

## Implementierung III

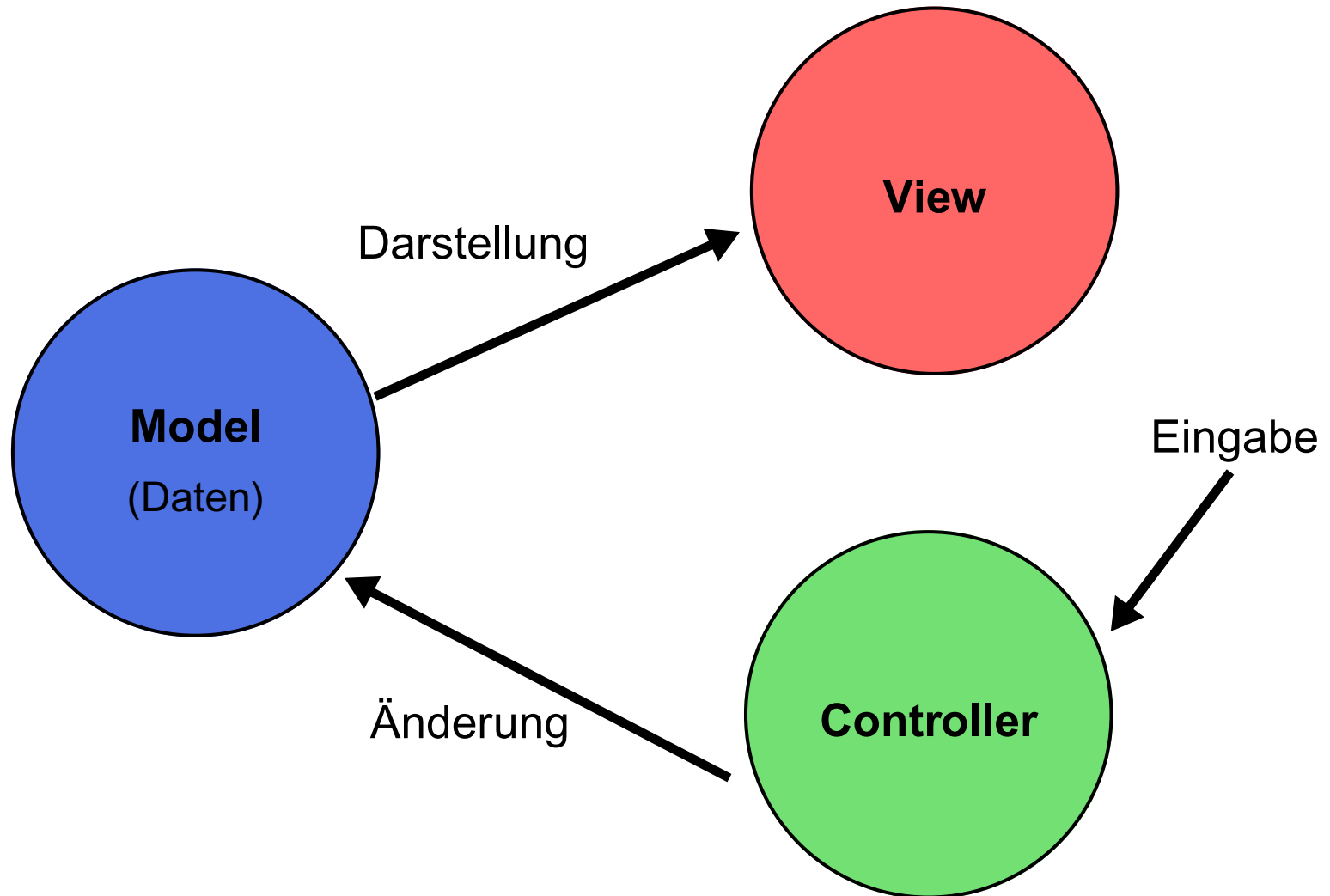
GUI und  
Verhalten (Teil 2)

## Implementierung IV

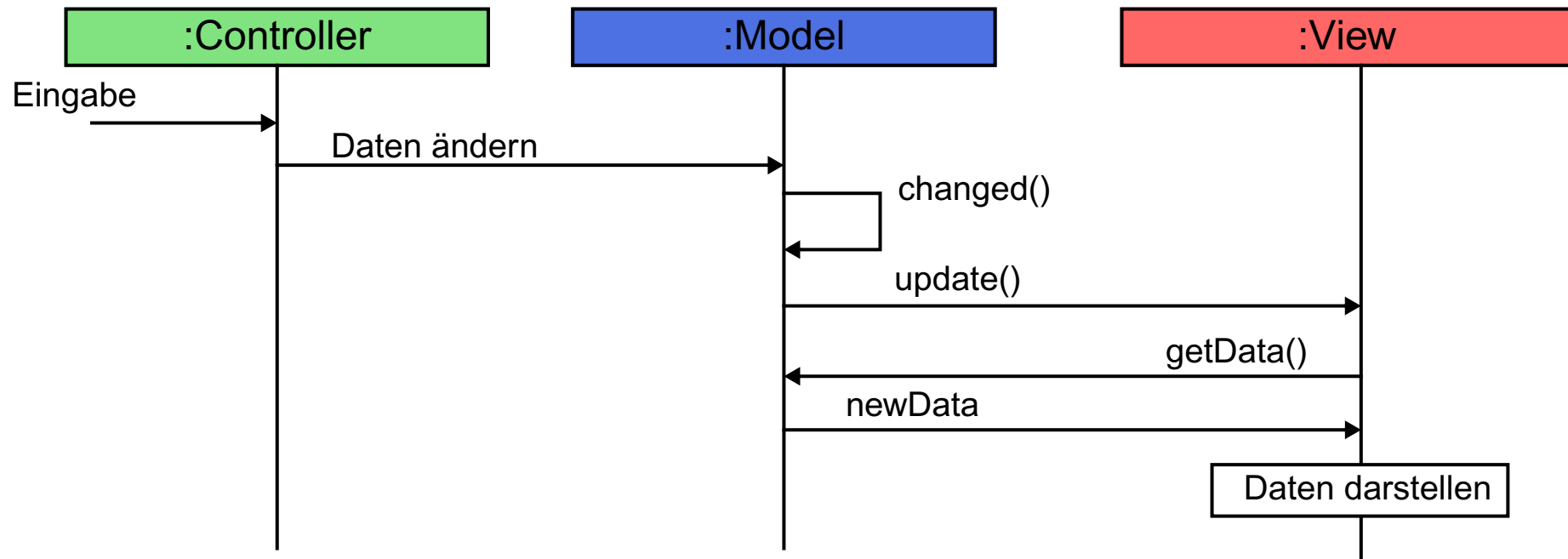
Schnittstelle zur Umgebung

Integration und Test

## Architektur der Benutzerschnittstelle



## M-V-C Kommunikationsschema

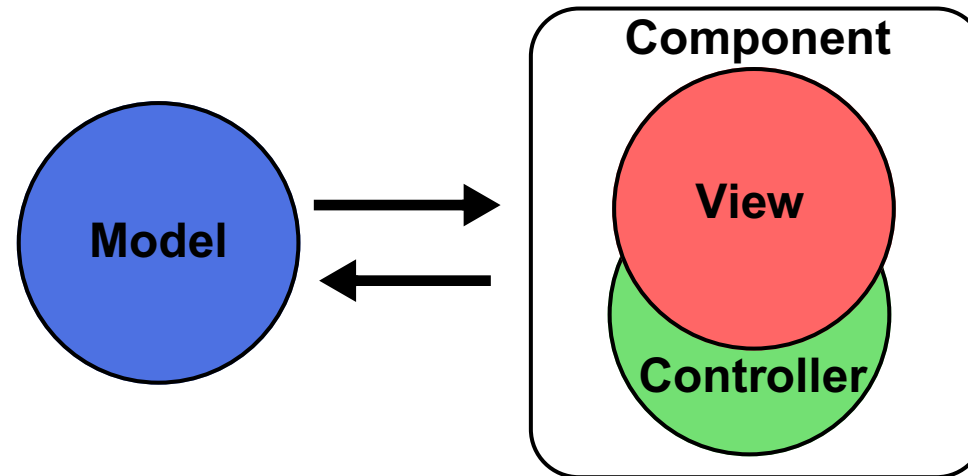


• **Observer-Pattern:**

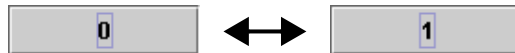
Observable: **Model**

Observer: **View**

## JFC/Swing-Realisierung



### Beispiel:



```
public class MyButton extends
    JButton implements Observer {

    public MyButton(MyModel model) {
        setText(model.getState());
        model.addObserver(this);
        addActionListener(
            new ActionListener() {
                public void actionPerformed(
                    ActionEvent e) {
                    model.toggleState();
                }
            });
    }
}
```

```
public void update(Observable o,
    Object arg) {
    setText(model.getState());
}
```

```
public class MyModel extends Observable {
    int state = 1;
    public void toggleState() {
        state = 1 - state;
        setChanged(); notifyObservers();
    }
}
```

## Thread Handling in JFC/Swing

- nachdem Swing Komponente (*Component*) dargestellt (*realized*) wurde gilt:

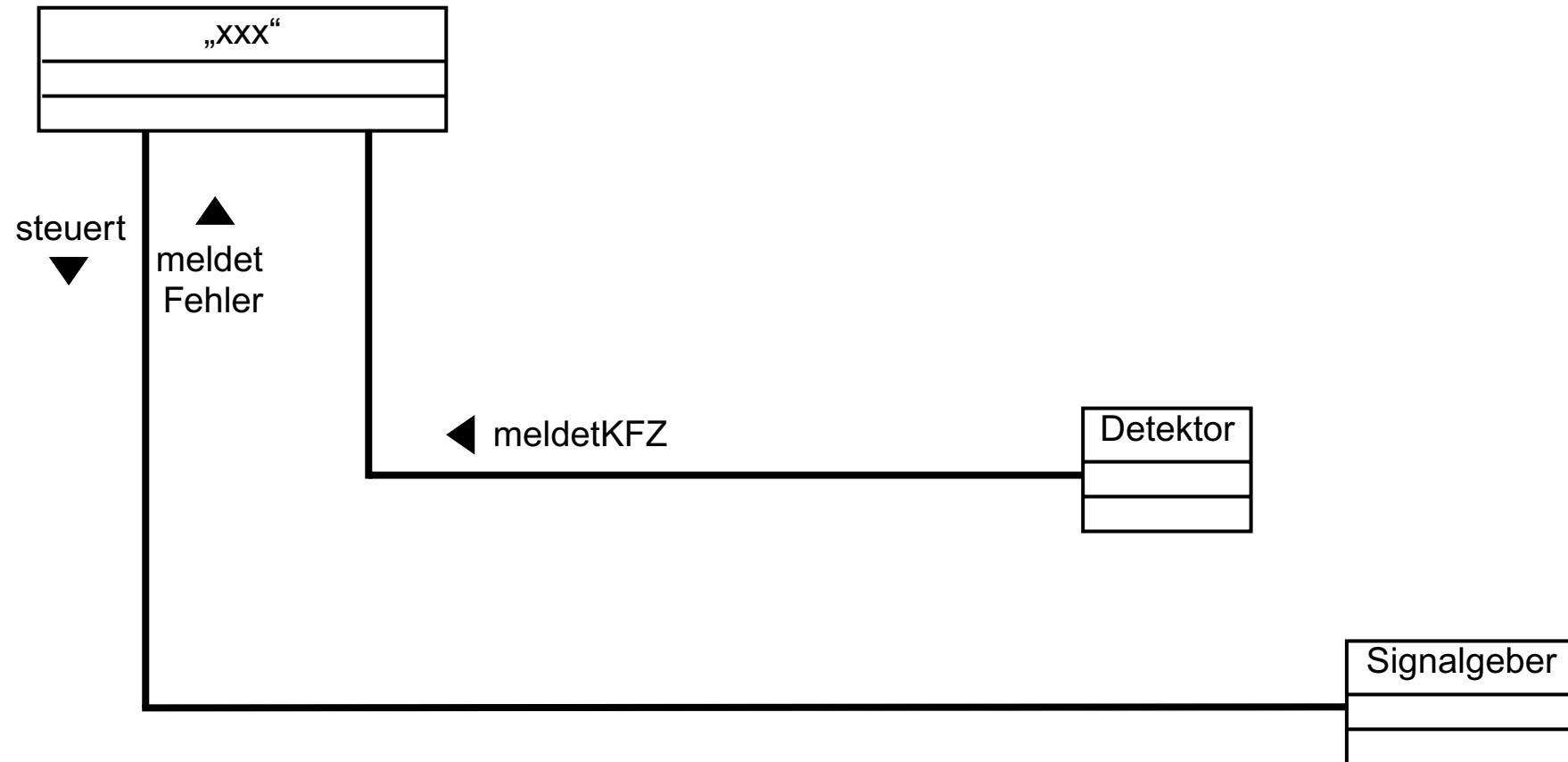
**Sämtlicher Code, welcher die Komponente betrifft, oder von deren Zustand abhängt darf nur im “*Event-Dispatching-Thread*” ausgeführt werden!**

- **Grund:** die meisten Swing-Methoden sind **nicht „thread safe“**
  - nur dann unproblematisch, wenn alle Methoden in **einem** Thread ausgeführt werden
- **Ausnahmen:**
  - `repaint()`, `revalidate()`
  - `add...Listener()`, `remove...Listener()`
  - Methoden mit “This method is thread safe, although most Swing methods are not.”
- **Aufruf von Code im “Event-Dispatching-Thread”:**
  - `SwingUtilities.invokeLaterAndWait(Runnable doRun)`  
blockierend (kehrt nach Ausführung des Codes zurück)
  - `SwingUtilities.invokeLaterLater(Runnable doRun)`  
nicht-blockierend (ggü. `invokeAndWait()` vorzuziehen!)

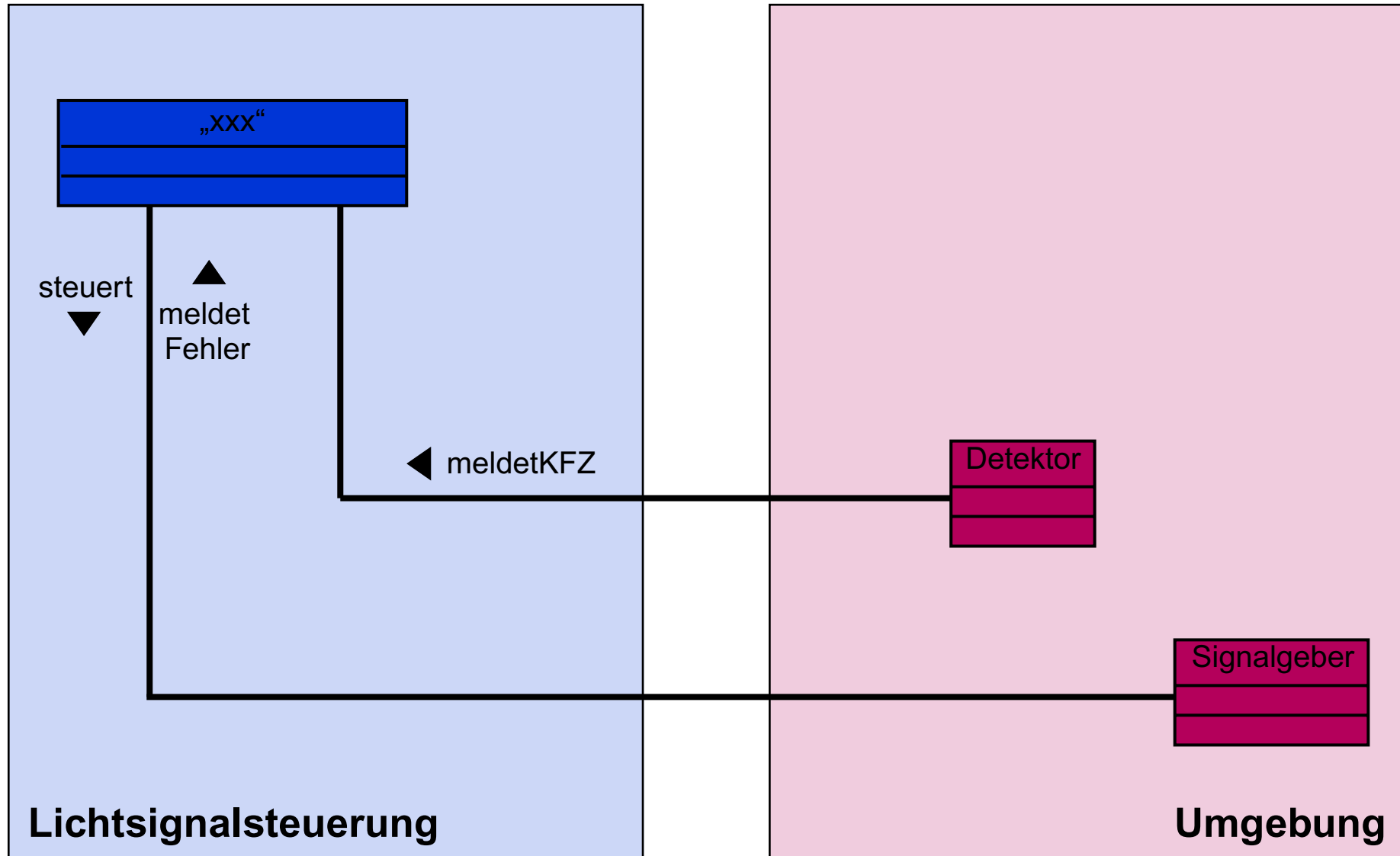
## Aufgaben

- Implementierung der GUI
  - **Trennung** von Model und Component!
  - GUI-Code nur im “Event-Dispatching-Thread”!
- GUI soll beinhalten:
  - Darstellung des Verkehrsdurchsatzes (gesamt, entlang der „Grünen Welle“)
  - Anzeige der Fehlerzustände
  - Möglichkeit GUI und Steuerung „sauber“ zu beenden
- Implementierung des Gesamtverhaltens
  - Realisierung der kommunizierenden Zustandsautomaten (benutzerdefinierte Events)
  - Test!
- Abgabe:
  - Java-Sourcen **aller** Klassen (inkl. GUI)
  - Javadoc (vollständig und ausführlich/sinnvoll)
  - mind. 10 protokollierte Testläufe (Nachweis der Korrektheit an Hand der Sequenzdiagramme)

## Realisierung des Systems auf Modellebene

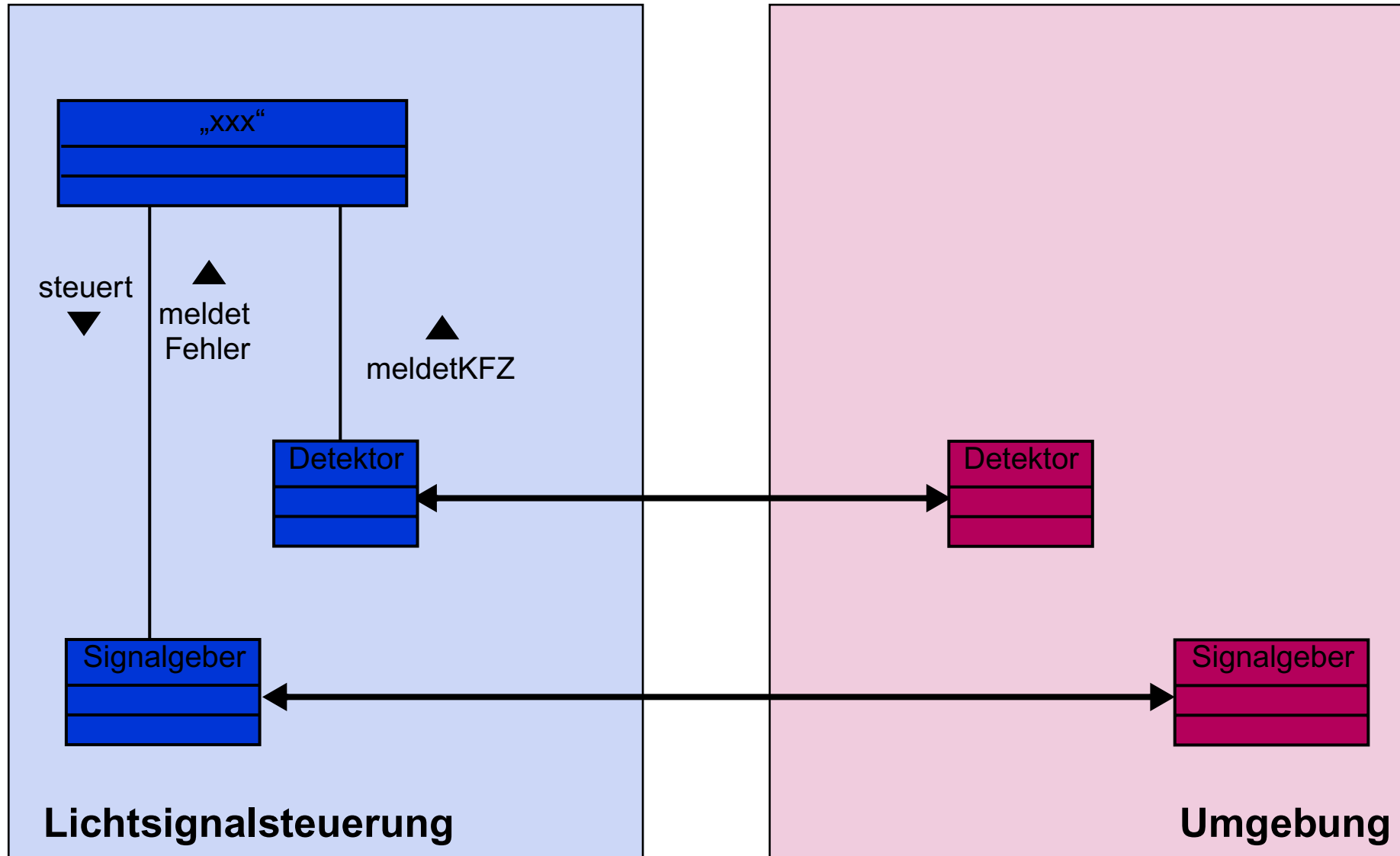


## Problem: Systemgrenzen



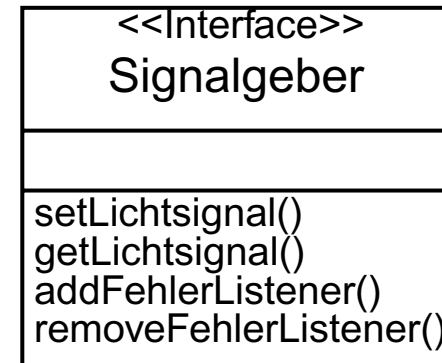
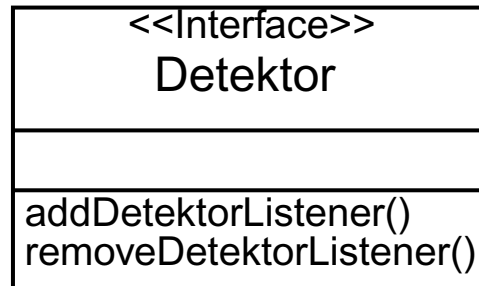


## Lösung: Stellvertreter-Klassen



## Technische Realisierung: Java-RMI

- Definition der Funktionalitäten: **Remote-Interface**

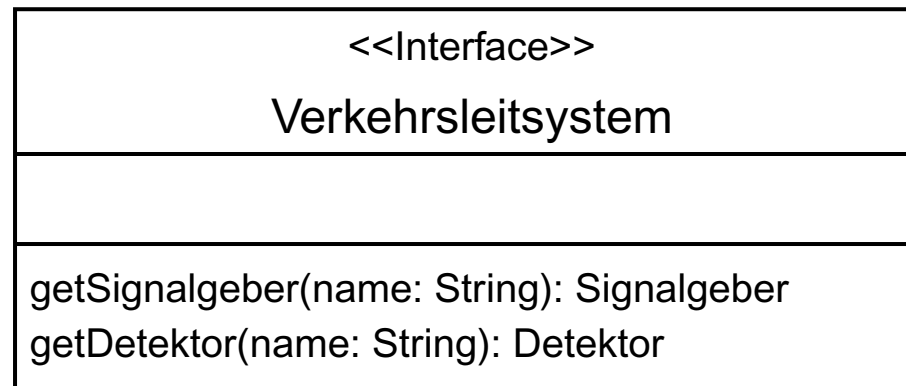


- Implementierung des Interfaces ...  
–z.B. `class DetektorImpl implements Detektor { /* ... */ }`
- ... und Instanziierung auf Server-Seite: **Remote-Objekt**  
–z.B. `detImpl1 = new DetektorImpl();`
- Aufruf der Remote-Methoden durch **Remote-Referenzen** auf Client-Seite  
(z.B. `Detektor det1 = /* ... */; det1.addDetektorListener();`)



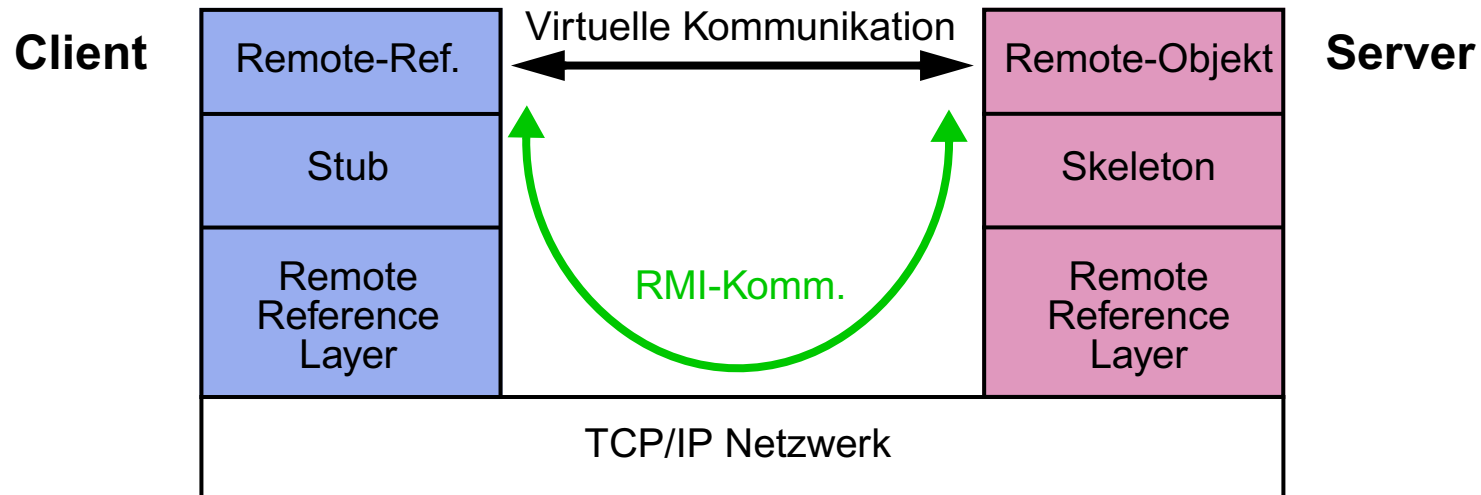
## Java-RMI: Erhalten von Remote-Referenzen

- Möglichkeit 1: Naming-Service: **RMI-Registry**
  - Registry liegt an einer allgemein bekannten Stelle (z.B. Port 20010)
  - Remote-Objekte registrieren sich bei der Registry unter einem „sinnvollen“ Namen
  - Nachteil: aufwändig bei vielen benötigten Remote-Referenzen
- Möglichkeit 2: „**Bootstrapping**“-Objekt
  - initiale Referenz kommt von Registry
  - alle anderen Referenzen kommen als Rückgabewerte vom „Bootstrapping“-Objekt
- „Bootstrapping“-Klasse für das Praktikum:



- Name des „Bootstrapping“-Objektes in der Registry: **verkehrsleitsystem**

## Java-RMI: Kommunikationsschichten



- Stub und Skeleton implementieren **dasselbe** Remote-Interface wie das Remote-Objekt
- Stub- und Skeleton-Klassen werden mit `rmic` (JDK-Tool) aus der **Implementierung** des Remote-Objekts generiert

## Java-RMI: Übertragung von Byte-Code

- Stub- und Skeleton-Klassen liegen **immer** auf der Seite des Servers
  - **nur** Remote-Interfaces sind zur Compile-Zeit auf Client-Seite bekannt!
- daher: Übertragung des Byte-Codes der Stub-Klasse zur Laufzeit notwendig
  - Möglichkeit 1: Proprietäres Protokoll
  - Möglichkeit 2: HTTP, FTP, ... (Standard-Protokolle)
- Java-RMI nutzt Möglichkeit 2
  - Web-/FTP-Server benötigt!
  - **ClassFileServer** (spezieller Web-Server in Java implementiert) kann genutzt werden, damit Unabhängigkeit von „echtem“ Web-Server


- **Annotieren** der Klassen, damit Client den Ort der Class-Binaries kennt:

```
System.setProperty("java.rmi.server.codebase", "http://myhost:20000/");
```

 **wichtig!**

- sonst sucht Client nur im (lokalen) CLASSPATH

## Sicherheit von RMI?

- RMI erfordert „Download“ von Bytecode!
  - könnte potenziell gefährlichen Code enthalten 
  - also: kein RMI ohne Sicherheitskonzept!

## Entwicklung des Sicherheitskonzeptes in Java

- JDK 1.0
  - **sandbox model**
  - lokaler Code: „full access“
  - remote Code: „restricted access“
- JDK 1.1
  - sandbox model
  - lokaler Code + *trusted Code*: „full access“
  - (untrusted) remote Code: „restricted access“
- JDK 1.2
  - kein Konzept von trusted Code mehr
  - fine-grain access control (**policy-based**)

## Das Policy-File (“java.policy”)

- Sicherheitskonzept wird vom `SecurityManager` realisiert, der Policy-File liest
- der Einfachheit halber („full access“) oder wenn man Code wirklich vertrauen kann:  
**gefährlich** ⚡

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

- für endgültiges Produkt:  
**minimaler** Satz an Rechten, z. B.:

```
grant {  
    permission java.net.SocketPermission “myhost:20000-20099”, “connect, accept”;  
    permission java.io.FilePermission “/home/test/-”, “read”;  
    permission java.util.PropertyPermission “user.dir”, “read”;  
};
```

## Beispiel: „simples Steuerungssystem“

```
/* ... */  
// Import remote interfaces  
import environment.rmi.*;  
  
public class ControlSystem {  
  
    public static void main(String[] args) {  
  
        // Install security manager  
        System.setProperty("java.security.policy", "./java.policy");  
        if(System.getSecurityManager() == null) {  
            System.setSecurityManager(new SecurityManager()); }  
  
        // Get classServerPort  
        int portClassServer = /* ... */;  
  
        // Install class file server  
        try {  
            String classpath = System.getProperty("user.dir");  
            ClassFileServer cfs =  
                new ClassFileServer(portClassServer, classpath);  
        } catch (Exception exc) { System.exit(0);}  
    }  
}
```



## Beispiel: „simples Steuerungssystem“ (Forts.)

**// Annotate remote classes**

```
String localhost = /* ... */;
```

```
System.setProperty("java.rmi.server.codebase", "http://" +  
    localhost + ":" + portClassServer + "/"); ← wichtig!
```

**// Retrieve object references**

```
try {
```

```
    Verkehrssystem vl = (Verkehrssystem)
```

```
        Naming.lookup("//myhost:20010/Verkehrssystem");
```

```
    Signalgeber sg7_1;
```

```
    sg7_1 = vl.getSignalgeber("Isa7_sg1");
```

```
    sg7_1.setZustand(Signalgeber.GRUEN);
```

```
    } catch(Exception exc) { }
```

```
}
```

## Aufgaben

- Ergänzung der Implementierung: Kopplung zum Simulator
  - Verwendung der Detektoren und Signalgeber des AmpelSimulators

## Aufgaben

- Test mit Hilfe des AmpelSimulators
  - „Grüne Welle“
  - Verkehrsstatistiken (über einen längeren Zeitraum beobachtet)
- Anpassung der Parameter
- Evtl. Modifikation der Strategien
  
- Dokumentation der Testläufe
  - Testprotokolle
  - Übereinstimmung mit den Sequenzdiagrammen
- Abgabe:
  - Java-Sourcen + Javadoc
  - evtl. modifizierte Sequenzdiagramme
  - Testprotokolle (inkl. sinnvoller Dokumentation)

## Wissenschaftliches Experiment zur Qualität der Softwaredokumentation

- Unabhängig von Gruppeneinteilung und bisheriger Arbeit
- Betreuer des Experimentes: Joachim Bayer, bayer@iese.fhg.de
- Durchführung: 23.6.2003, ab 15:30 (bis spätestens 18:30)
- **Anwesenheitspflicht !**

**Rechtzeitig absagen und ggfls. Ersatztermin vereinbaren!**

## Gruppen bei jeweiligen Betreuer/-in:

- Zeitraum: 28.7. - 1.8.
- 10min: Systemvorführung: Start des Systems, GUI, ...
- 30min: „Benchmark“ (während der Vorträge)
- je ca. 5min: Einzelvorträge
  - ⇒  $\sum$  max. 40 min (+ Fragen am Ende) je Gruppe
  - ⇒  $\sum$  max. 3 Stunden für 3 Gruppen
- **Anwesenheitspflicht**