

Modulair redeneren over programma's waarin dynamisch gebonden methode-oproepen voorkomen

Overzicht theorie

- Modulair redeneren: herhaling
- Dynamische binding: herhaling
- Modulair redeneren in aanwezigheid van dynamische binding
 - Basisprincipe
 - Verstrengen van specificaties
 - Het “Liskov Substitution Principle”

Overzicht theorie

- **Modulair redeneren: herhaling**
- Dynamische binding: herhaling
- Modulair redeneren in aanwezigheid van dynamische binding
 - Basisprincipe
 - Verstrengen van specificaties
 - Het “Liskov Substitution Principle”

Modulair redeneren: herhaling

- Niet-modulair redeneren:
 - Om te redeneren over een methode-oproep: kijk naar de implementatie van de opgeroepen methode
 - Wanneer je een implementatie wijzigt, moet je alle oproepers vinden en nakijken (eigenlijk het hele programma)

Modulair redeneren: herhaling

```
String[] getLocations() {  
    return new String[]  
        {"Brussels", "Paris", "Berlin"};  
}  
  
void printLocations(PrintWriter w) {  
    String[] locations = getLocations();  
    for (int i = 0; i < 3; i++)  
        w.println(locations[i]);  
}
```

Modulair redeneren: herhaling

```
String[] getLocations() {  
    return new String[]  
        {"Brussels", "Paris"};  
}
```

```
void printLocations(PrintWriter w) {  
    String[] locations = getLocations();  
    for (int i = 0; i < 3; i++)  
        w.println(locations[i]);  
}
```

Modulair redeneren: herhaling

```
/** @return An array of strings */
String[] getLocations() {
    return new String[]
        {"Brussels", "Paris", "Berlin"};
}

void printLocations(PrintWriter w) {
    String[] locations = getLocations();
    for (int i = 0; i < 3; i++)
        w.println(locations[i]);
}
```

Modulair redeneren: herhaling

```
/** @return An array of strings */  
String[] getLocations() {  
    return new String[]  
        {"Brussels", "Paris", "Berlin"};  
}
```

```
void printLocations(PrintWriter w) {  
    String[] locations = getLocations();  
    for (int i = 0, i < 3; i++)  
        w.println(locations[i]);  
}
```


Modulair redeneren: herhaling

```
/** @return An array of strings */  
String[] getLocations() {  
    return new String[]  
        {"Brussels", "Paris", "Berlin"};  
}  
  
void printLocations(PrintWriter w) {  
    for (String loc : getLocations())  
        w.println(loc);  
}
```

Modulair redeneren: herhaling

```
/** @return An array of strings */  
String[] getLocations() {  
    return new String[]  
        {"Brussels", "Paris"};  
}  
  
void printLocations(PrintWriter w) {  
    for (String loc : getLocations())  
        w.println(loc);  
}
```

Modulair redeneren: herhaling

- Niet-modulair redeneren:
 - Om te redeneren over een methode-oproep: kijk naar de implementatie van de opgeroepen methode
 - Wanneer je een implementatie wijzigt, moet je alle oproepers vinden en nakijken (eigenlijk het hele programma)
- Modulair redeneren:
 - Voorzie elke methode van een specificatie
 - Om te redeneren over een methode-oproep: kijk enkel naar de specificatie van de opgeroepen methode
 - De implementatie van elke methode M moet voldoen aan de specificatie van M
 - Wanneer je de implementatie van M wijzigt, moet je enkel nakijken dat je nieuwe implementatie ook voldoet aan de specificatie van M

Overzicht theorie

- Modulair redeneren: herhaling
- **Dynamische binding: herhaling**
- Modulair redeneren in aanwezigheid van dynamische binding
 - Basisprincipe
 - Verstrengen van specificaties
 - Het “Liskov Substitution Principle”

Dynamische binding: Herhaling

- Statische binding: opgeroepen methode = *opgeloste* methode (oplossing van *method call resolution*-proces tijdens compilatie, gebaseerd op statisch type ontvanger-**uitdrukking** en argument-uitdrukkingen)
- Dynamische binding: opgeroepen methode = opgeloste methode OF methode die opgeloste methode overschrijft (afhankelijk van klasse ontvanger-**object** tijdens uitvoering)

Dynamische binding: Herhaling

```
abstract class Company {  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
        String[] getLocations()  
        {  
            return new String[]  
                {"Brussels", "Paris",  
                "Berlin"};  
        }  
    }
```

Opgeloste
methode

Statisch type ontvanger-uitdrukking

Oproep

Opgeroepen
methode

```
class Program  
    void printLocations  
        (Company c) {  
        String[] locations =  
            c.getLocations();  
        for (int i = 0; i < 3; i++)  
            println(locations[i]);  
    }  
    void main() {  
        printLocations(  
            new CompanyA());  
    }
```

Klasse ontvanger-object

Overzicht theorie

- Modulair redeneren: herhaling
- Dynamische binding: herhaling
- **Modulair redeneren in aanwezigheid van dynamische binding**
 - Basisprincipe
 - Verstrengen van specificaties
 - Het “Liskov Substitution Principle”

Overzicht theorie

- Modulair redeneren: herhaling
- Dynamische binding: herhaling
- Modulair redeneren in aanwezigheid van dynamische binding
 - **Basisprincipe**
 - Verstrengen van specificaties
 - Het “Liskov Substitution Principle”

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
    String[] getLocations()  
    {  
        return new String[]  
        {"Brussels", "Paris",  
        "Berlin"};  
    }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        String[] locations =  
            c.getLocations();  
        for (int i = 0; i < 3; i++)  
            println(locations[i]);  
    }  
    void main() {  
        printLocations(  
            new CompanyA());  
    }  
}
```

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
            {"Brussels", "Paris",  
             "Berlin"};  
    }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        String[] locations =  
            c.getLocations();  
        for (int i = 0; i < 3; i++)  
            println(locations[i]);  
    }  
    void main() {  
        printLocations(  
            new CompanyA());  
    }  
}
```

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    abstract String[]  
        getLocations();  
}
```

```
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
            {"Brussels", "Paris",  
             "Berlin"};  
    }  
}
```

```
class CompanyB extends Company {  
    /** @return An array of strings of length 2 */  
    String[] getLocations() { return new String[] {"Vienna", "Prague"}; }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        String[] locations =  
            c.getLocations();  
        for (int i = 0; i < 3; i++)  
            println(locations[i]);  
    }  
    void main() {  
        printLocations(  
            new CompanyB());  
    }  
}
```

Modulair redeneren i.a.v. dynamische binding

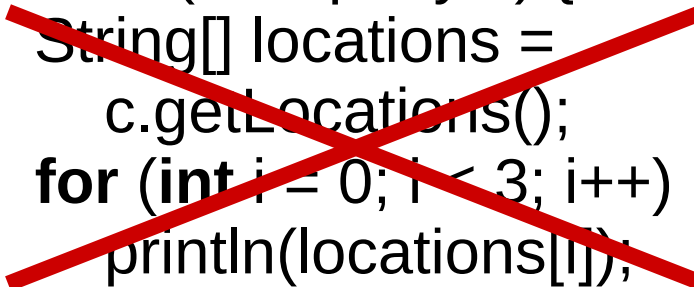
- Niet-modulair redeneren:
 - Kijk naar implementatie van alle mogelijke opgeroepen methodes
- Half-modulaire aanpak:
 - Kijk naar specificatie van alle mogelijke opgeroepen methodes
 - Wanneer je een overschrijvende methode toevoegt, moet je alle potentiële oproepers vinden en nakijken

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    /** @return An array of strings */  
    abstract String[]  
        getLocations();  
}  
  
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
            {"Brussels", "Paris",  
             "Berlin"};  
    }  
}
```

```
class CompanyB extends Company {  
    /** @return An array of strings of length 2 */  
    String[] getLocations() { return new String[] {"Vienna", "Prague"}; }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        String[] locations =  
            c.getLocations();  
        for (int i = 0; i < 3; i++)  
            println(locations[i]);  
        }  
    void main() {  
        printLocations(  
            new CompanyB());  
        }  
}
```



Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    /** @return An array of strings */  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
            {"Brussels", "Paris",  
             "Berlin"};  
    }  
}
```

```
class CompanyB extends Company {  
    /** @return An array of strings of length 2 */  
    String[] getLocations() { return new String[] {"Vienna", "Prague"}; }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        for (String loc :  
            getLocations())  
            println(loc);  
    }  
    void main() {  
        printLocations(  
            new CompanyB());  
    }  
}
```

Modulair redeneren i.a.v. dynamische binding

- Niet-modulair redeneren:
 - Kijk naar implementatie van alle mogelijke opgeroepen methodes
- Half-modulaire aanpak:
 - Kijk naar specificatie van alle mogelijke opgeroepen methodes
 - Wanneer je een overschrijvende methode toevoegt, moet je alle oproepers vinden en nakijken
- Modulair redeneren:
 - Kijk enkel naar specificatie **opgeloste** methode
 - We zeggen: “*M wordt opgeroepen via specificatie S*” als S de specificatie is van de opgeloste methode.
 - Basisprincipe: *De implementatie van elke methode M moet voldoen aan elke specificatie via dewelke M opgeroepen kan worden.*
 - = specificatie van M en specificatie van elke methode die door M overschreven wordt
 - Wanneer je een overschrijvende methode toevoegt, moet je enkel nakijken dat de implementatie van deze methode voldoet aan de specificatie van de overschreven methode

Overzicht theorie

- Modulair redeneren: herhaling
- Dynamische binding: herhaling
- Modulair redeneren in aanwezigheid van dynamische binding
 - Basisprincipe
 - **Verstrengen van specificaties**
 - Het “Liskov Substitution Principle”

Modulair redeneren i.a.v. dynamische binding

- Basisprincipe: *De implementatie van elke methode M moet voldoen aan elke specificatie via dewelke M opgeroepen kan worden.*
- Afgeleid principe: *Het is voldoende als de implementatie van elke methode M voldoet aan de specificatie van M, EN de specificatie van M is strenger dan de specificatie van elke methode die door M overschreven wordt.*
- We zeggen “specificatie S' is strenger dan specificatie S” als elke denkbare implementatie die voldoet aan S' ook voldoet aan S.

Verstrengen van specificaties

- Een specificatie met preconditionie P' en postconditie Q' is strenger dan een specificatie met preconditionie P en postconditie Q a.s.a.
 - P impliceert P' , EN
 - $(P \text{ en } Q')$ impliceert Q

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    /** @return An array of strings */  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
        {"Brussels", "Paris",  
        "Berlin"};  
    }  
}
```

```
class CompanyB extends Company {  
    /** @return An array of strings of length 2 */  
    String[] getLocations() { return new String[] {"Vienna", "Prague"}; }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        for (String loc :  
            getLocations())  
            println(loc);  
    }  
    void main() {  
        printLocations(  
            new CompanyB());  
    }  
}
```

Overzicht theorie

- Modulair redeneren: herhaling
- Dynamische binding: herhaling
- Modulair redeneren in aanwezigheid van dynamische binding
 - Basisprincipe
 - Verstrengen van specificaties
 - **Het “Liskov Substitution Principle”**

Gedragstypes

- We kunnen een klasse C interpreteren als een *gedragstype* (*behavioral type*)
- Een object O is van gedragstype C als elke methode van O voldoet aan de specificatie van de overeenkomstige methode van C (als C een overeenkomstige methode heeft)
- Een klasse D is een behavioral subtype van een klasse C, als elk object van gedragstype D ook van gedragstype C is (m.a.w. als D voor elke methode van C een overeenkomstige methode heeft met een strengere specificatie)
- Onze aanpak voor modulair redeneren komt neer op:
 - Kijk bij het redeneren over een oproep enkel naar het statisch type van de ontvanger, geïnterpreteerd als gedragstype.
 - Zorg ervoor dat voor elk object O met `O.getClass() == C.class` geldt: O is van gedragstype C (m.a.w. elke methode moet voldoen aan zijn eigen specificatie).
 - Als een klasse D overerft van een klasse C, moet D een behavioral subtype zijn van C.

Het “Liskov Substitution Principle”

- Als een klasse D overerft van een klasse C, moet D een behavioral subtype zijn van C.
- Met andere woorden: *Als een klasse D overerft van een klasse C, dan moet D voldoen aan de voorwaarde “als er in een bepaalde context een object van gedragstype C verwacht wordt, dan volstaat een object van gedragstype D”.*
- Dit wordt het Liskov Substitution Principle genoemd (ook al gaat het in feite om eenzelfde object dat zowel van gedragstype C als van gedragstype D is; er is dus geen sprake van substitutie van een object O' met O'.getClass() == D.class voor een object O met O.getClass() == C.class.)

Modulair redeneren i.a.v. dynamische binding

```
abstract class Company {  
    /** @return An array of strings */  
    abstract String[]  
        getLocations();  
}  
class CompanyA  
    extends Company {  
    /** @return  
        An array of strings  
        of length 3 */  
    String[] getLocations()  
    {  
        return new String[]  
            {"Brussels", "Paris",  
             "Berlin"};  
    }  
}
```

```
class CompanyB extends Company {  
    /** @return An array of strings of length 2 */  
    String[] getLocations() { return new String[] {"Vienna", "Prague"}; }  
}
```

```
class Program {  
    void printLocations  
        (Company c) {  
        for (String loc :  
            getLocations())  
            println(loc);  
    }  
    void main() {  
        printLocations(  
            new CompanyB());  
    }  
}
```