::rmi::Remote
mediator
<<interface>>
RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Server
+ Server(model : Model)
model
<<interface>>
Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
+ method3(arg3 : Type3) : ReturnType3
+ method4(arg4 : Type4) : ReturnType4
ModelManager
+ ModelManager()
Type1
ReturnType2
ReturnType3
Type2
Type4
viewmodel
VM1
view

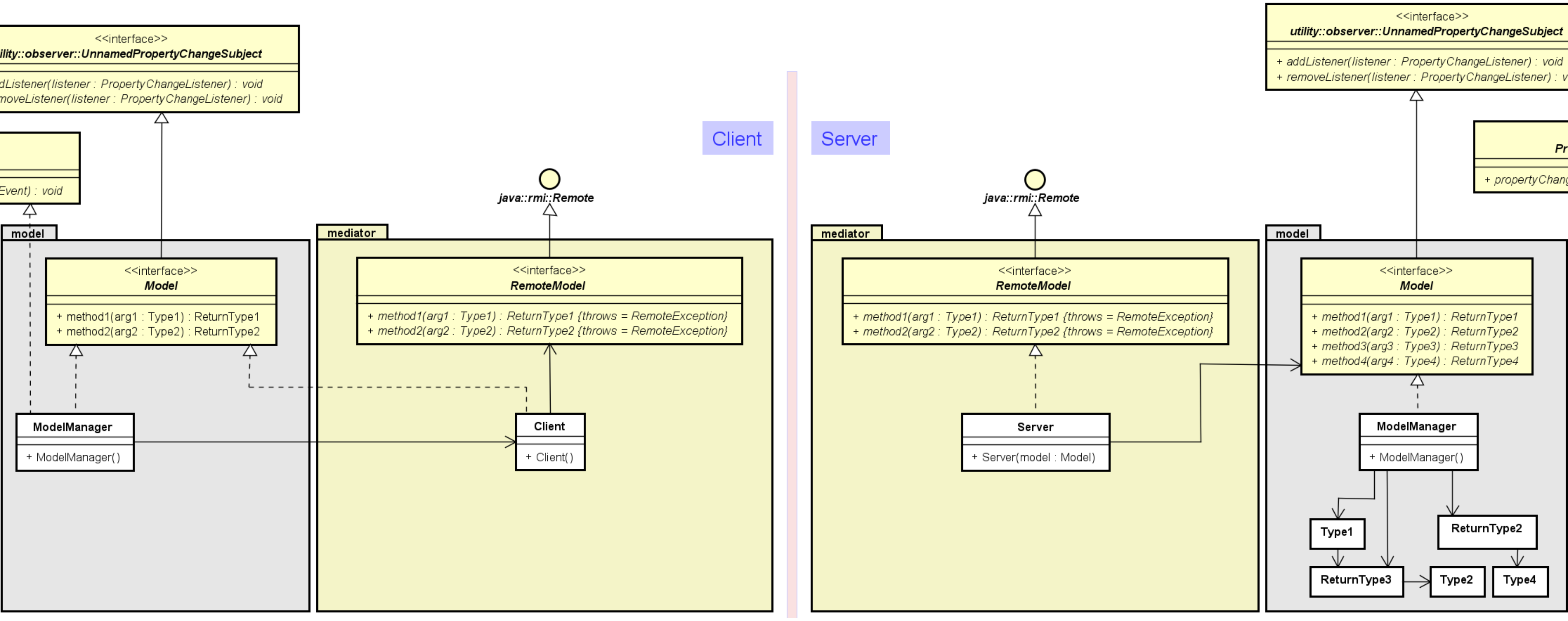
MVVM + Mediator (Client)

Client

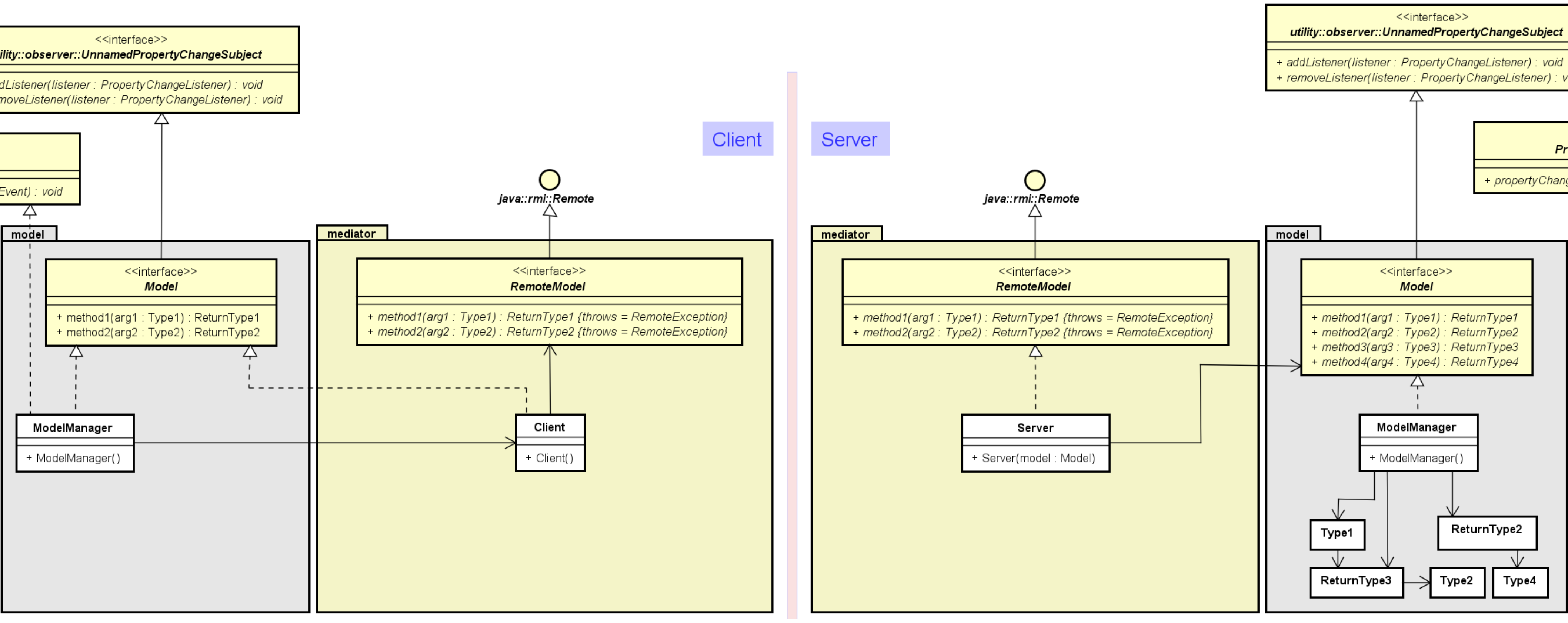


<<interface>>
utility::observer::UnnamedPropertyChangeSubject
+ addListener(listener : PropertyChangeListener) : void
+ removeListener(listener : PropertyChangeListener) : void
<<interface>>
PropertyChangeListener
+ propertyChange(evt : PropertyChangeEvent) : void
java::rmi::Remote
view
viewmodel
VM3
model
<<interface>>
Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
ModelManager
+ ModelManager()
mediator
<<interface>>
RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Client
+ Client()

Mediator (Server and Client)

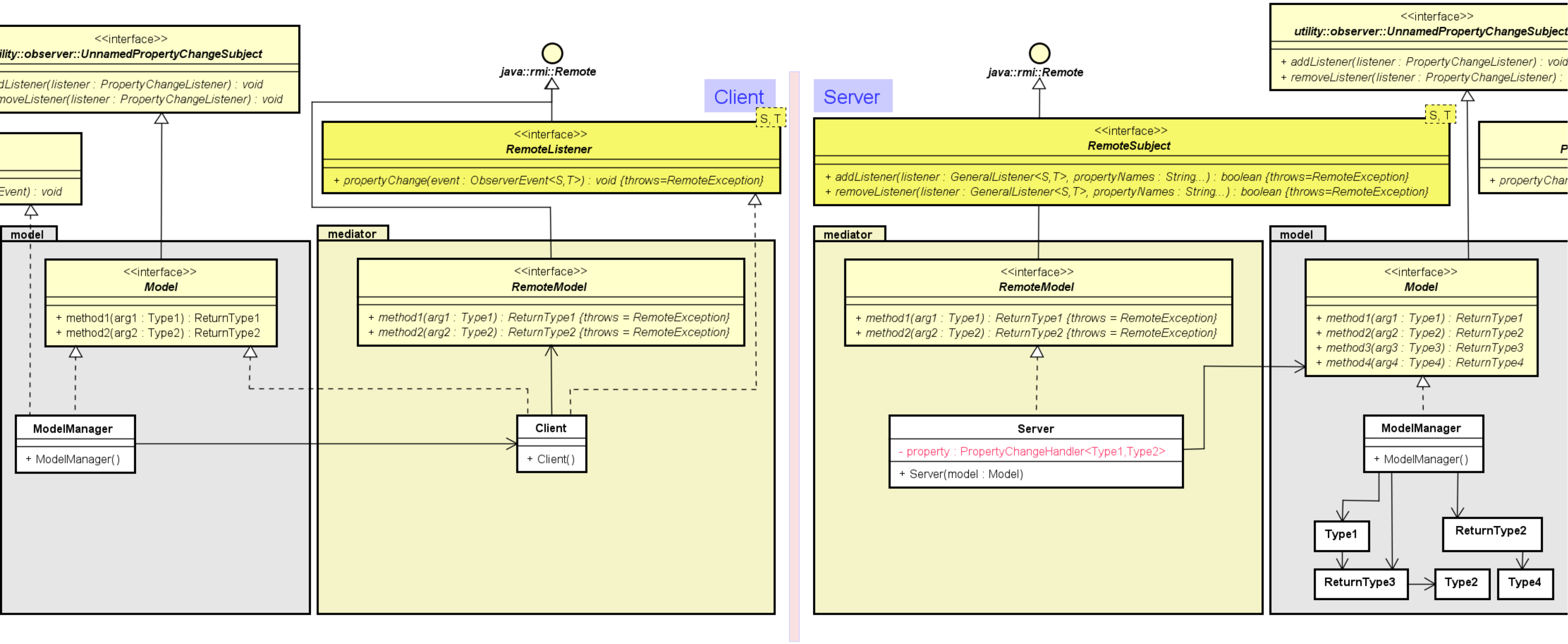

Client
Server
<<interface>>
utility::observer::UnnamedPropertyChangeSubject
+ addListener(listener : PropertyChangeListener) : void
+ removeListener(listener : PropertyChangeListener) : void
Event) : void
model
mediator
java::rmi::Remote
<<interface>>
Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
+ method3(arg3 : Type3) : ReturnType3
+ method4(arg4 : Type4) : ReturnType4
<<interface>>
RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
ModelManager
+ ModelManager()
Client
+ Client()
Server
+ Server(model : Model)
Type1
ReturnType2
ReturnType3
Type2
Type4
+ propertyChang

What about MVVM and Observer?



<<interface>>
ility::observer::UnnamedPropertyChangeSubject
dListener(listener : PropertyChangeListener) : void
moveListener(listener : PropertyChangeListener) : void
Event) : void
model
<<interface>>
Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
ModelManager
+ ModelManager()
mediator
java::rmi::Remote
<<interface>>
RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Client
+ Client()
Client
Server
mediator
java::rmi::Remote
<<interface>>
RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Server
+ Server(model : Model)
<<interface>>
utility::observer::UnnamedPropertyChangeSubject
+ addListener(listener : PropertyChangeListener) : void
+ removeListener(listener : PropertyChangeListener) : v
Pr
+ propertyChang
model
<<interface>>
Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
+ method3(arg3 : Type3) : ReturnType3
+ method4(arg4 : Type4) : ReturnType4
ModelManager
+ ModelManager()
Type1
ReturnType2
ReturnType3
Type2
Type4

What about MVVM and Observer?



<<interface>> utility::observer::UnnamedPropertyChangeSubject
addListener(listener : PropertyChangeListener) : void
removeListener(listener : PropertyChangeListener) : void
Event) : void
model
<<interface>> Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
ModelManager
+ ModelManager()
java::rmi::Remote
Client
S, T
<<interface>> RemoteListener
+ propertyChange(event : ObserverEvent<S,T>) : void {throws=RemoteException}
mediator
<<interface>> RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Client
+ Client()
Server
java::rmi::Remote
<<interface>> RemoteSubject
+ addListener(listener : GeneralListener<S,T>, propertyNames : String...) : boolean {throws=RemoteException}
+ removeListener(listener : GeneralListener<S,T>, propertyNames : String...) : boolean {throws=RemoteException}
S, T
mediator
<<interface>> RemoteModel
+ method1(arg1 : Type1) : ReturnType1 {throws = RemoteException}
+ method2(arg2 : Type2) : ReturnType2 {throws = RemoteException}
Server
- property : PropertyChangeHandler<Type1,Type2>
+ Server(model : Model)
<<interface>> utility::observer::UnnamedPropertyChangeSubject
+ addListener(listener : PropertyChangeListener) : void
+ removeListener(listener : PropertyChangeListener) :
P
+ propertyChar
model
<<interface>> Model
+ method1(arg1 : Type1) : ReturnType1
+ method2(arg2 : Type2) : ReturnType2
+ method3(arg3 : Type3) : ReturnType3
+ method4(arg4 : Type4) : ReturnType4
ModelManager
+ ModelManager()
Type1
ReturnType2
ReturnType3
Type2
Type4

RMI and MVVM (Server) –Two design principles

1. Remote and local server model:
   Maintain the (local) server model, create a Remote model interface with similar methods and an RMI server class implementing the interface. The RMI server class has an association to the local server Model

2. Server model implements the remote interface:
   a) change existing model interface to be Remote, or
   b) Implement two interfaces
   i.e. let the ModelManager act as an RMI server too

RMI and MVVM (Client) –Two design principles

1. ModelManager implements the local Model and delegates to an RMI client class having association to a remote model (interface). The RMI client class either implements the local Model too or implements another interface.

2. ModelManager has an association to the remote Model (interface) – omitting the need for the RMI client class

Difference between socket and RMI systems

- Sockets
  - TCP or UDP
  - Protocol for communication
  - Define classes to send/receive
  - Create threads
  - Open and close connection
  - Full control over classes
  - Full control over ports
  - Full control over security
  - More code, more control

- RMI
  - TCP Socket based (but hidden)
  - “Protocol” is a method call
  - Methods work as send/receive
  - Using threads (hidden from us)
  - No manual clean up
  - Have to follow RMI rules
  - Less control over ports
  - Security manager
  - Less code, less control