Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Komponentenarchitektur

Software Engineering

Prof. Dr. Bernd Hafenrichter

Implementierung einer Komponentenarchitektur

- Die Implementierung einer Komponentenarchitektur ist abhängig von der verwendeten Laufzeitumgebung.
- Für jede Umgebung muss die Abbildung der Architektur auf die konkrete Programmiersprache definiert werden. Darunter fallen:
  - Definition der Schnittstellen
  - Implementierung der Komponenten
  - Dependency-Management zwischen den Komponenten.
- Dependency-Management: Wer verbindet die Komponenten zu einem lauffähigen Ganzen
  - Das ist die Aufgabe der Konfiguration
  - Die Konfiguration verbindet also zur Laufzeit/Compilezeit eine Schnittstelle mit der zugehörigen Implementierung

Der Konfiguration – Komponenten zu einem Ganzen verbinden

Compiltetime Dependency auf Implementierungsklassen
(Strukturkopplung)
Alle internen Implementierungsdetails sind zur Übersetzungszeit sichtbar

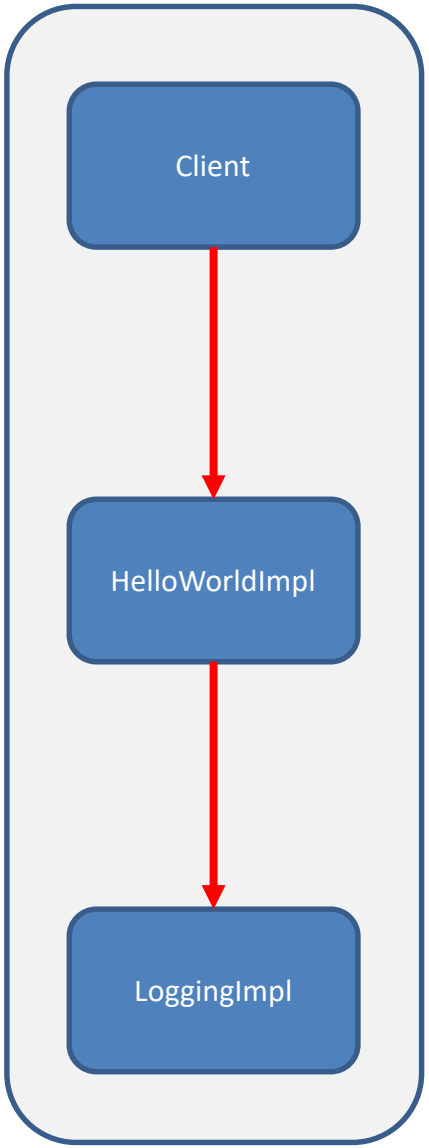
Compiltetime Dependency auf öffentliche API-Artefakte
(Schnittstellenkoppung)
Nur öffentliche Artefakte sind zur Übersetzungszeit sichtbar

Runtime Dependency auf Implementierungsklassen
Nicht sichtbar zur übersetzungszeit.
Können zur Laufzeit dynamisch geladen werden

```
public class Client {

public static void main(String[] args) {
        HelloWorldImpl helloWorld = new HelloWorldImpl();
        helloWorld.sayHelloTo( "the world" );
    }
}
```

```
public class HelloWorldImpl {

private LoggingImpl logging = new LoggingImpl();

public void sayHelloTo(String name ) {
       System.out.println( "Hello world dear " + name );
       logging.writeLogMessage( "Message to the log sub system" +
name );
   }
}
```

```
public class LoggingImpl  {

public void writeLogMessage( String message ) {
      System.out.println( "LoggingImpl: " + message );
   }
}
```



HelloWorld_ComponentArchitecture_Monolith

Umsetzung einer Komponentenarchitektur – Ein erster Versuch

Probleme die enthalten sind:

- Keine Komponentenarchitektur
- Hohe Kopplung der Module
- Keine Austauschbarkeit
- Schlechte Wartbarkeit

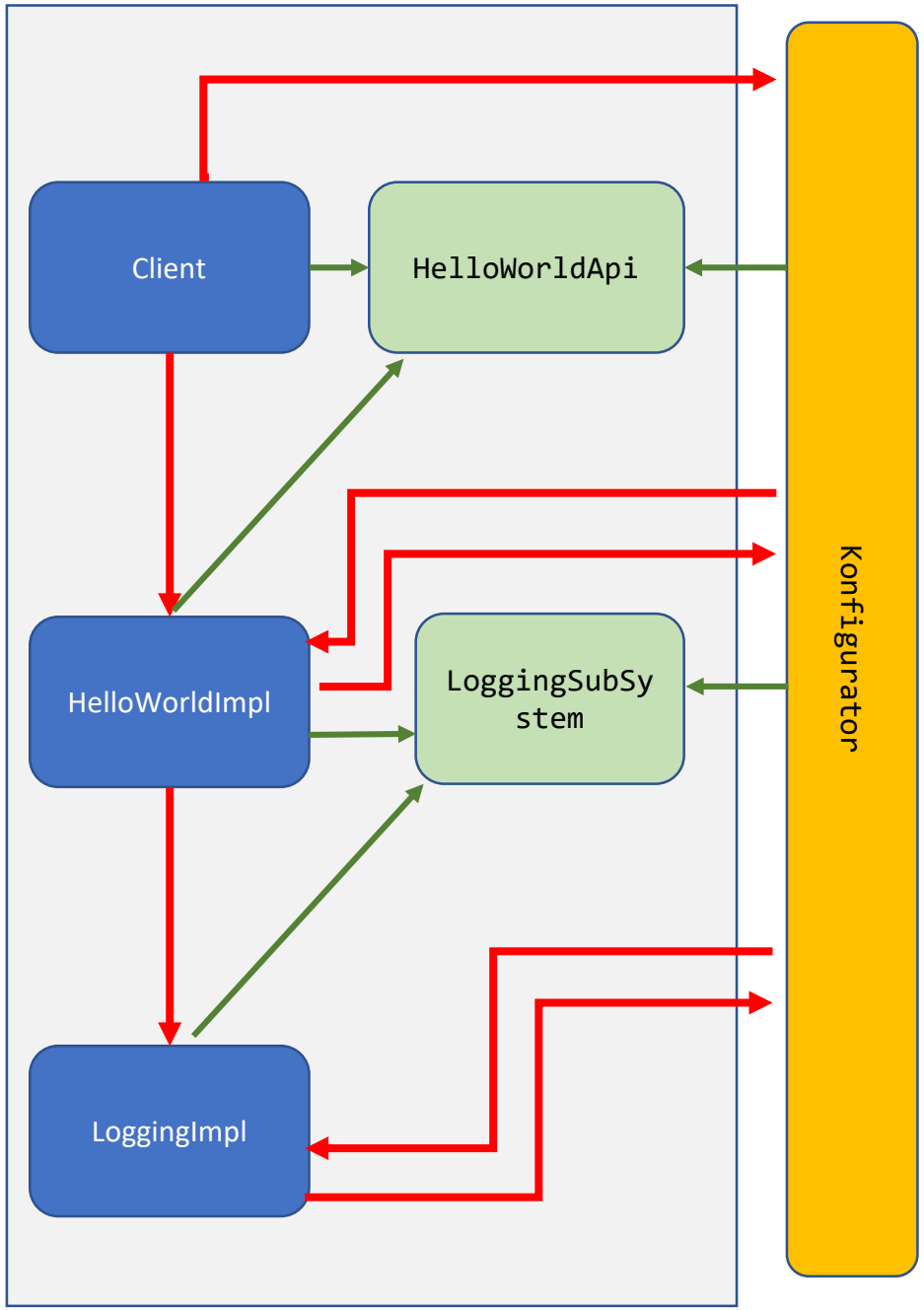
Refakturierung:

- Erweitere das Programm so das die grundlegenden Prinzipien einer Komponenten-Architektur erfüllt werden
- Definiere für jede Komponenten eine API in Form einer Schnittstelle
- Trenne die Client einer Komponente von dessen Implementierung.
- Führe einen Konfigurator ein welche die Komponenten erzeugt und den Clients zugriff über die Schnittstellen gewährt

```
public class KonfigImpl {

    public static final LoggingSubSystem logApi
                                = new
LoggingImpl();

    public static final HelloWorldApi helloApi
                                = new
HelloWorldImpl();
}
```

```
public class Client {
public static void main(String[] args) {
      HelloWorldApi helloWorld =
KonfigImpl.helloApi;
      helloWorld.sayHelloTo( "world" );
   }
}
```

```
public class HelloWorldImpl implements
HelloWorldApi {

private LoggingSubSystem logging;

public HelloWorldImpl() {
      logging = KonfigImpl.logApi;
}

public void sayHelloTo(String name ) {
      logging.writeLogMessage( "Message " + name
);
   }
}
```



Umsetzung einer Komponentenarchitektur – Dependency Management

Probleme die enthalten sind:

- Komponenten-Implementierungen können direkt auf die Implementierung der anderen Komponenten zugreifen

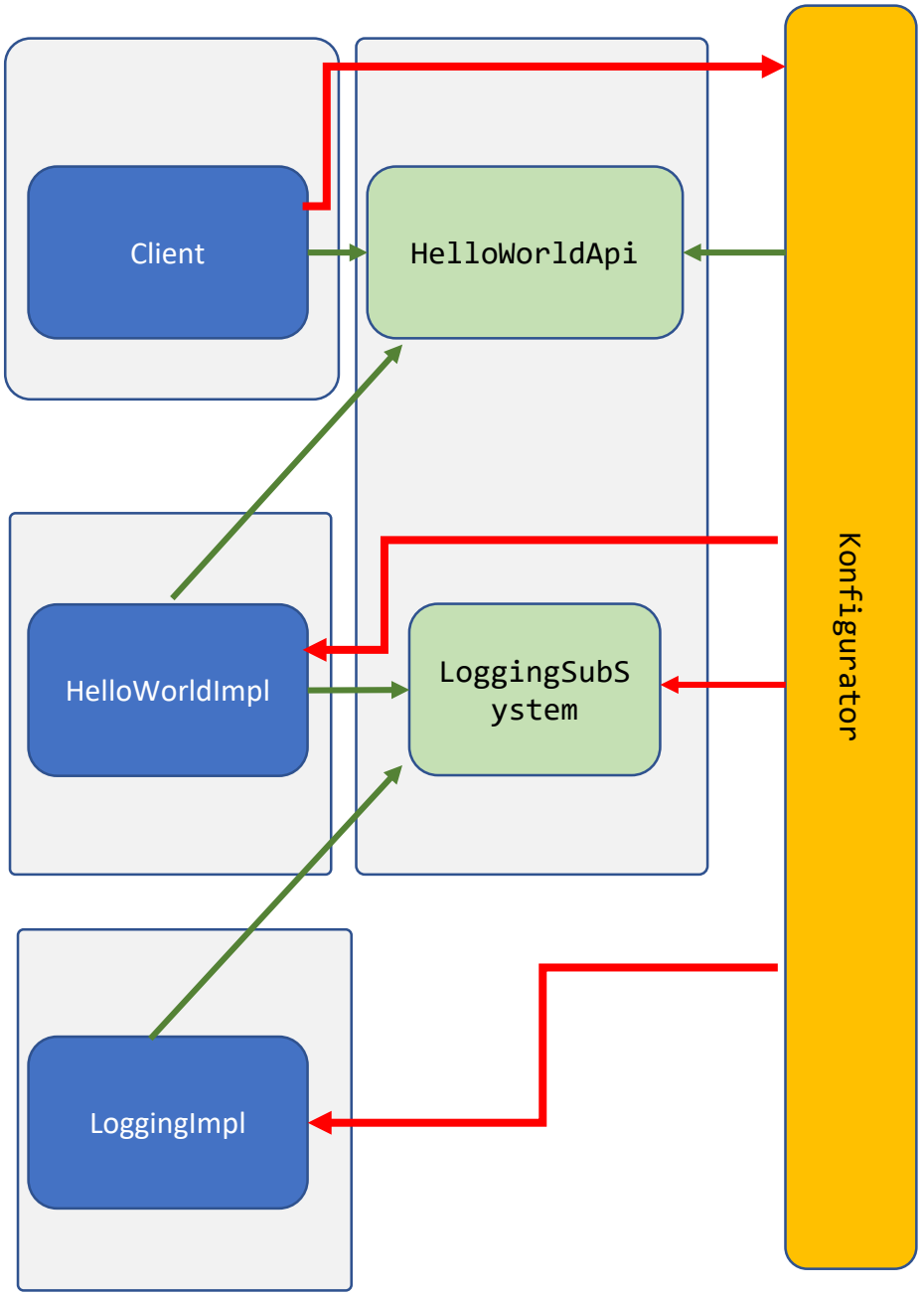
Refakturierung:

- Aufteilen des Programms in verschieden Projekte
- Einführen von Dependency-Management.
- Compile-Time-Dependencies: Komponenten-Implementierungen dürfen zur Übersetzungszeit nur die öffentliche API sehen

```
public class KonfigImpl {

    public static final LoggingSubSystem logApi
                        = new LoggingImpl();

    public static final HelloWorldApi helloApi
                        = new
HelloWorldImpl(logApi);
}
```

```
public class Client {
public static void main(String[] args) {
        HelloWorldApi helloWorld =
KonfigImpl.helloApi;
    helloWorld.sayHelloTo( "the world" );
    }
}
```

```
public class HelloWorldImpl implements
HelloWorldApi {

private LoggingSubSystem logging;

public HelloWorldImpl(LoggingSubSystem logging) {
      this.logging = logging;
}

public void sayHelloTo(String name ) {
      logging.writeLogMessage( "Message " + name
);
    }
}
```



HelloWorld_ComponentArchitecture_V1

Umsetzung einer Komponentenarchitektur – Wiederverwendbarkeit

Probleme die enthalten sind:

- Konfigurator ist sehr starr
- Keine Wiederverwendbarkeit
- Re-Konfiguration der erzeugten Komponenten benötigt eine "Neu-Kompilierung"

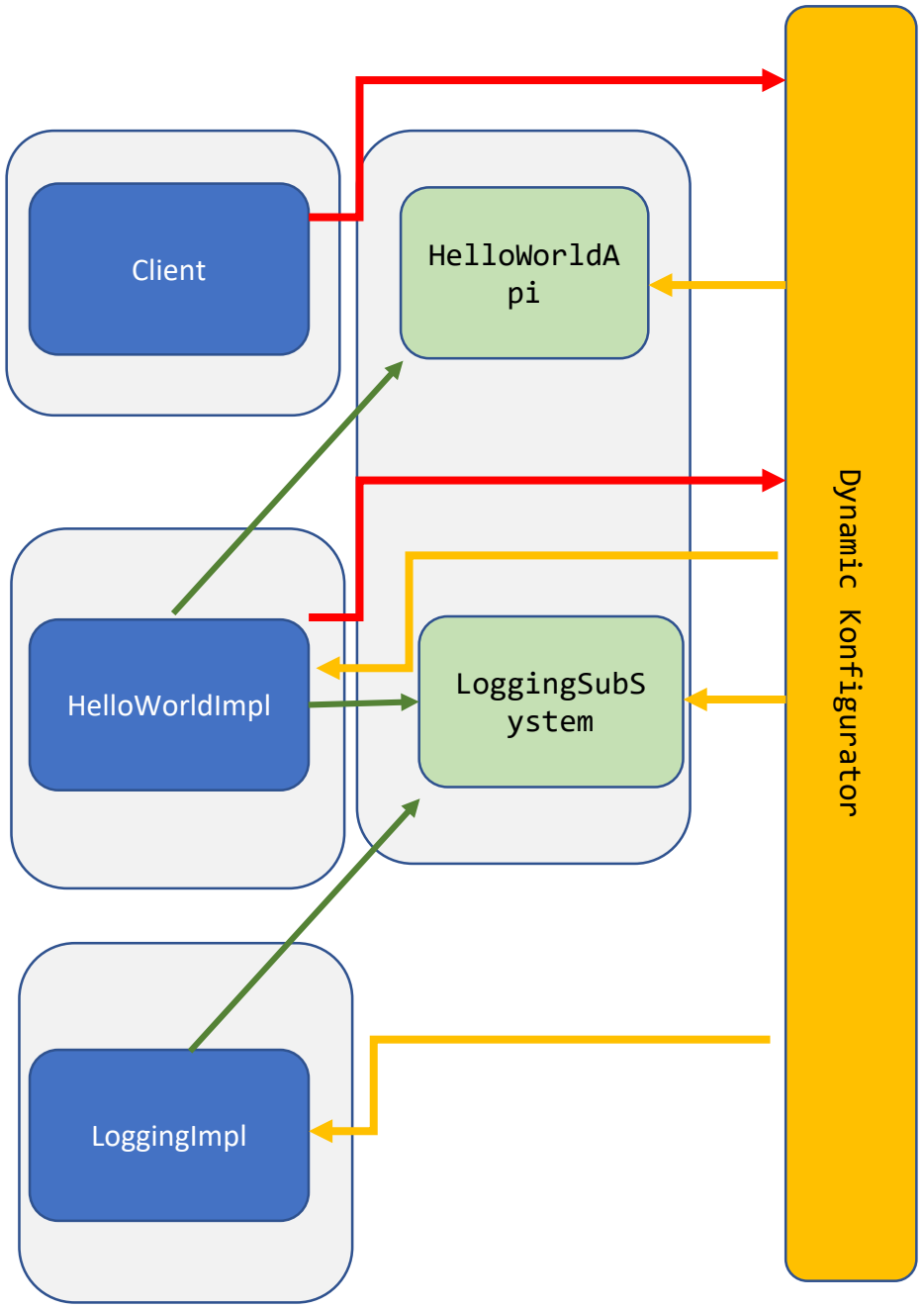
Refakturierung:

- Der Konfigurator soll so implementiert werden dass er universell für verschiedene Projekte benutzt werden kann.
- Entkopplung des Konfigurator von den konkreten Komponenten
- Verwendung von xml-Konfigurations-Dateien welche die implementierten Komponenten beschreiben
- Automatisches auffinden der XML-Konfiguration und erzeugen der Komponenten

```java
public class ServiceConfigurator {
    /*
     * Dynamische Konfiguration über XML
     * und java reflection
     */
}
```

```java
public class Client {
public static void main(String[] args) {
HelloWorldApi helloWorld =
ServiceConfigurator.getInstance().getService(
"HelloWorldComponent", HelloWorldApi.class );
helloWorld.sayHelloTo( "the world" );
    }
}
```

```java
public class HelloWorldImpl implements
HelloWorldApi {

@InjectComponent
private LoggingSubSystem logging;

public void sayHelloTo(String name ) {
      logging.writeLogMessage( "Message " + name
);
    }
}
```



HelloWorld_ComponentArchitecture_V2

Umsetzung einer Komponentenarchitektur – Vereinfachung

Probleme die enthalten sind:

- Es ist eine synchronisation der XML-Konfiguration und der internen Paket- und Klassennamen notwendig.
- Wird dies vergessen kann es zu einem nicht ausführbaren Programmsystem kommen

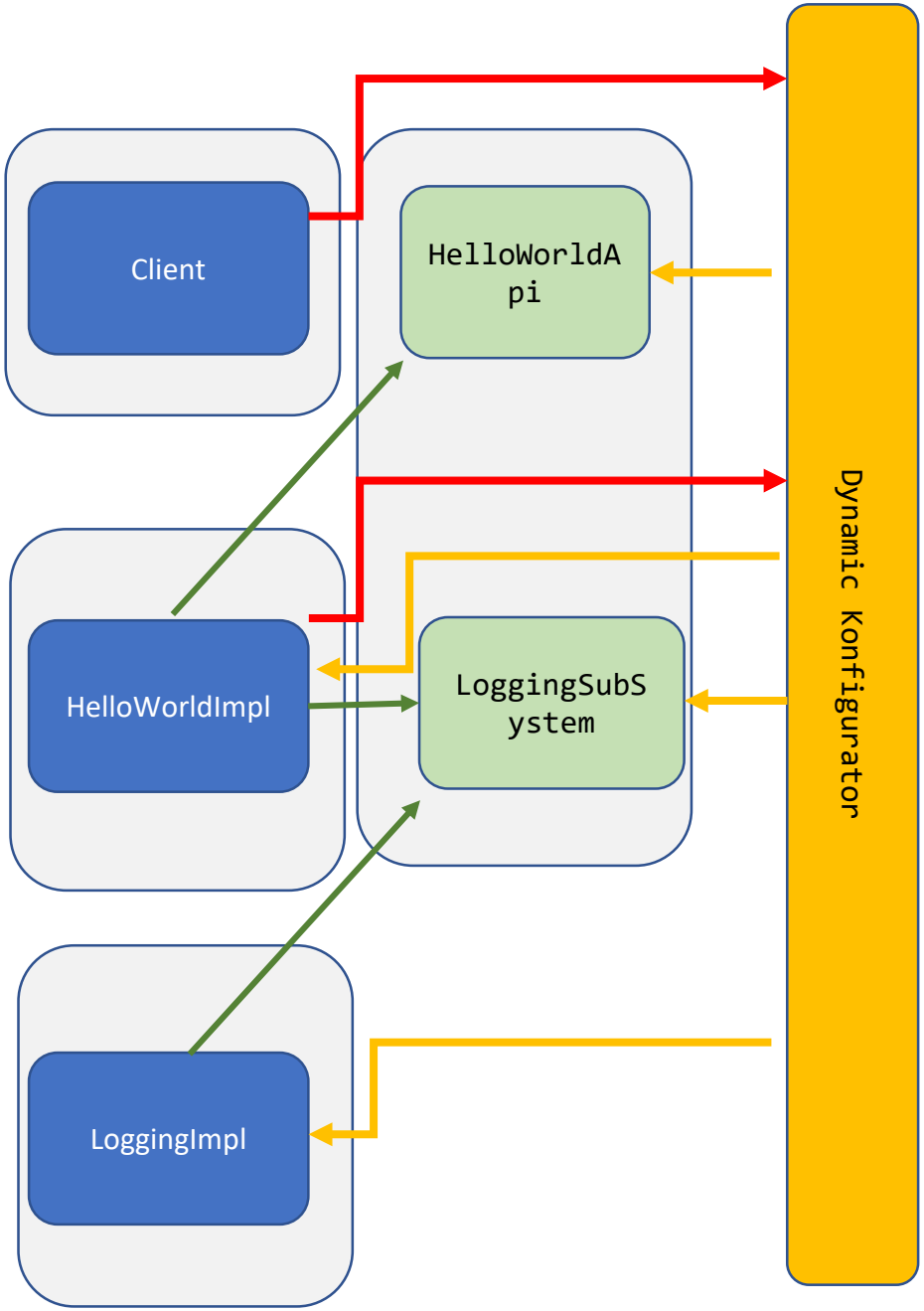
Refakturierung:

- Verwendung von Annotationen um eine Klasse als Implementierung einer Komponenten zu kennzeichen
- Automatisches auffinden und erzeugen der Komponenten auf Basis der Annotationen

```java
public class ServiceConfigurator {
    /*
     * Dynamische Konfiguration über Annotations
     * und java reflection
     */
}
```

```java
public class Client {
public static void main(String[] args) {
HelloWorldApi helloWorld =
ServiceConfigurator.getInstance().getService(
"HelloWorldComponent", HelloWorldApi.class );
helloWorld.sayHelloTo( "the world" );
    }
}
```

```java
@ManagedComponent
public class HelloWorldImpl implements
HelloWorldApi {

@InjectComponent
private LoggingSubSystem logging;

public void sayHelloTo(String name ) {
      logging.writeLogMessage( "Message " + name
);
    }
}
```



HelloWorld_ComponentArchitecture_V3

Umsetzung einer Komponentenarchitektur – Aufteilen des Frameworks

Probleme die enthalten sind:

- Ein Entwickler hat Zugriff auf die Implementierungsklassen des Komponenten-Framework

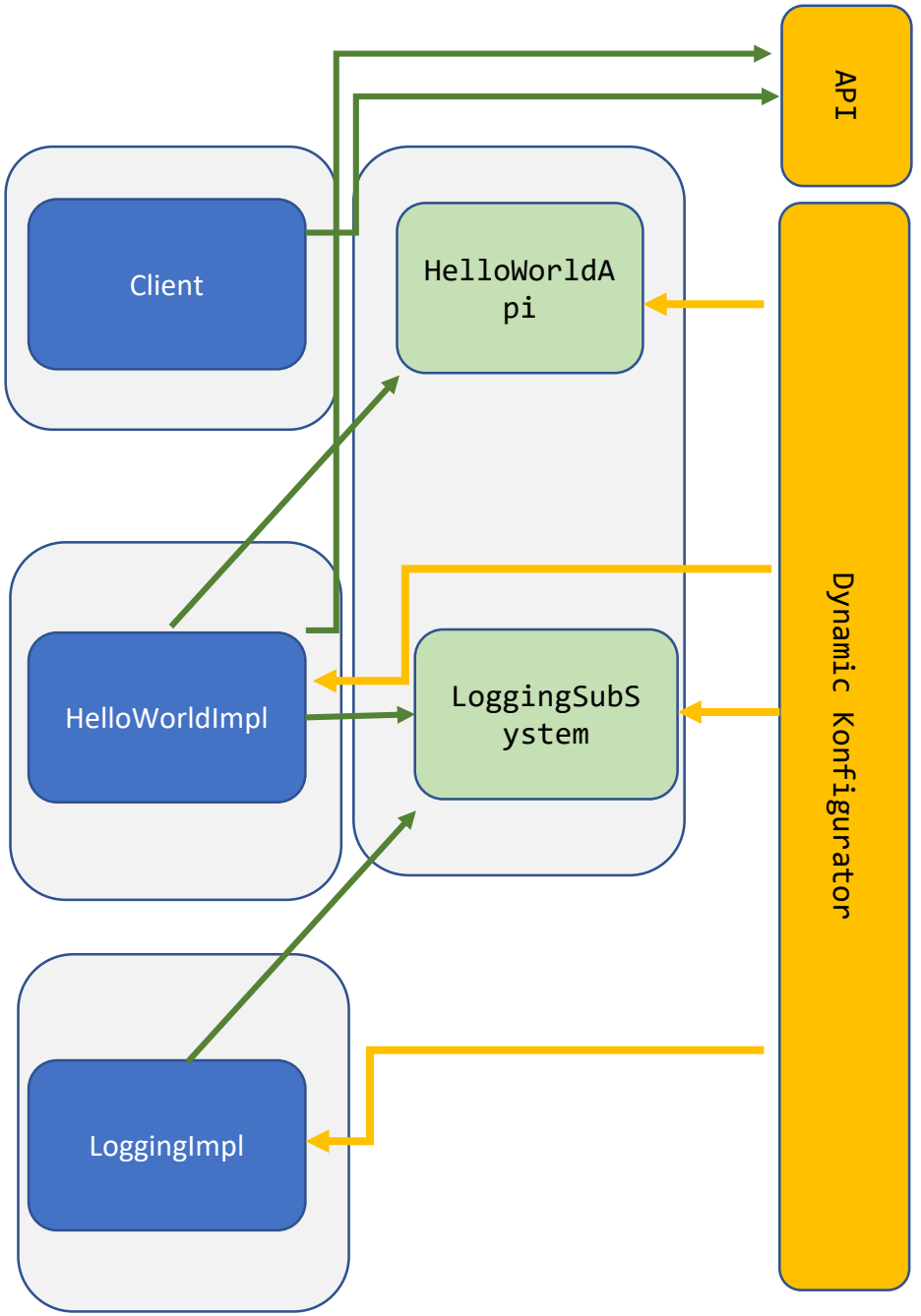
Refakturierung:

- Auch die Implementierung des Komponentenframeworks sollte guten Design-Prinzipien unterliegen. D.h. auch hier solle in Client nur die minimale Information zur Nutzung des Frameworks besitzen. D.h. auch hier sollte es eine öffentliche API geben die von der Implementierung getrennt ist.
- Idee:
- Trenne das Komponenten-Framework in ein API und ein Implementierungsprojekt
- Instanziiere das Framework dynamisch mit Hilfe einer Factory-Klasse

```java
public class ServiceConfigurator {
    /*
     * Dynamische Konfiguration über Annotations
     * und java reflection
     */
}
```

```java
public class Client {
public static void main(String[] args) {
HelloWorldApi helloWorld =
ServiceConfigurator.getInstance().getService(
"HelloWorldComponent", HelloWorldApi.class );
helloWorld.sayHelloTo( "the world" );
    }
}
```

```java
@ManagedComponent
public class HelloWorldImpl implements
HelloWorldApi {

@InjectComponent
private LoggingSubSystem logging;

public void sayHelloTo(String name ) {
      logging.writeLogMessage( "Message " + name
);
    }
}
```



Der Konfiguration – Komponenten zu einem Ganzen verbinden

Weitere Erweiterungen:

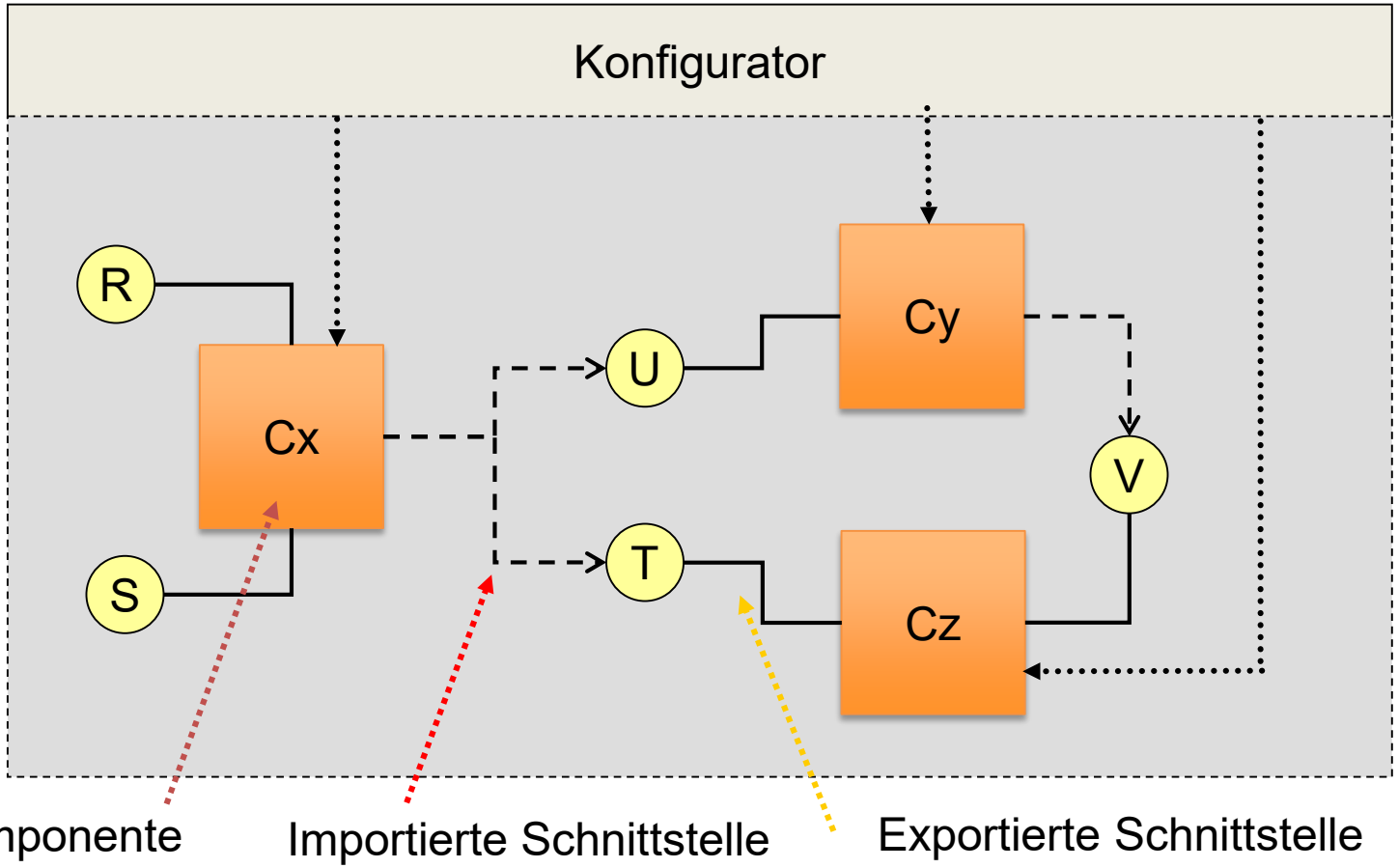
- Oftmals ist es notwendig. das für die gleiche Schnittstelle mehrere Komponentenimplementierungen zur selben Zeit existieren. Dies ist dann notwendig wenn die gleiche Funktionalität mit verschiedenen Nicht-Funktionalen-Anforderungen benötigt wird. Beispiel: Logging soll Plaintext oder verschlüsselt sein. Beides soll gleichzeitig möglich sein und druch den Benutzer der Komponente definiert werden können.
- Annotiere jede Komponentenimplementierung zusätzlich mit einem Attribute subType welches die besondere Eigenschaft der Komponente definiert
- Erweitere das Injektions-Ziel (@InjectComponent) ebenfalls ume in Attribute subType. Dadurch kann der Client definieren welche besondere Eigenschaft einer Komponente benötigt wird.

Der Konfiguration – Komponenten zu einem Ganzen verbinden

Weitere Erweiterungen:

- Komponentenarchitkturen erlauben die bessere Testbarkeit einer Applikation da man leichter das Prinzip des isolierten Tests herstellen kann. Soll eine Komponente getestet werden welche Abhängigkeiten zu weiteren Komponenten hat müssen deshalb alle abhängigen Komponenten durch Platzhalter (Mock's) ersetzt werden. Grundlegende Frage: Wie können Abhängigkeiten zu Komponenten durch Testplatzhalter ersetzten werden?
- Hierzu kommen sog. Alternatives zum Einsatz. Alternatives sind alternative Implementierungen einer Komponente, wobei immer nur eine der Implementierungen aktiv sein darf. Folgendes Grundprinzip gilt: Es gibt eine "produktive Implementierung" einer Komponente und mehre Alternative. Soll eine Alternative verwendet werde muss dies expliziet freigeschalten werden.
- Definiere für eine Kompnenten ob es sich um eine Standardkomponenten oder um eine Alternative handelt
- Definiere beim Initialisieren des Systems ob eine Alternative oder der Standard verwendet werden soll

Der Konfiguration – Komponenten zu einem Ganzen verbinden



Konfiguration: Komponenten verbinden

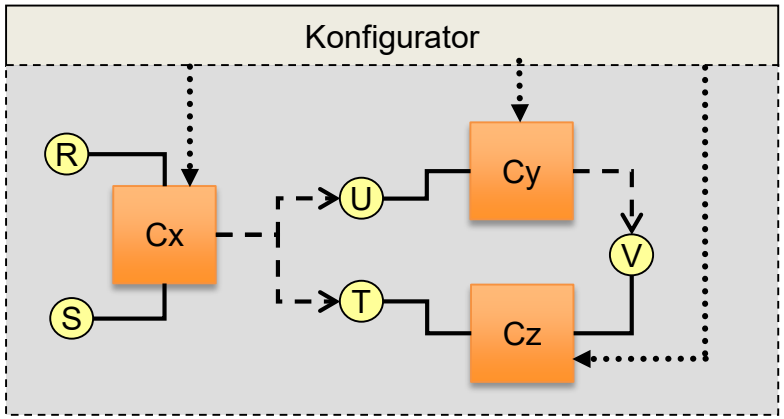
- Der Konfigurator (= Software welches die Konfiguration vornimmt)
  - sieht sowohl die Schnittstellen der Komponenten
  - als auch die Implementierung der Komponenten

- Die Konfiguration kann auf verschiedene Arten implementiert werden
  - Direkter Aufruf eines Konstruktors
  - Indirekter Aufruf eines Konstruktors über das Muster „Factory“
  - Verwendung eines Namendienstes
  - Konfiguration zur Übersetzungszeit (Compiler/Linker)

Konfiguration: Komponenten verbinden (Objektorientiert,Java)

Schritt 1:

- Festlegen der Schnittstellenimplementierung
- Implementieren der Komponenten



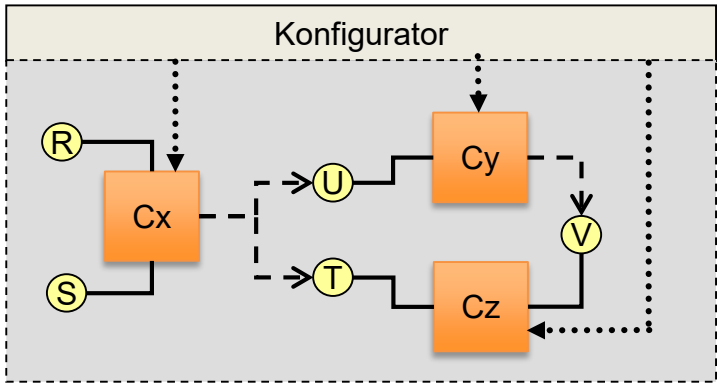

```
// Komponenten Cx, Cy, Cz als Java-Klassen implementieren
// inkl. der zugehörigen Interfaces

public class Cx implements R, S { //... }

public class Cy implements U    { //... }

public class Cz implements V, T { //... }
```

Konfiguration: Komponenten verbinden (Objektorientiert,Java)

```
public class Konfigurator {

  private static Cx cx;
  private static Cy cy;
  private static Cz cz;

  public static void configureSystem() {

    cx = Cx.getComponent();
    cy = Cy.getComponent();
    cz = Cz.getComponente();

    cy.setV( cz );
    cx.setT( cz );
    cx.setU( cy );
  }

  public static R getR() {
    return cx;
  }

  public static S getS() {
    return cx;
  }
}
```

Schritt 2:

- Verbindung zwischen den Komponenten herstellen

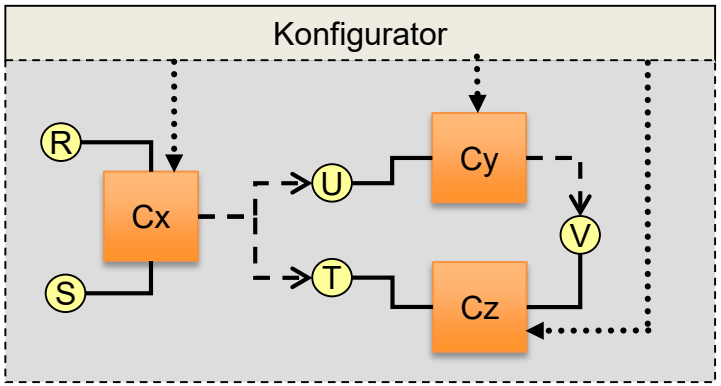


Konfiguration: Komponenten verwenden (Objektorientiert,Java)

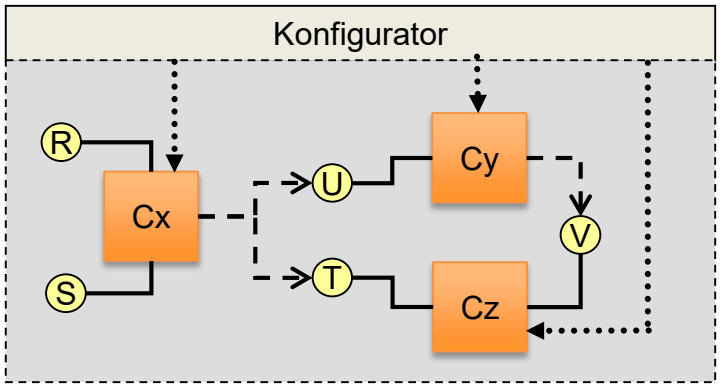
- Komponenten werden ausschließlich über Schnittstellen angesprochen.
- Nur der Konfiguration kennt Implementierung und Schnittstelle
- Nur dadurch ist die optimale Entkopplung gewährleistet.

Schritt 3:
- Komponenten über die Schnittstelen ansprechen

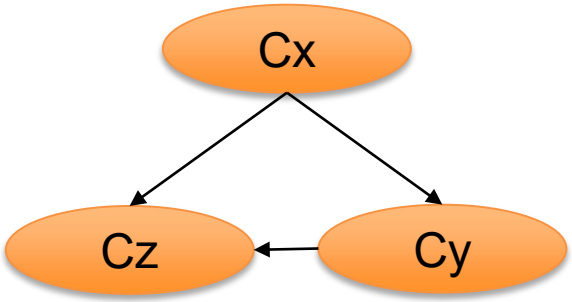
```
...

  R r = Konfigurator.getR();

  r.doSomething();

...
```



Konfiguration: Dependencymanagement

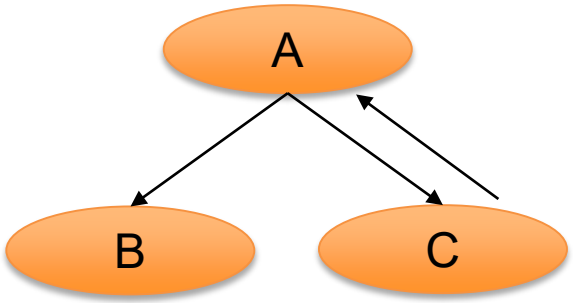


Erstellung des Konfigurationsgraphen

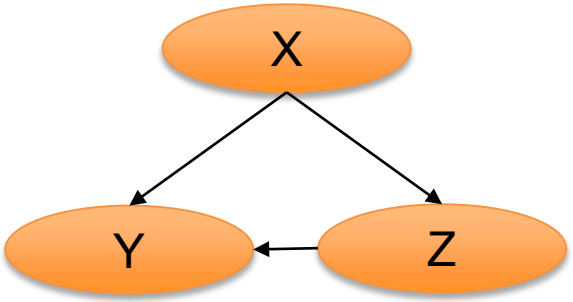


- Die Komponente definieren die Knoten
- Die Importbeziehungen definierten die Kanten

Konfiguration: Komponenten verbinden



- Der Konfigurationsgraph muss „zyklenfrei“ sein. Andernfalls entstehen unbeherrschbare Abhängigkeiten zwischen den Komponenten.
- Keine definierte Initialisierungsreihenfolge



- Der Konfigurationsgraph sollte auch „kreisfrei“ sein. Andernfalls entstehen Seiteneffekte die evtl. zu Problemen führen können

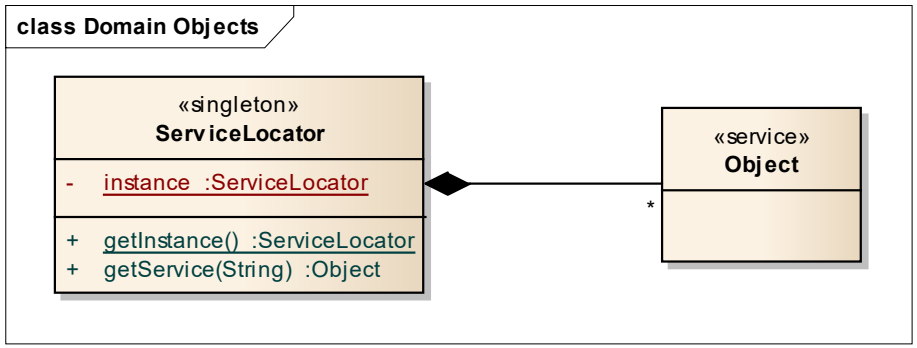
Implementierung eines Konfigurators

Anforderungen an eine Komponentenarchitektur

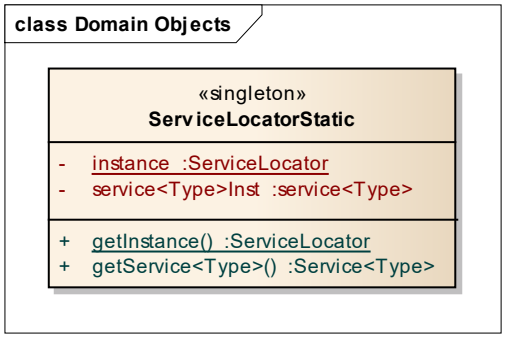
- Die Komponenten müssen unabhängig von der Implementierung der referenzierten Komponenten sein
- Der Konfigurationsmanager muss das „**Inversion of Control**“ – Prinzip anwenden um das wissen der Komponenten in Richtung des Containers zu verlagern.
- **Inversion of Control**: Die Komponenten wissen nichts über Ihre Umgebung. Die Umgebung steuert die Komponenten und deren Beziehungen zueinander
- Mögliche Implementierungsvarianten
  - Service Locator
  - Dependency Injection (Setter, Interface und Constructor Injection)
  - Compiler/Linker zur Übersetzungszeit

Implementierung eines Konfigurators – Service Locator

- Grundidee: Es existiert eine ServiceLocator-Klasse welches alles Wissen über die vorhanden Services (Komponenten) hat
- Eine Komponente frägt diese ServiceLocator-Instanz an um die Referenz auf eine weitere Komponente zu erhalten.



Dynamischer Service Locator



Statischer Service Locator

Implementierung eines Konfigurators – Service Locator

```
//Prinzipieller Aufbau eines ServiceLocators

public class ServiceLocator {
  private static ServiceLocator locator;
  private Map<String,Object> serviceMap = …;

  public void initialize() {
     // load an initialize all services
     // put each service into the service map
  }

  public static ServiceLocator getInstance() {
    if( locator == null ) {
        locator = new ServiceLocator();
        locator.initialize();
    }
    return locator;
  }

  public <T> T getService( String serviceName, Class<T> cls ) {
      return cls.cast( serviceMap.get( serviceName ) );
  }
}
```

Service Locator Dynamsich

- Die Konfiguration des ServiceLocators kann manuell bzw. dynamisch per XML erfolgen

```
MailKomp service = ServiceLocator.getIntsance().getService( "MailKomp", MailKomp.class );
```

Implementierung eines Konfigurators – Dependency Injection

- Grundidee:
  - Die Abhängigkeit zu den importierten Schnittstellen wird durch den Container(=Laufzeitumgebung) verwaltet.
  - Wird eine Komponente erzeugt werden die Abhängigkeit von außen durch den Container injiziert.
  - Die Komponenten selbst haben keine Abhängigkeiten zu den importierten Schnittstellen

Implementierung eines Konfigurators – Dependency Injection

- Es existieren mehrere Arten von Injection Mechanismen
  - **Setter Injection**: Für jede importiert Schnittstelle existiert ein set-Methode welche extern durch den Konfigurator versorgt wird
  - **Constructor Injection**: Alle importierten Schnittstellen werden bei der Objekterzeugung (Konstruktor) übergeben.
  - **Interface Injection**: Für jedes importierte Interface existiert ein „InjectionInterface“ welches die zugehörige Set-Methode definiert.
  - **Attribute Injection**: Für jede importiert Schnittstelle existiert ein Attribut welches direkt durch den Konfigurator befüllt wird

Dependency Injection

```
//Prinzipieller Aufbau eines ServiceLocators

public class KomponentImpl implements KomponentIf {

    private KomponentOther ref;

    public void setKomponentOther( KomponentOther ref ) {
        this.ref = ref
    }
}
```

Setter Injection

```
//Prinzipieller Aufbau eines ServiceLocators

public class KomponentImpl implements KomponentIf {

    private KomponentOther ref;

    public KomponentImpl( KomponentOther ref, … ) {
        this.ref = ref
    }
}
```

Constructor Injection

Dependency Injection

```
//Beispiel für ein InjectionInterface

public interface KompoInjection {
   public void setKomponentOther( KomponentOther other );
}
```

```
public class KomponentImpl implements KomponentIf,
KompoInjection {

    private KomponentOther ref;

    public void setKomponentOther( KomponentOther ref ) {
        this.ref = ref
    }
}
```

Interface Injection

Dependency Injection

```
//Prinzipieller Aufbau eines ServiceLocators

public class Attribute implements KomponentIf {

    @Inject
    private KomponentOther ref;

}
```

Attribute Injection

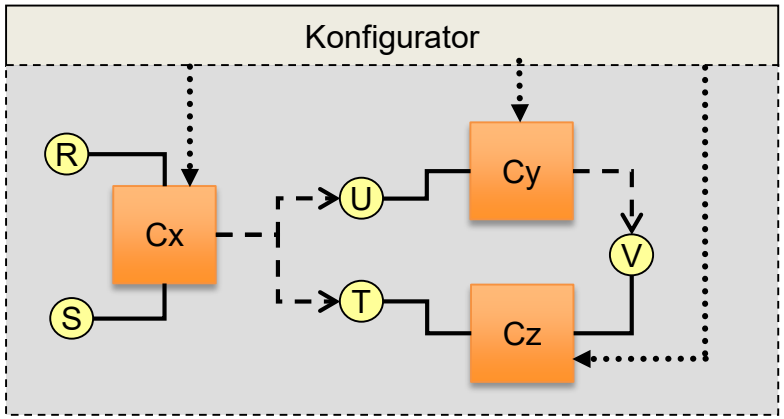
Implementierung des Komponentenkonzepts in C

- Grundidee:
  - Schnittstellen werde über Header-Dateien definiert
  - Jede Komponente wird als eigenständige Bibliothek definiert, wobei die Schnittstellen öffentlich zugänglich sind
  - Zur Übersetzungszeit wird durch den Linker welche Bibliotheken in das Gesamtsystem zu integrieren sind.
  - Der Linker übernimmt also die Aufgabe des Konfiguratons
  - Als Bibliotheken könne statische oder dynamische Bibliotheken verwendet werden.
  - Dynamische Bibliotheken könne zur Laufzeit nachgeladen werden. D.h. auch hier ist eine dynamische Konfiguration des Systems möglich

Implementierung des Komponentenkonzepts in C

Schritt 1:

- Festlegen der Schnittstellenimplementierung
- Implementieren der Komponenten



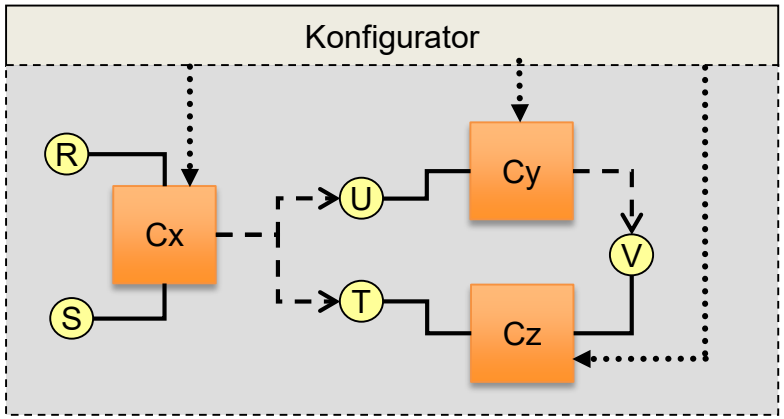

```
// interfacer.h
// Pro Schnittstelle existiert eine
// Header-Datei

#ifndef _INTERFACE_R_
#define _INTERFACE_R_

int methoda( <parameters> )

#endif
```

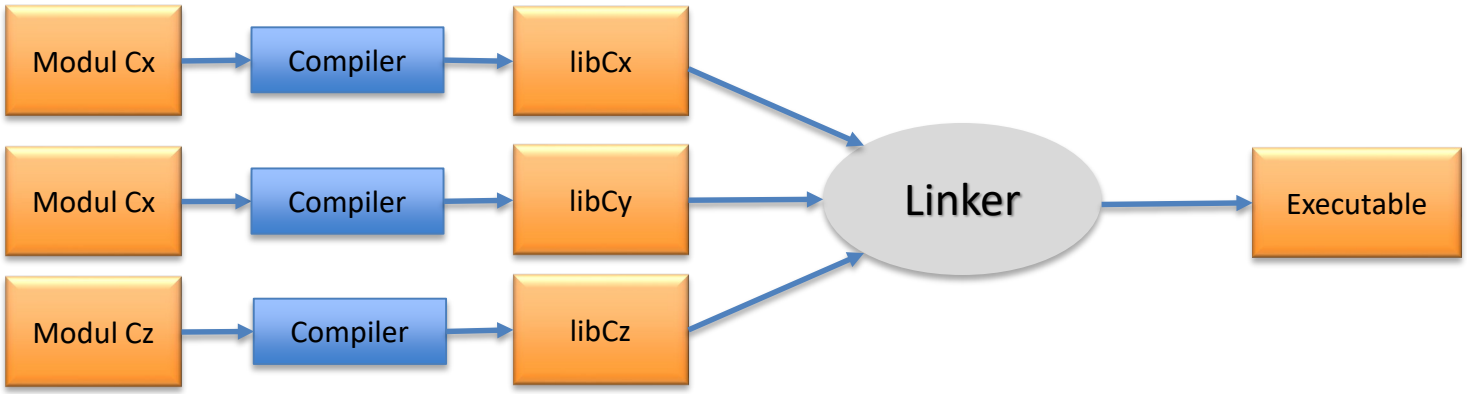
```
#include "interfacer.h"

int methoda( <parameters> )
{
   // Implementierung
}
```

Implementierung des Komponentenkonzepts in C

Schritt 2:

- Verbindung zwischen den Komponenten herstellen





Komposition & Konfiguration – Abschließende Hinweise

- Einen Kompositionsmanager gibt es immer (im einfachsten Fall das Hauptprogramm)
- Komplexe Kompositionsmanager werden z.B. über XML-konfiguriert (z.B. Spring)
- Komponenten können in unterschiedlichen Umgebung laufen solange die importierten Schnittstellen zur Verfügung stehen
- Eine Komposition von Komponenten kann eine Teilmenge der vorhanden Exportschnittstellen exportieren
- Kompositionsmanager können in mehreren Ebenen angeordnet werden