Implementeren in C++

[referentie]

psavalle@synalyse.com

http://www.synalyse.com

“Do not comment bad code -- rewrite it.”

Kernighan and Plauger

Uitgegeven door Synalyse B.V., Zoetermeer.

Opmaak van tekst en ontwerp van omslag door Patrick Savalle.

Druk- en bindwerk door de Borrias Groep.

Dit boek is gemaakt in MS-Word en MS-Visio, aangeleverd in Adobe Acrobat-formaat en Adobe Photoshop-formaat (kaft) en geproduceerd met een Printing-on-demand proces.

ISBN

NUGI 851 (informaticatheorie)

De website van Synalyse is http://www.synalyse.com/

Het adres van het Synalyse-discussieforum is http://synalyse.startforum.nl/

Het adres van de Synalyse C++-codeerstandaard is
http://www.synalyse.com/cppcodeerstandaard/

1. Inhoudsopgave

2. Referentie

2.1 Het Synalyse class-model

Een C++-implementatie wordt gemaakt naar aanl.eiding van (voornamelijk) een class-diagram. Class-diagrammen bestaan uit de volgende elementen (Synalyse objectmetamodel):

- Classes

  Een class is de typering van een object.

- Berichtrelaties

  Een berichtrelatie geeft aan dat tussen objecten van de betreffende classes een berichtrelatie bestaat.

- Aggregatierelaties

  Een aggregatierelatie kan classes van objecten die voorkomen op verschillende niveaus van aggregatie met elkaar relateren. Hiermee kunnen hiërarchische structuren ‘plat’ worden weergegeven.

- Associatierelaties

  Een associatierelatie geeft aan dat tussen objecten van de betreffende classes een passieve koppeling bestaat (omdat objecten bij elkaar horen) en dat die objecten op hetzelfde niveau van aggregatie bestaan.

- Interfaces

  Een interfaces is opsomming van berichten die tussen objecten worden uitgewisseld. Een interface geeft aan welke berichten een object van een bepaalde class kan sturen naar een object van een andere (of dezelfde) class.

- Generalisatierelaties

  Een generalisatierelatie is een middel om verschillende classes met zo min mogelijk herhaling van specificatie te kunnen noteren.

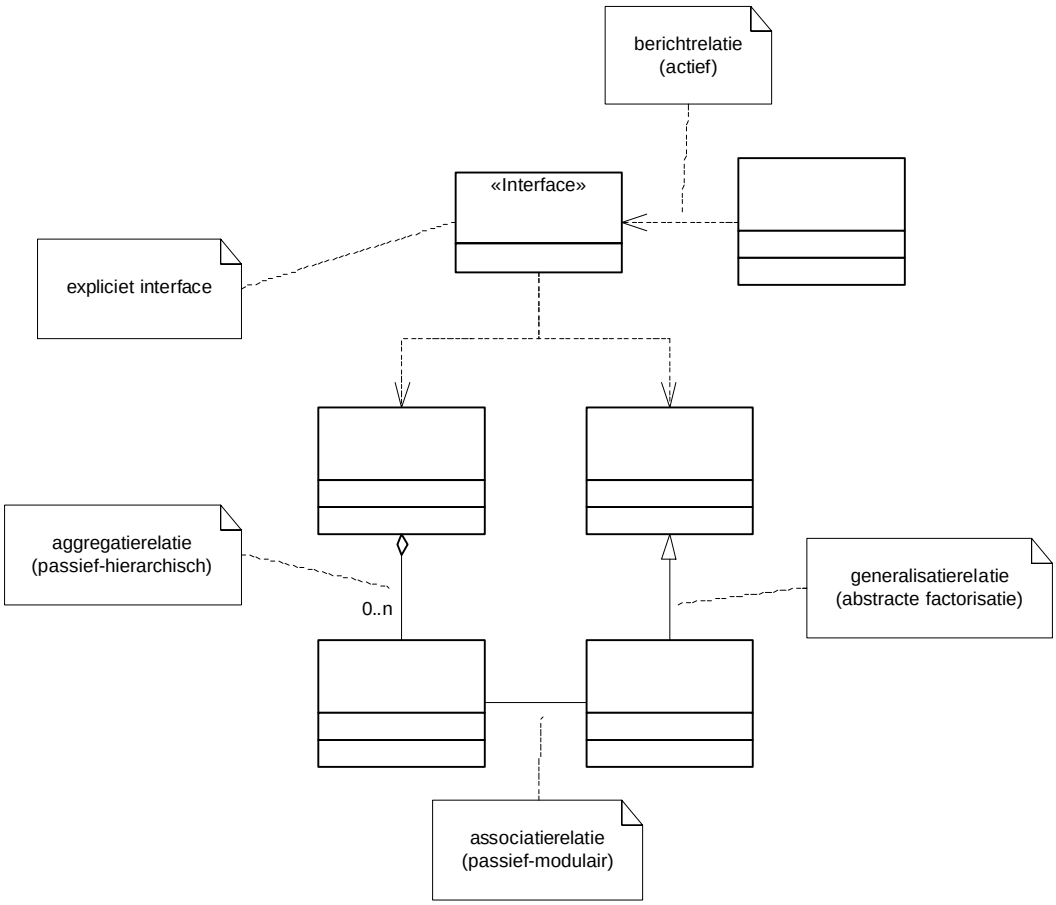
- Class-templates

  Een class-template is een geparameteriseerde class.

- Design-Patterns

  Een design-pattern is een geparameteriseerd class-model.

De onderstaande figuur toont alle elementen van een Synalyse class-diagram.



2.2 Eigenschappen van hoogwaardige implementaties

Een implementatie is een systeem dat is gebouwd volgens een specificatie. De eisen die worden gesteld aan een implementatie komen daarom voort uit de wens om de specificaties zo goed mogelijk te volgen.

De eisen die worden gesteld aan een systeem dat wordt gebouwd volgens specificatie zijn:

- Evenredigheid[1]

  Een goed systeem is evenredig *met de specificatie.* Er is sprake van evenredigheid als er in alle opzichten een vaste verhouding is tussen implementatie en specificatie. De specificatie is altijd gemakkelijk te herkennen in de implementatie. De modules en koppelingen die de specificatie vermeldt, komen ook voor in de implementatie. Een kleine wijziging in specificatie leidt tot een evenredig kleine wijziging in de implementatie. Evenredigheid moet vanuit elk gezichtspunt gelden.

  In plaats van evenredigheid zou ook van *herkenbaarheid* kunnen worden gesproken.

- Correctheid

  Correcte implementaties werken precies zoals is gespecificeerd, zijn dus consistent met de functionele specificatie. Correctheid mag nooit worden opgeofferd ten gunste van andere eisen!

- Robuustheid

  *Robuustheid is de bestendigheid tegen abnormale condities***.** Abnormale condities zijn condities die niet expliciet worden voorzien door de specificaties.

  Robuustheid en correctheid zijn complementaire eigenschappen. Correctheid betreft de werking volgens specificatie en robuustheid de werking buiten specificatie.

Een belangrijke opmerking bij correctheid is dat het grootste deel van de specificatie toch altijd impliciet zal (moeten) zijn. Een voorbeeld van impliciete specificaties zijn natuurkundige wetten of wiskundige regels en definities. Alles specificeren is onmogelijk en ongewenst.

2.3 De kwaliteit van een C++-implementatie

De kwaliteit van een C++-implementatie, los van correctheid, kan dan aan vier grootheden worden afgelezen. Deze grootheden zijn:

- De structurele gelijkwaardigheid

  Een goede implementatie heeft dezelfde structuur als het ontwerp (dezelfde afhankelijkheden en koppelingen tussen de verschillende modules).

[1] ‘Klein probleempje, klein systeempje!’

Tastbare / objectieve meetpunten van de fysieke structuur zijn bv.:

- de include-graaf
- de fysieke module-structuur
- functionele clustering / componentenstructuur

- De semantische integriteit

  Een goede implementatie stemt gebruikte syntaxis zodanig af op gewenste semantiek dat vergissingen zoveel mogelijk kunnen worden gezien of vermeden door de compiler.

  De semantische integriteit kan worden beoordeeld aan de hand van bijvoorbeeld:

  - eenvoud van formulering
  - interface-kwaliteit
  - de canonieke vormen
  - const-correctheid (zogenaamde type-conversie lussen)

- De syntactische kwaliteit

  Een goede implementatie heeft eenvoudige en leesbare code van uniforme kwaliteit.

  Syntactische kwaliteit kan worden beoordeeld aan:

  - leesbaarheid
  - eenvoud
  - naamgeving
  - uniformiteit

- De technische volwaardigheid of instrumentatie

  Een goede implementatie heeft voldoende instrumentatie om effectief te zijn, maar niet zoveel dat het de eigenlijke functionele code vertroebeld.

Instrumentatie mag nooit de eigenlijke functionaliteit beïnvloeden.

2.4 Incrementele implementatie

Het proces dat de implementatiekwaliteit maximaal garandeert vanuit een werkwijze bestaat uit zeven stappen:

1 - Het begrijpen van het ontwerp

Behalve de notatie moet ook de bedoeling van de gekozen constructies worden begrepen.

2 - Het bepalen van de gewenste fysieke afhankelijkheden en bouwvolgorde

De logische structuur vertalen in de gewenste (meestal een gelijkwaardige) fysieke structuur.

Het vertalen van de class-diagrams in include-graven.

De include-graven voorzien van laagnummers.

3 - Het bouwen en opleveren van het 'structurele' raamwerk

Modules aanmaken en voorzien van de include-statements.

4 - Het bouwen en opleveren van het 'semantische' raamwerk

De declaratiebestanden voorzien van correcte, initiële class-declaraties.

5 - Het bouwen en opleveren van het 'technische' raamwerk

De definitiebestanden voorzien van functie-karkassen, voorzien van instrumentatie zoals conditie-controles. logging en tracing.

6 - Het verdelen van het werk

7 - Het afbouwen en opleveren van initiële versies

3. Structurele gelijkwaardigheid

3.1 Volgorde in structuur

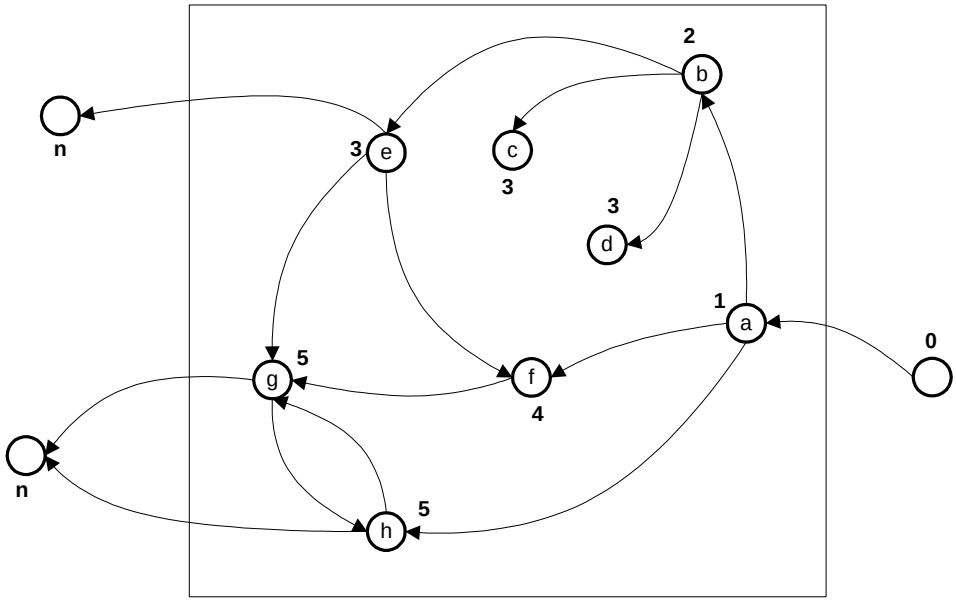
Bij het wijzigen, bekijken of maken van modules of files of classes die afhankelijk zijn van elkaar is er altijd een optimale volgorde om dat te doen. In elke afhankelijkheidsstructuur kan namelijk een ‘natuurlijke’ volgorde worden onderscheiden. Deze volgorde kan worden getoond door het expliciet aanbrengen van *volgnummering*. Het aanbrengen van volgnummering is een belangrijke bewerking op een structuur die later in verschillende contexten zal terugkomen. De volgnummers zijn bijvoorbeeld een belangrijk gegeven bij het kiezen van de volgorde waarin de delen van een nieuw systeem moet worden gebouwd. De volgnummering geeft de volgorde en richting aan waarin effecten zich door het systeem voortplanten en volgorde waarin het systeem moet worden opgebouwd, getest en veranderd.

> *Elke structuur heeft een natuurlijke volgorde waarin effecten zich door de structuur voortplanten.*

De regels die moeten worden gevolgd om een structuur van volgnummers te voorzien zijn eenvoudig:

- Modules die niet tot de betreffende structuur horen maar wel direct daardoor worden gebruikt, de aanbieders van het systeem, hebben *volgnummer 0*.
- Modules die niet tot de betreffende structuur horen maar dit wel direct gebruiken, de afnemers van het systeem, hebben per definitie *volgnummer n* (letter n).
- Modules die wel tot de betreffende structuur horen maar geen interne afhankelijkheden hebben, hebben *volgnummer 1*.
- Modules die tot de betreffende structuur horen en ook nog eens een interne afhankelijkheid hebben krijgen als volgnummer *één nummer hoger dan de modules met het hoogste volgnummer waarvan ze zelf afhankelijk zijn*. Het volgnummer van een dergelijke module is daarmee ook altijd de langste afstand gemeten in 'aantal' afhankelijkheden naar laag 0.

- Indien de structuur cyclische afhankelijkheden bevat, *dan krijgen alle modules in die daarmee gekoppeld zijn hetzelfde volgnummer*.



Figuur 1: de volgnummering van een gerichte graaf

Enkele consequenties van het op deze manier aanbrengen van volgorde zijn:

- Alle modules met hetzelfde volgnummer vormen een *laag*.
- Een module is nooit afhankelijk van modules met een hoger volgnummer.
- Directe en indirect *cyclische afhankelijkheden* (lussen) maken de betreffende structuur minder volgordelijk en meer monolithisch.
- De volgnummers zijn géén identificerende nummers, er kunnen dubbele volgnummers voor komen.

Op basis van volgnummers kan ook worden voorspeld wat voor gevolgen wijzigingen in de structuur hebben:

- Een module met een bepaald volgnummer afhankelijk maken van een module met lager volgnummer geeft nooit lussen en verandert de volgnummers niet. Deze wijziging beïnvloedt de bestaande structuur dus niet.
- Een module met een bepaald volgnummer afhankelijk maken van een module met hetzelfde volgnummer geeft nooit lussen maar verandert de volgnummering.

- Een module met een bepaald volgnummer afhankelijk maken van een module met een hoger volgnummer geeft lussen en verandert de volgnummering. Dit is een ingrijpende wijziging voor de bestaande structuur.

3.2 Drie semantische stelsels of soorten structuur

Een C++-programmeur moet rekening houden met drie verschillende 'semantische stelsels' of of soorten structuur:

- Het class meta-model van de ontwerpen (Synalyse, UML, Booch, enz.)
- Het logische C++ class-model
- De fysieke C++ file-structuur

Het Synalyse class-model kent meerdere soorten relaties:

- Specialisatie
- Aggregatie
- Associatie
- Bericht

Het C++ class-model kent ‘slechts’ de volgende relaties:

- Overerving
- Methodekoppeling (aanroep van methoden)
- Attribuutkoppeling (attributen van class-type)

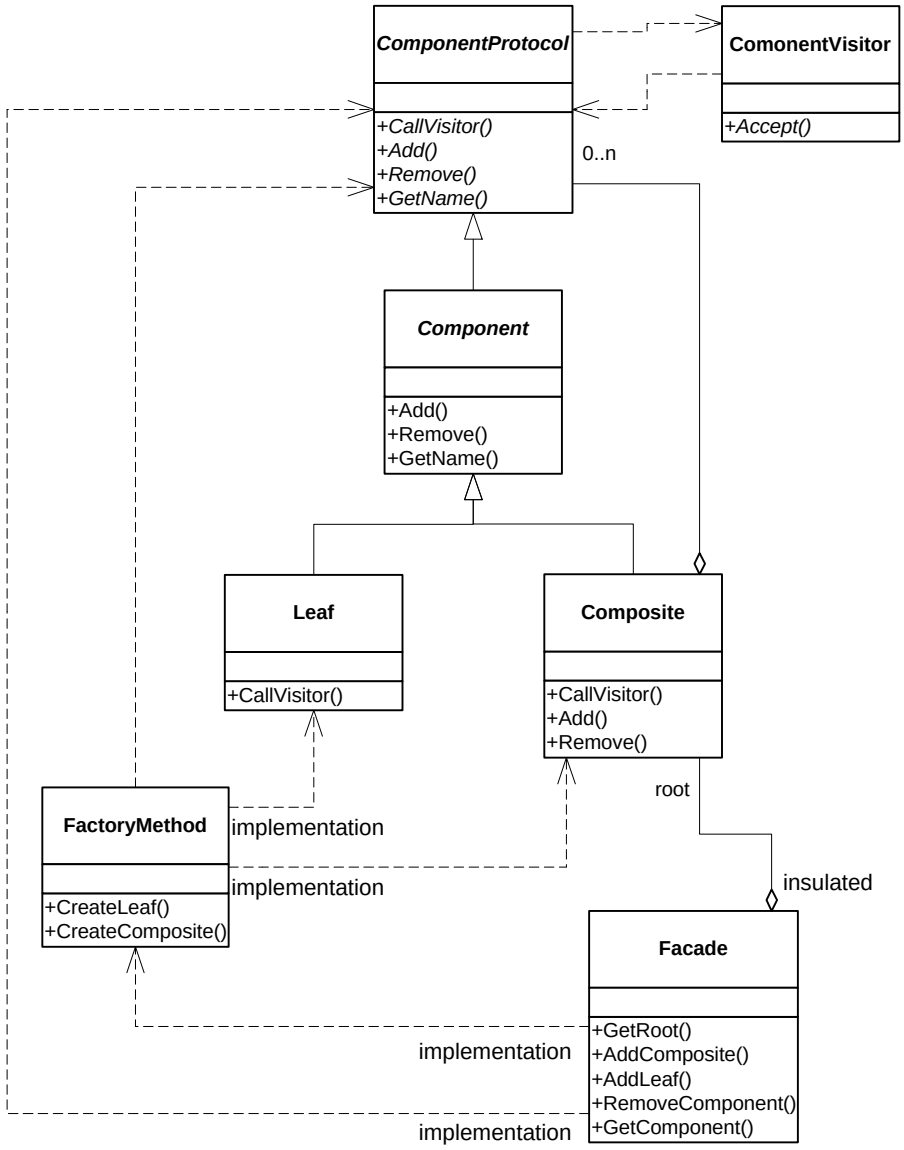
De file-structuur van een C++ model kent tenslotte maar één soort relatie:

- #include of file-koppeling (een transitieve koppeling)

Door de verschillende semantieken is het zeer moeilijk om een ontwerp waarheidsgetrouw na te maken in C++!

> *C++ werkt met een logische model én een fysiek model en die twee moeten op elkaar én op het ontwerp worden afgestemd. Het logische model heeft andere soort relaties en modules dan het fysieke model.*

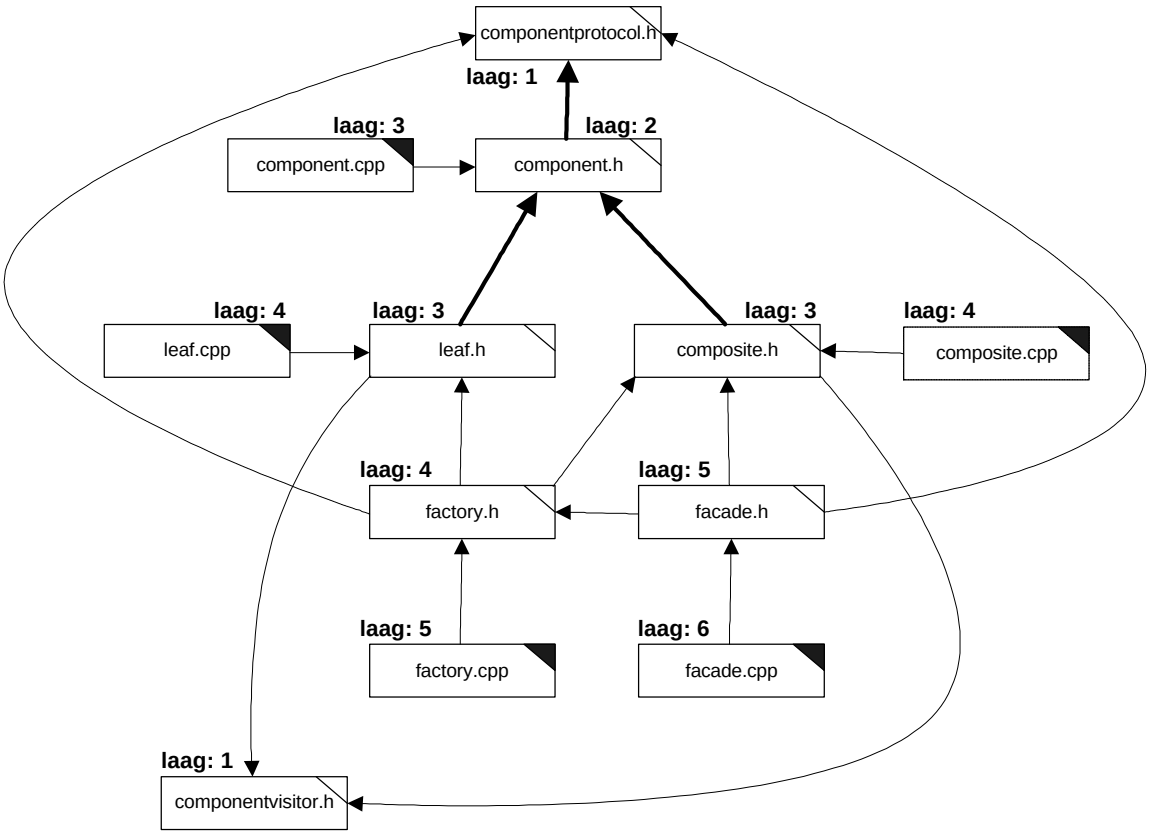
3.3 Voorbeeld: UML class-diagram 'Patterns'



3.4 Voorbeeld: include-graaf 'Patterns'

– redundante includes in streepjeslijnen

– specialisatie-hierarchie in dikke lijnen



3.5 Encapsulatie en insulatie

Een goed object-model scheidt interface van implementatie.

> *C++ scheidt niet interface van implementatie, maar declaratie van definitie en dat is niet hetzelfde.*

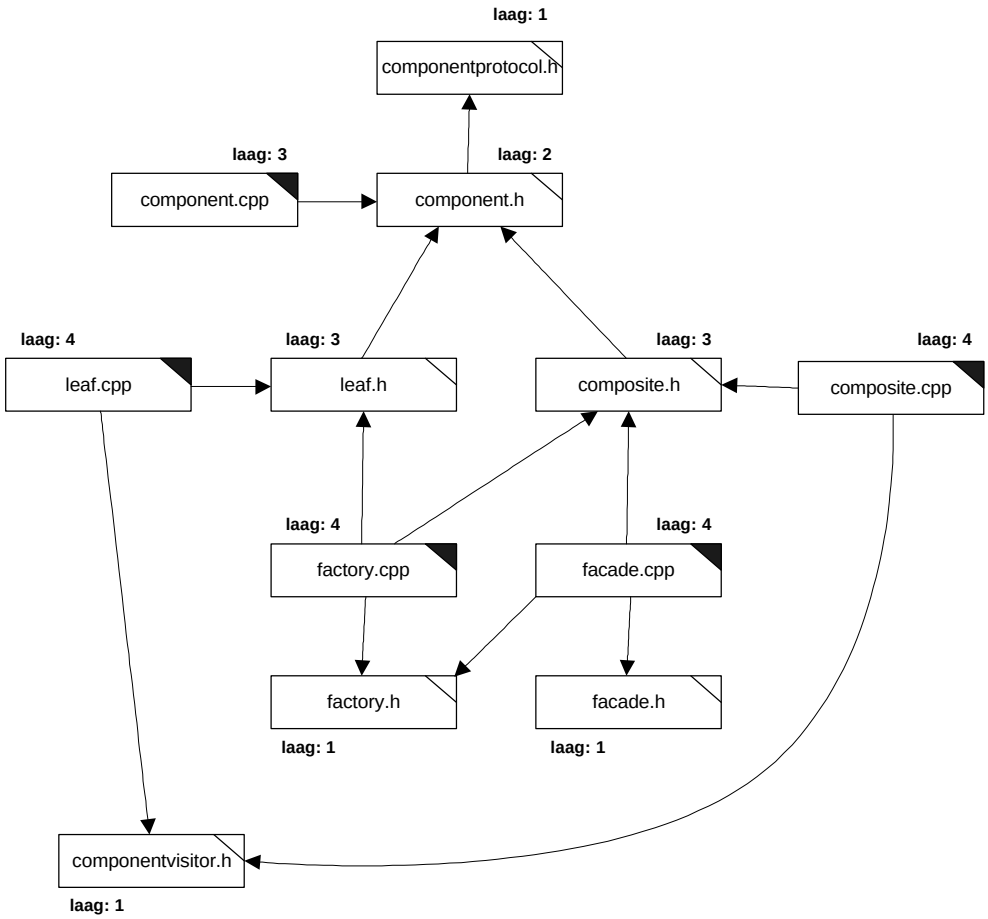
Het onderstaande voorbeeld toont het probleem, de class-declaratie is groter dat de class-interface:

```
#include “otherclass.hpp”

class SomeClass
{
public:
  void Operation();
private:
  OtherClass Attribute;
};
```

De private-section en het include-statement horen niet tot het public-interface maar wel tot de declaratie en is dus indirect toch van invloed op clients. De technieken die nodig zijn om de declaratie te ontdoen van elementen die niet tot de public-interface horen, heten **insulatietechnieken**. Insulatie is dus C++ encapsulatie.

3.6 Voorbeeld: ideale include-graaf 'Patterns'

Een include-graaf toont de file-structuur van C++. De graaf kan worden voorzien van volgnummers.



3.7 Forward-declaration

De meest eenvoudige techniek voor insulatie is de forward-reference. De toepassing van forward-references maakt soms een include-statement overbodig.

Een include-statement is voor een (class-)declaratie alleen nodig wanneer het betreffende type ‘by-value’ als class-attribute wordt gebruikt.

De volgende statements hebben dus **geen** include-statement nodig:

```
// note forward declaration instead of #include
class Otherclass;

class SomeClass
{
public:

  OtherClass function1() const;
  OtherClass& function2() const;
  OtherClass* function3() const;

  void procedure( OtherClass );
  void procedure( OtherClass& );
  void procedure( OtherClass* );

private:

  OtherClass* variable1;
  OtherClass& variable2;

};
```

3.8 Protocol-class

Synalyse-interfaces kunnen in C++ worden gerealiseerd met protocol-classes. Een class is een protocol-class als:

- het declaratiebestand van de class geen include-statements heeft zodat er geen interface-accumulatie kan optreden.
- (In geval van specialisatie is er wel een include-statement nodig. In dit geval moet de geinclude class zelf ook een protocol-class zijn.)
- de class alleen een public-section heeft zodat er geen implementatie-details zichtbaar zijn.
- de class alleen pure-virtual functions bevat (alleen declaraties, geen definities).
- de class geen functies met default-argumenten heeft omdat deze implementatie-afhankelijk zijn en wijzigingen van de class-implementatie kunnen laten doorwerken naar clients.

- de class een lege, niet inline, virtual destructor bevat zodat deze niet later nog eens moet worden toegevoegd.

Een voorbeeld van een protocol-class:

(figureprotocol.h)

```
#ifndef FIGUREPROTOCOL_H
#define FIGUREPROTOCOL_H

class FigureProtocol
{
public:
  virtual ~FigureProtocol();
  void Draw() const = 0;
  void Save() const = 0;
  void Read() = 0;
  ProtocolClass& Translate( double dx, double dy ) = 0;
  ProtocolClass& Rotate( double angle ) = 0;
};

#endif
```

(figureprotocol.cpp)

```
#ifndef   FIGUREPROTOCOL_H
#include "figureprotocol.h"
#define   FIGUREPROTOCOL_H
#endif

FigureProtocol::~FigureProtocol() {}
```

3.9 Insulator-class

C++ classes kunnen perfect worden geinsuleerd door ze te implementeren als insulator-classes. Bij een insulator-class is de class-declaratie hetzelfde als het class-interface.

Een class is een insulator-class als:

- Het declaratiebestand van de class geen include-statements heeft
- De class alleen maar een public-section heeft
- De class als enige member-variable z’n eigen private implementor-class reference heeft (zie voorbeeld)
- De class geen inline functies heeft
- De class geen functies met default-argumenten heeft
- de class alle benodigde canonieke members heeft (de default constructor, de operator= en de copy-constructor)

- de class een lege, niet inline, virtual destructor bevat

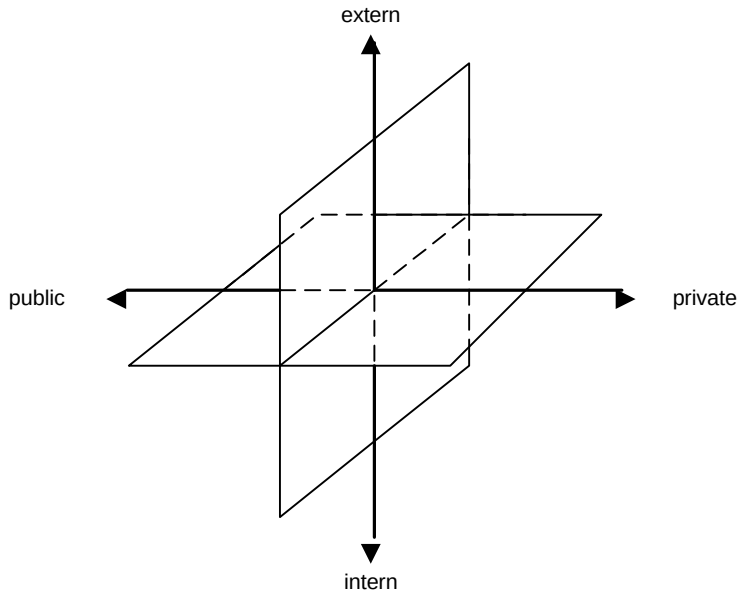
(parser.hpp)

```
#ifndef PARSER_HPP
#define PARSER_HPP

class ParserIMP;
class Parser
{
public:
  Parser();
  Parser( const Parser & copy );
  Parser& operator=( const Parser & copy );
  ~Parser();

  double evaluateExpression ( const char* expression );

private:
  ParserIMP* mImplementor;
};

#endif
```

4. Semantische integriteit

4.1 Statische toestand

Er zijn van een bepaalde C++-class/object verschillende soorten toestand te benoemen:



Figuur 2: de verschillende soorten statische toestand van een C++-class

- Interne toestand

  Interne toestand is de toestand van het object zelf

- Externe toestand

  Externe toestand is de toestand van de omgeving van het object.

- Public toestand

  Public toestand is de toestand van een object die via het interface zichtbaar of opvraagbaar is.

- Private toestand

  Private toestand is de toestand van een object die niet via het interface zichtbaar of opvraagbaar is.

4.2 Bijwerkingen

Methoden zijn er in twee varianten:

- Procedures

  Procedures veranderen de toestand van de class. Procedures hebben geen ‘functie-resultaat’.

- Functies

  Een functie geeft een functie-resultaat, net zoals een zuiver wiskundige functie, bekijkt de toestand van de class of doet een berekening zonder de toestand te wijzigen.

Als een functie (of const-gedeclareerde procedure) toch de toestand van een class (object) wijzigt, dan heet dit een *bijwerking*. Een functie heeft bijwerkingen als hij

- een attribuut van de class wijzigt
- een modifier of procedure aanroept

Een functie met bijwerkingen heeft tot gevolg dat garanties met betrekking tot natuurlijke, wiskundige eigenschappen zoals *referentiele transparantheid* verdwijnen en software niet meer logisch en voorspelbaar werkt. Een (overdreven) voorbeeld:

```
double sqr( double& x )
{
  return x = x * x;   // deliberate side-effect!
}
```

Deze functie is niet meer ‘referentieel transparant’, de volgende code bewijst dat:

```
double a = 2;
double b = sqr( a ) - sqr( a );  // b is not equal to zero!!!
```

Informeel: functies met bijwerkingen zijn onbetrouwbaar omdat het stellen van een vraag het antwoord verandert of kan veranderen.

Bijwerkingen kunnen worden geclassificeerd naar de vier soorten objecttoestand:

- Extern

  Een externe bijwerking is een ongewenste verandering van de externe toestand.

- Intern

  Een interne bijwerking is een ongewenste verandering van de interne toestand.

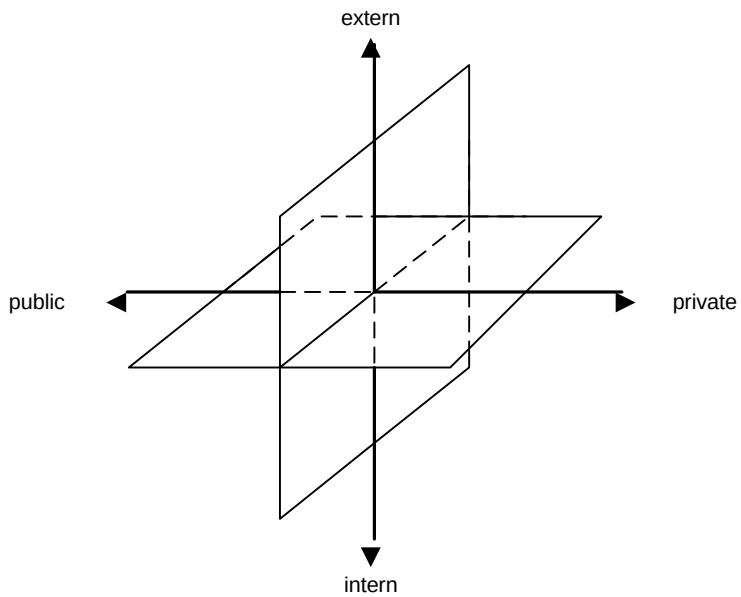
- Public

  Een public bijwerking is een ongewenste verandering van de public toestand.

- Private

  Een private bijwerking is een ongewenste verandering van de private toestand.

Private bijwerkingen zijn onschadelijk en mogen worden toegestaan.



Figuur 3: vier mogelijke soorten bijwerkingen van functies

4.3 Operaties en methoden

Operaties zijn de reacties op berichten die objecten ontvangen. Een operatie wordt geïmplementeerd door een methode. Er zijn verschillende soorten operaties (en methodes):

- constructor

  Een constructor is de operatie die nodig is om een object te creëren. Een object begint zijn leven in zijn constructor-aanroep. De constructor initialiseert het object. Het is de verantwoordelijkheid van de constructor dat het object wordt gecreeerd in een toestand die voldoet aan de invariant.

- modifier

  Een modifier is een operatie die de toestand van het object verandert. Een modifier wordt geïmplementeerd als **procedure**.

- selector

  Een selector is een operatie de toestand van het object opvraagt zonder de toestand te veranderen. Een selector wordt geïmplementeerd als **functie**.

- iterator

  Een iterator is een operatie die de toestand van het object in een bepaalde volgorde opsomt zonder zelf de toestand te veranderen. Een iterator wordt geïmplementeerd als **procedure**.

- destructor

  De destructor is de operatie die nodig is om een object te vernietigen en op te ruimen. Een object eindigt zijn leven in zijn destructor, de destructor geeft alle middelen waar het object beslag op legde weer vrij.

Een operatie heeft enkele belangrijke eigenschappen (Engels acroniem is ACID):

- Atomiciteit

  Een operatie moet of helemaal slagen, of helemaal niet. Als de operatie faalt, moet de oorspronkelijke toestand van het systeem weer gelden (de post-toestand moet weer gelijk zijn aan de pre-toestand).

- Geisoleerdheid

Een operatie is een proces dat als eenheid kan worden aangevraagd. De opbouw van de operatie (de methode) en de tijdelijke tussenresultaten zijn onzichtbaar voor concurrerende operatie.

- Consistentie

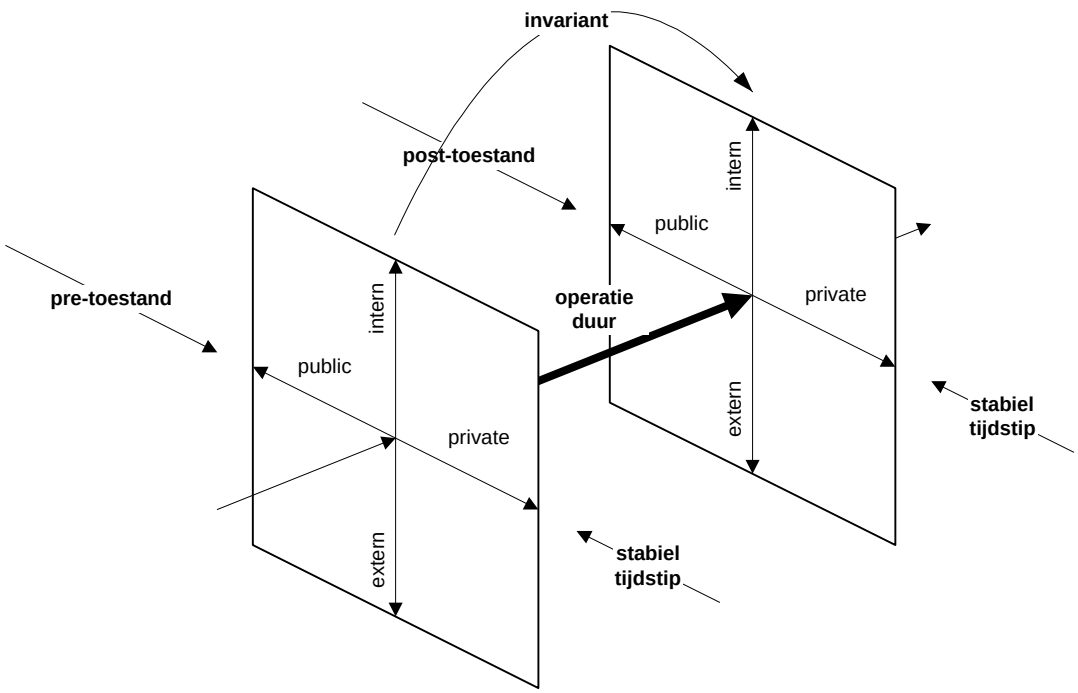
  De operatie laat het systeem altijd in een geldige toestand achter. Elke keer dat een operatie wordt geactiveerd met dezelfde pre-toestand, moet deze dezelfde post-toestand opleveren.

- Duurzaamheid

  De resultaten van de operatie moeten de operatie overleven, zodat het externe systeem ze kan waarnemen.

4.4 Dynamische toestand

Als een operatie de toestand van een object kan veranderen dan is er dus een verschil in toestand voor en na uitvoering van de operatie. Vanuit het gezichtspunt van de operatie zijn er daarom de volgende verschillende toestanden:



Figuur 4: de verschillende soorten dynamische toestand van een object

- Pre-toestand

  De pre-toestand is de toestand voor uitvoering van een operatie.

- Post-toestand

  De post-toestand is de toestand na uitvoering van een operatie

- Invariant-toestand

  De invarianttoestand is de toestand die moet gelden als er geen operatie actief is (op stabiele tijdstippen).

Omdat de tussenresultaten en de werking van een operatie onzichtbaar zijn voor externe objecten, is het ook niet belangrijk of het object tussendoor aan de invariant voldoet. De tijdstippen waarop de toestand van een object kan worden opgevraagd zijn de tijdstippen waarop het object niet actief is, dus waarop er geen operatie in uitvoering is. Deze tijdstippen heten *stabiele tijdstippen.*

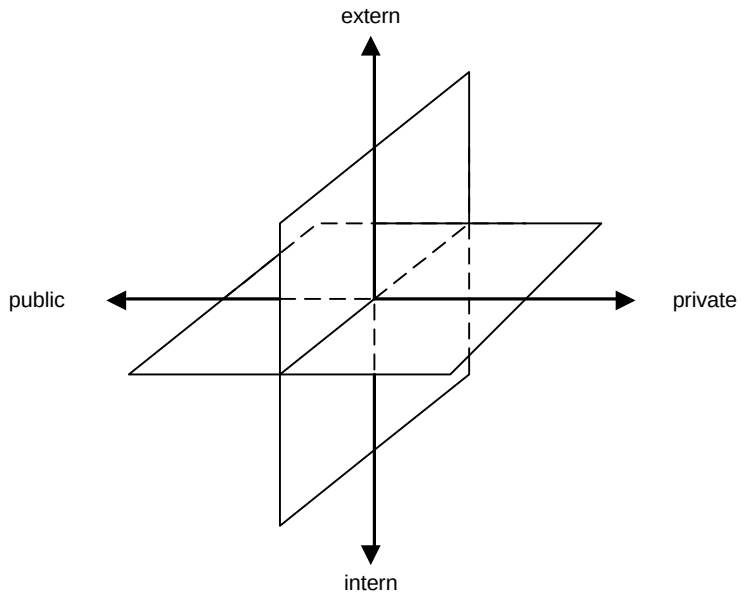
> *Een object moet op zijn stabiele tijdstippen aan zijn invariant voldoen, daartussen hoeft dat niet. De toestand van een object op zijn instabiele tijdstippen is toch niet opvraagbaar door andere objecten.*

4.5 Const-correctheid

Const-correctheid is de eigenschap van een taal die ervoor zorgt dat ongewenste toestandsveranderingen door de compiler kunnen worden gedetecteerd. Bijwerkingen van functies vallen hier ook onder.

Om ervoor te zorgen dat de compiler schendingen van deze eigenschap kan controleren moet de taal deze eigenschap expliciet ondersteunen. In C++ gebeurt dit door gebruik van de keywords **const** en **mutable** en typecasting. (Mutable wordt gebruikt om concrete toestand mee aan te geven.)

Const-correctheid kent vier aspecten, gebaseerd op de verschillende soorten toestand van een object:



Figuur 5: vier aspecten van const-correctheid

- Interne const-correctheid

  Intern const-correct wil zeggen dat een object geen ongewenste toestandsveranderingen in zijn eigen toestand toestaat of veroorzaakt.

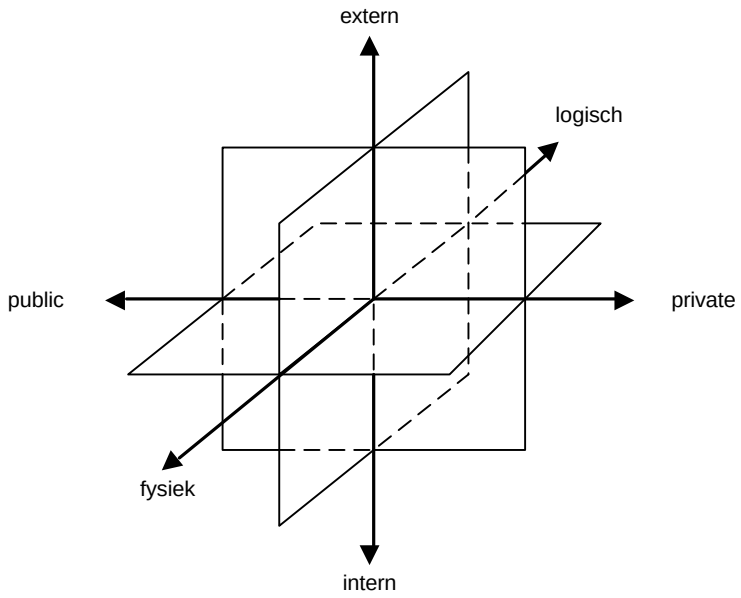
- Externe const-correctheid

  Externe const-correct wil zeggen dat een object geen ongewenste toestandsveranderingen in de toestand van andere objecten toestaat of veroorzaakt.

> *In C++ kan public toestand worden gescheiden van private toestand met behulp van het keyword **mutable**. Alle mutable attributen behoren tot de private toestand.*

4.6 Const-correctheid in C++

C++ heeft door de aanwezigheid van fysieke en logische toestand ook een extra dimensie van const-correctheid die loodrecht staat op die van het ideale model:



Figuur 6: C++ const-correctheid

De extra richtingen zijn:

- Fysieke const-correctheid

  Fysieke const-correctheid betreft de toestand van alle direct en indirect in de interface-file gedeclareerde symbolen.

- Logsiche const-correctheid

  Logisch const-correctheid betreft de toestand van alle direct en indirect in de interface-file gedeclareerde symbolen *die public-toegang heeft*..

Een goede implementatie brengt de C+ const-correctness in overeenstemming met de normale const-correctness van het object-model.

4.7 Const-keyword

Het const-keyword kan worden gebruikt om de compiler de mogelijkheid te geven om de const-correctness te bewaken.

- De const-member-functie

```
class SomeClass
{
```

```
public:
  void ConstMember() const;
};
```

- De const-functieargumenten

```
class SomeClass
{
public:
  void ConstArgument( const OtherClass& Object );
  void SomeFunction ( OtherClass Object );
  void ConstArgument( const OtherClass Object );
};
```

- De const-pointer

```
static const int* Pointer = "Apple";

Pointer++;           // legal
Pointer = "Peach";   // legal
Pointer[0] = 'B';    // illegal
*Pointer = 'C';      // illegal
```

- De const-pointerwaarde

```
static int* const Pointer = "Apple";

Pointer++;           // illegal
Pointer = "Peach";   // illegal
Pointer[0] = 'B';    // legal
*Pointer = 'C';      // legal
```

- De const-variabele

```
static const int* const Pointer = "Apple";
static const int Integer = 10;
```

4.8 Mutable-keyword

Het keyword 'mutable' kan worden gebruikt om fysieke en logische const-correctheid op elkaar af te stemmen. De compiler zal fysieke const-correctheid bewaken, maar de programmeur wil public const-correctheid bereiken. Onderstaande voorbeeld toont een conflict tussen deze twee concepten.

```
class ImageProxy
{
public:
  void Rotate()
  {
    changed = true;
    // bla bla
  }

  void Resample( int new_width, int new_height )
  {
    changed = true;
```

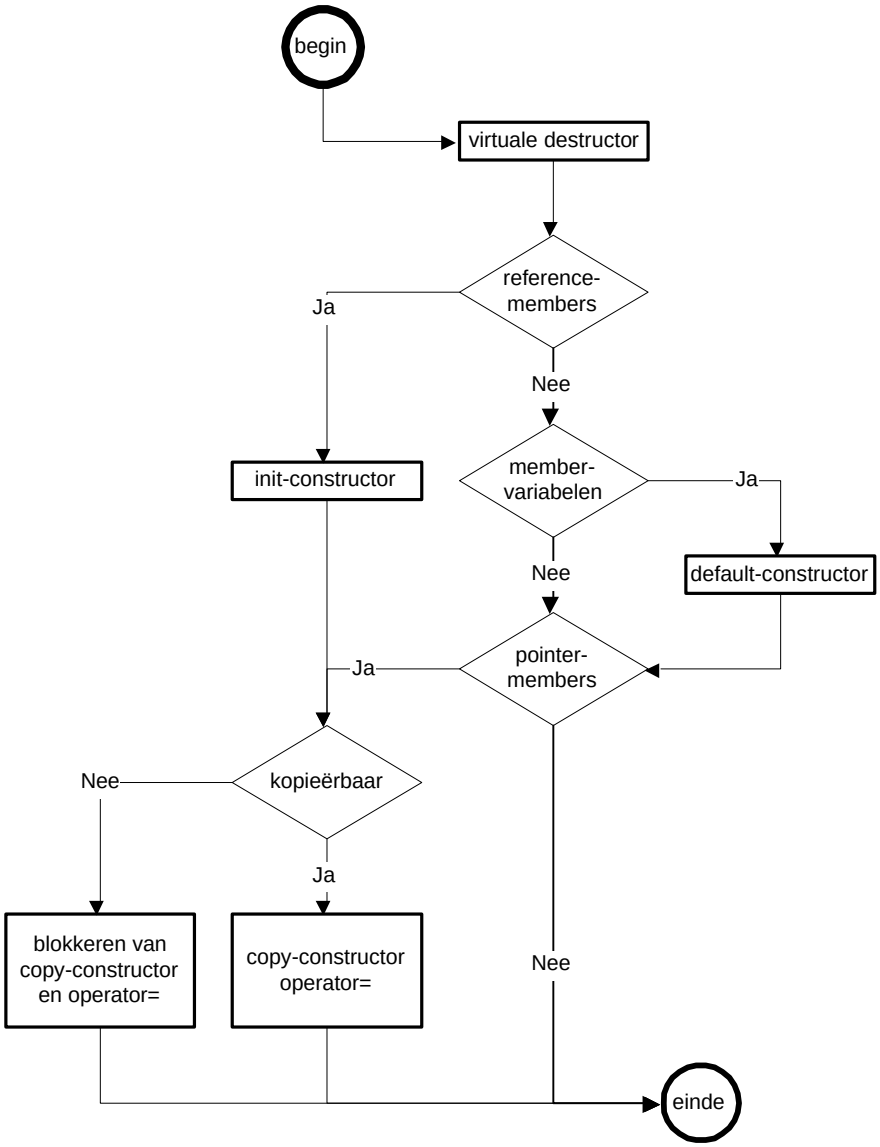
```
    // bla bla
  }

  void Load( const char* filename )
  {
    changed = false;
    // bla bla
  }

  void Save() const
  {
    if (changed)
    {
      changed = false;       // logical const-correct not desired
      // bla bla
    }
  }

private:
  mutable bool changed;
  int widht;
  int height;
  char* buffer;
};
```

```
  // type-cast alternative
  void Save() const
  {
    if (changed)
    {
      const_cast<ImageProxy*>(this)->changed = false;
      // bla bla
    }
  }
```

4.9 Bepalen canonieke class-inhoud

De canonieke class-inhoud is de inhoud die de compiler zelf genereert voor de gegeven lay-out van de class. Om te voorkomen dat de compiler (door gebrek aan semantisch inzicht) verkeerde versies genereerd, maakt de programmeur deze functies zelf.

Elke class in het systeem moet een correcte canonieke vorm hebben.



4.10 Explicit-keyword

Om te voorkomen dat de compiler constructors met één argument (zogenaamde cast-constructors) impliciet of onbedoeld gebruikt voor type-conversies moeten alle constructors met uitzondering van de copy-constructor worden voorzien van het explicit-keyword.

vb.

```
class TypeA;

class TypeB
{
public:
  TypeB();
  TypeB( const TypeB& );           // copy-ctor
  explicit TypeB( const TypeA& );  // cast-ctor
};
```

4.11 Resource-managment

Een resource is elke bron of voorraad die een programma tijdens uitvoering alloceert en dealloceert. Voorbeelden zijn files, geheugen. semaphoren, locks enz. Resource-management is de techniek die correct vrijgeven van de bronnen garandeert.

Het correct omgaan met resources is gekoppeld aan slechts één belangrijke regel:

- De levensduur van een resource moet **altijd** zijn gekoppeld aan één bepaalde 'eigenaar' van de resource. Zodra er geen eigenaar meer is, moet de resource worden opgeruimd, ook wanneer er een exceptie optreedt.

Om dit te bewerkstelligen moet de allocatie en deallocatie van een resource gebeuren in de constructur resp. destructor van een class (de eigenaar of gebruiker). Voorbeeld:

```
class ResourceOwner
{
public:
  ResourceOwner()
  : p( new char[100] )
  {}

  ~ResourceOwner()
  {
   delete [] p;
  }

private:
  char* p;
};
```

Het delen en overgeven van resources moet zorgvuldig (expliciet) worden geregeld om te vermijden dat twee eigenaren dezelfde resource gaan vrijgeven bij beëindiging van gebruik. Voorbeeld:

```
class ResourceOwner
{
public:
  // copy constructor: what to do with resources???
  ResourceOwner(const ResourceOwner& org )
  : p( org.p ) // error, invalid resource sharing
  {}

  ~ResourceOwner()
  {
   delete [] p; // pointer could be deallocated by other object
  }
```

```
private:
  char* p;
};
```

Correcte canonieke members zijn heel belangrijk in dit opzicht.

5. Instrumentatie

5.1 Instrumentatie

Instrumentatie is code die niet tot de primaire functionaliteit behoort (bv. controle-code), het is een soort secundair systeem. Instrumentatie moet aan een paar voorwaarden voldoen:

- Instrumentatie mag *geen* toestandsveranderingen in het de primaire systeem veroorzaken
- Instrumentatie moet altijd conditioneel compileerbaar zijn
- Instrumentatie moet altijd minimaal een eenvoudig zijn.
- Instrumentatie moet duidelijk te herkennen zijn.

Conditionele compilatie wordt als volgt gerealiseerd (praktijk voorbeeld):

```
#ifndef NXDEBUG
#ifdef _MSC_VER
#pragma message ( "x-testing enabled (disable with #define NXDEBUG)" )
#endif
#define XPRECOND(exp)   XErrorHandler::checkPrecond( #exp, int(exp), __LINE__, __FILE__ )
#define XPOSTCOND(exp)  XErrorHandler::checkPostcond( #exp, int(exp), __LINE__, __FILE__ )
#define XINVARIANT(exp) XErrorHandler::checkInvariant( #exp, int(exp), __LINE__, __FILE__ )
#define XASSERT(exp)    XErrorHandler::check( #exp, int(exp), __LINE__, __FILE__ )
#else
#ifdef _MSC_VER
#pragma message ( "x-testing disabled" )
#endif
#define XPRECOND(exp)   ((void)0)
#define XPOSTCOND(exp)  ((void)0)
#define XINVARIANT(exp) ((void)0)
#define XASSERT(exp)    ((void)0)
#endif
```

De conditionele compilatie wordt gestuurd door de **NXDEBUG**-define.

De herkenbaarheid is gewaarborgd doordat alle instrumentatiecode met een **X** begint.

5.2 Correctheid

Een goede methode om correcte modulen te bouwen is “Design-by-contract” van Bertrand Meyer (bedenker van de taal Eiffel). In dit concept hebben de aanbieder en afnemer van een service een ‘formeel’ contract met elkaar dat beiden moeten naleven. Een contract is een duidelijke specificatie van verantwoordelijkheden zodat bij contractbreuk duidelijk blijkt bij wie de fout ligt.

In DBC wordt altijd de volledige specificatie *expliciet* genoteerd. De formele specificatie omvat de volgende onderdelen:

- Precondities

  Een pre-conditie is een voorwaarde die moet gelden bij aanvang van een operatie / service om deze succesvol te kunnen uitvoeren. *De pre-conditie is de verantwoording van een afnemer van de service.*

- Postcondities

  Een post-conditie is een voorwaarde die na uitvoering van een operatie / service moet gelden. *De post-conditie is de verantwoording van de aanbieder van de service.*

- Invariantcondities

  De invariantcondities zijn de voorwaarden die op de inactieve momenten van een object gelden.

Een belangrijk principe van het DBC is dat een het contract *altijd maar door één van de ‘contractuele partners’ wordt gecontroleerd*. Dit in tegenstelling tot bijvoorbeeld ‘defensive programming[2]’. De situatie waarin de aanbieder de condities controleert heet ‘tolerant’, de situatie waarin de afnemer dat doet heet ‘veeleisend (demanding)’.

C++ ondersteunt geen expliciete DBC, wel impliciet, in de vorm van instrumentatie.

Conditiecontroles controleren de pre-, post- en invariantcondities.

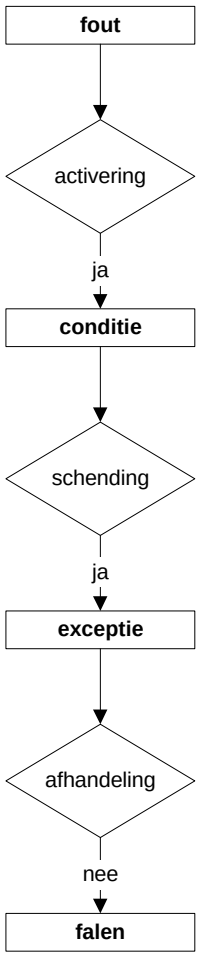
- Alleen de aangeroepen method controleert de condities
- De pre-condities en de invariant-condities worden aan het begin van de method getest

[2] Gevoelsmatig lijkt het logisch om zowel de aanbieder als de afnemer hun pre- én postcondities te laten controleren. In praktijk blijkt dit vaak contraproduktief en in conflict met het doel (correctheid).

- De post-condities en de invariant-condities worden aan het einde van de method getest

5.3 Robuustheid

Robuustheid is betrouwbare werking onder niet-gespecificeerde of onvoorziene omstandigheden. Onvoorziene omstandigheden ontstaan door fouten en kunnen uiteindelijk leiden tot falen. Het onderstaande figuur toont te verschillende transformaties die een fout ondergaat voordat het zich uiteindelijk zal uiten als een falen:



Figuur 7: De fout-falen keten

- Fouten

Een fout is een verkeerde specificatie in analyse, ontwerp of code. Een fout veroorzaakt geen conditie als de foutieve code niet wordt geactiveerd.

Als dat deel van het systeem waar de fout zich bevindt nooit wordt geactiveerd, zal een fout ongemerkt blijven.

- Condities

  Een conditie ontstaat als code waarin zich een fout bevindt daadwerkelijk wordt uitgevoerd. Als de conditie echter door de actuele toestand van het systeem niet wordt geschonden, zal de conditie geen exceptie veroorzaken.

- Exceptie

  Een exceptie is de schending van een conditie. Het programma kan niet voldoen aan de specificaties door de aanwezigheid van een conditie die door de actuele toestand van het systeem wordt geschonden. Een concretisering van een conditie. Een exceptie hoeft nog niet tot falen te leiden.

  Als de exceptie netjes kan worden afgehandeld, dan leidt de exceptie niet tot een falen.

- Falen

  Een programma faalt als een exceptie niet kan worden afgehandeld, bijvoorbeeld door correctie van de conditie die de exceptie veroorzaakte. Falen leidt altijd tot irritatie.

Goede implementaties behandelen condities en excepties als gescheiden concepten. Goede programmeurs proberen eerst fouten te vermijden. Goede testers proberen condities te vinden.

Alle maatregelen die dienen om de propagatie van fouten tot falen te stoppen behoren tot de *foutafhandeling* van een implementatie.

5.4 Foutafhandeling

In het algemeen bestaan er een paar systemen van foutafhandeling:

- Foutfuncties of foutvariabelen

  Bij constatering van een fout wordt er een foutvariabele gezet. De aanroepende functie moet voor elke aangeroepen functie de status controleren door de foutvariabele of -functie te controleren.

- Funny-return values

  Functies retourneren een waarden die buiten het normale bereik vande functie liggen als er een fout is opgetreden.

- Errorhandlers

  Bij constatering van een een fout wordt er een speciale handler aangeroepen.

- Excepties

  Bij constatering van een een fout wordt er een exceptie gegenereerd.

Exceptie-handling is het superieure systeem van foutafhandeling:

- Excepties kunnen niet worden genegeerd
- Normale expressie-notatie mogelijk
- Afhandeling gebeurt altijd binnen de context met de meeste relevante informatie
- Minimale hoeveelheid codeerwerk

6. Review

6.1 C++ Review-controlepunten

Gebruik de codeerstandaard. Let extra op de volgende punten:

- Kwaliteit van class (interface/implementatie, cohesie, koppeling)

  Probeer indien nodig door reverse-engineering het class-diagram te reconstrueren en vergelijk dat met het ontwerp.

- Kwaliteit van module/file (declaratie/definitie)

  Teken indien nodig de include-graaf en vergelijk die met het ontwerp.

- Insulatie

  Controleer of het verschil tussen interface en declaratie niet te groot is. Controleer of de kritieke classes correct geinsuleerd zijn, hetzij door de toepassingen van ontwerpconstructies, hetzij door insulatie.

- Canonieke vormen

  Controleer of alle canonieke vormen aanwezig en correct zijn. Gebruik de eerder gegeven flowchart.

- Const-correctheid

  Controleer de const-correctheid. Kijk of de scheiding selector, modifier strikt is doorgevoerd en of alle selectors const zijn. Controleer of geoptimaliseerde value-argument by-const-reference worden doorgegeven.

- Exception-handling

  Controleer of de constructors een goede exception-handling hebben. Controleer dat de destructors geen exceptions mogen gooien. Controleer dat alle operaties bij het optreden van een exceptie weer hun pre-conditie herstellen.

- Resource-management

Controleer of de alle resources die worden gealloceerd weer correct worden vrijgegeven, ook in het geval van excepties.

- Syntaxis

  Alles moet goed leesbaar zijn en efficient zijn geprogrammeerd

- Instrumentatie

  Controleer of de juiste instrumentatie aanwezig is.

Gebruik de C++-codeerstandaard op www.synalyse.com/cppcodeerstandaard

6.2 Incrementele review

Naar analogie van de incrementale implememtatie, kan de review het beste in meerdere slagen worden uitgevoerd, waarbij tijdens op een ander aspect wordt gelet.:

1 - Het controleren van het 'structurele' raamwerk

2 - Het controleren van het 'semantische' raamwerk

3 - Het controleren van het 'technische' raamwerk

4 - Het controleren van de syntaxis

7. Canonieke vormen

7.1 Canonieke insulator

Een insulator is een class waarvan de fysieke-interface exact dezelfde inhoud heeft als de logische-interface.

(bundel.h)

```
#ifndef BUNDEL_H
#define BUNDEL_H

class BundelIMP;
class Bundel
{
public:
  Bundel();
  Bundel( const Bundel& org );
  Bundel& operator=( const Bundel& rhs );
  virtual ~Bundel();
private:
  BundelIMP*   mImplementor;
};

#endif
```

(bundel.cpp)

```
#ifndef   BUNDEL_H
#include "bundel.h"
#define   BUNDEL_H
#endif

#ifndef   CONTAINER_H
#include "container.h"
#define   CONTAINER_H
#endif

#ifndef   LIST
#include <list>
#define   LIST
#endif

class BundelIMP
{
public:
  BundelIMP();
  BundelIMP( const BundelIMP& org );
  BundelIMP& operator=( const BundelIMP& rhs );
  virtual ~BundelIMP();
};
```

```
// place BundelIMP-methods here or default-inline in
// class-declaration above

Bundel::Bundel ()
: mImplementor( new Bundel IMP )
{}

Bundel::Bundel( const Bundel& org )
: mImplementor( new BundelIMP( *rhs.mImplementor ) )
// note use of copy-ctor of 'org' here
{}

Bundel& Bundel::operator=( const Bundel& rhs )
{
  if ( &rhs!=this )          // note self-protection
  {
    delete mImplementor;
    mImplementor = 0;        // set to 0 to be exception-proof
    mImplementor = new BundelIMP( *rhs.mImplementor );
                                    // note use of copy-ctor here
  }
  return *this;
}

Bundel::~Bundel()
{
  delete mImplementor;
}
```

7.2 Canonieke vorm canonieke class type-I (geen member-variabelen)

(declaratie)

```
#ifndef SJABLOONCLASS_H
#define SJABLOONCLASS_H

class CSjabloonClass
{
public:
  // constanten
  static const char* const C_TEXT_CONSTANT;

  // dtor
  virtual ~CSjabloonClass();
};

#endif  // SJABLOONCLASS_H
```

(definitie)

```
#ifndef   SJABLOONCLASS_H
#include "sjablooonclass.h"
#define   SJABLOONCLASS_H
#endif

// constanten
const char* const CSjabloonClass::C_TEXT_CONSTANT = "TEXT";

CSjabloonClass::~CSjabloonClass()
{
  // bla bla
}
```

7.3 Canonieke vorm canonieke class type-II (slechts value member-variabelen)

(declaratie)

```
#ifndef SJABLOONCLASS_H
#define SJABLOONCLASS_H

class CSjabloonClass
{
public:
  // ctor's
  CSjabloonClass();
  CSjabloonClass( const CSjabloonClass& org );
  CSjabloonClass& operator=( const CSjabloonClass& rhs );

  // dtor
  virtual ~CSjabloonClass();

  // selectors
  double selector() const;

  // modifiers
  void modifier();
  void modifier( const CSjabloonClass& copy );

  // operators
  bool operator==( const CSjabloonClass& rhs );

private:
  double attribute;
};

#endif  // SJABLOONCLASS_H
```

(definitie)

```
#ifndef   SJABLOONCLASS_H
#include "sjabloonclass.h"
#define   SJABLOONCLASS_H
#endif

#ifndef   ASSERT_H
#include <assert.h>
#define   ASSERT_H
#endif

CSjabloonClass::CSjabloonClass()
: attribute( 0.0 )
{
  // bla bla
}
CSjabloonClass::CSjabloonClass(  const CSjabloonClass& org )
: attribute( org.attribute )
{
  // bla bla
}
CSjabloonClass& operator=( const CSjabloonClass& rhs )
{
  if (&rhs!=this)
  {
    attribute = rhs.attribute
  }
  return *this;
```

```
}
CSjabloonClass::~CSjabloonClass()
{
  // bla bla
}
double selector() const
{
  return attribute;
}
void CSjabloonClass::modifier ()
{
  // bla bla
}
void CSjabloonClass::modifier ( const CSjabloonClass& copy )
{
  assert(&copy!=this);
  // bla bla
}
bool CSjabloonClass::operator==(  const CSjabloonClass& copy )
{
  // bla bla
}
```

7.4 Canonieke vorm canonieke class type-III (pointer-membervariabelen)

(declaratie)

```
#ifndef SJABLOONCLASS_H
#define SJABLOONCLASS_H
class CReferenceClass;

class CSjabloonClass
{
public:
  // ctor's
  CSjabloonClass();
  CSjabloonClass( const CSjabloonClass& org );
  CSjabloonClass& operator=( const CSjabloonClass& rhs );

  // dtor
  virtual ~CSjabloonClass();

  // selectors
  const CReferenceClass& selector() const;

  // modifiers
  void modifier();
  void modifier( const CSjabloonClass& copy );

  // operators
  bool operator==( const CSjabloonClass& rhs );

private:
  CReferenceClass* attribute;
};

#endif  // SJABLOONCLASS_H
```

(definitie)

```
#ifndef   SJABLOONCLASS_H
#include "sjabloonclass.h"
#define   SJABLOONCLASS_H
```

```
#endif

#ifndef   REFERENCECLASS_H
#include "referenceclass.h"
#define   REFERENCECLASS_H
#endif

#ifndef   ASSERT_H
#include <assert.h>
#define   ASSERT_H
#endif

CSjabloonClass::CSjabloonClass()
: attribute( new CReferenceClass )
{
  // bla bla
}
CSjabloonClass::CSjabloonClass(  const CSjabloonClass& org )
: attribute( new CReferenceClass( org.attribute ) )
{
  // bla bla
}
CSjabloonClass& operator=( const CSjabloonClass& rhs )
{
  if (&rhs!=this)
  {
    delete attribute;
    attribute = 0;
    attribute = new CReferenceClass( rhs.attribute );
  }
  return *this;
}
CSjabloonClass::~CSjabloonClass()
{
  delete attribute;
}
const CReferenceClass& selector() const
{
  return *attribute;
}
void CSjabloonClass::modifier ()
{
  // bla bla
}
void CSjabloonClass::modifier ( const CSjabloonClass& copy )
{
  assert(&copy!=this);
  // bla bla
}
bool CSjabloonClass::operator==(  const CSjabloonClass& copy )
{
  // bla bla
}
```

7.5 Canonieke vorm canonieke class type-IV (reference-membervariabelen)

(declaratie)

```
#ifndef SJABLOONCLASS_H
#define SJABLOONCLASS_H
```

```
class ReferenceClass;

class CSjabloonClass
{
public:
  // ctor's
  CSjabloonClass( const CSjabloonClass& org );
  CSjabloonClass( const CReferenceClass& ref );
  CSjabloonClass& operator=( const CSjabloonClass& rhs );

  // dtor
  virtual ~CSjabloonClass();

  // selectors
  const CReferenceClass& selector() const;

  // modifiers
  void modifier();
  void modifier( const CSjabloonClass& copy );

  // operators
  bool operator==( const CSjabloonClass& rhs );

private:
  CReferenceClass& attribute;
};

#endif  // SJABLOONCLASS_H
```

(definitie)

```
#ifndef   SJABLOONCLASS_H
#include "sjabloonclass.h"
#define   SJABLOONCLASS_H
#endif

#ifndef   REFERENCECLASS_H
#include "referenceclass.h"
#define   REFERENCECLASS_H
#endif

#ifndef   ASSERT_H
#include <exception.h>
#define   ASSERT_H
#endif

CSjabloonClass::CSjabloonClass( const CReferenceClass& ref)
: attribute( ref )
{
  // bla bla
}
CSjabloonClass::CSjabloonClass(  const CSjabloonClass& org )
: attribute( org.attribute )
{
  // bla bla
}
CSjabloonClass& operator=( const CSjabloonClass& rhs )
{
  if (&rhs!=this)
  {
    attribute = rhs.attribute
  }
  return *this;
}
CSjabloonClass::~CSjabloonClass()
{
```

```
  // bla bla
}
const ReferenceClass& selector() const
{
  return attribute;
}
void CSjabloonClass::modifier ()
{
  // bla bla
}
void CSjabloonClass::modifier ( const CSjabloonClass& copy )
{
  assert(&copy!=this);
  // bla bla
}
bool CSjabloonClass::operator==(  const CSjabloonClass& copy )
{
  // bla bla
}
```

7.6 Canonieke functies van class met enkelvoudige allocatie

Let op:

- Het gebruik van de copy-constructoren van het geaggegreerde object vanuit de eigen copy-constructor.
- Het op 0 zetten van pointers na het aanroepen van delete om de destructor te behoeden voor ongeldige de-allocaties.

```
class X
{
 // bla bla

private:
  Y* y;
};

//------------------------------

X::X()
  : y( new Y() )
{}

X::X( const X& org )
  : y( new Y( *org.y ) )     // note use of copy-ctor of 'org'
{}

X::~X()
{
  delete y;
}

X& X::operator=( const X& rhs )
{
  if (this!=&rhs)                    // note self-protection
  {
    delete y;
    y = 0;                           // note setting 'y' to 0
```

```
    y = new Y( *rhs.y );      // note use of copy-ctor of 'rhs'
  }
  return *this;
}
```

7.7 Canonieke functies van class met meervoudige allocatie

Let op:

- Het gebruik van de copy-constructoren van de geaggregeerde objecten vanuit de eigen copy-constructor en de `operator=`.
- De `init`-/`term`-functies voor het exceptie-veilig maken van de constructors.
- De zelfbescherming van de `operator=`.
- Het op 0 zetten van pointers na het aanroepen van delete om de destructor te behoeden voor ongeldige de-allocaties.

```
class X
{
  // bla bla

private:
  Y* y;
  Z* z;
  void init(Y* org_y = 0, Z* org_z = 0);
  void term();
};

// --------------------------------

X::X()
  : y( 0 )
  , z( 0 )
{
  init();
}

X::X( const X& org )
  : y( 0 )
  , z( 0 )
{
  init( org.y, org.z );
}

X::~X()
{
  term();
}

X& X::operator=( const X& rhs )
{
  if (this!=&rhs)                    // self-protection
  {
    term();
```

```
    init( rhs.y, rhs.z );
  }
  return *this;
}

void X::init( Y* org_y, Z* org_z )
{
  try
  {
    if (0!=org_y)
    {
      y = new Y( *org_y );   // note use of copy-ctor of 'org'
    }
    else
    {
      y = new Y();
    }
    if (0!=org_z)
    {
      z = new Z( *org_z );   // note use of copy-ctor of 'org'
    }
    else
    {
      z = new Z();
    }
  }
  catch(...)
  {
    term();
    throw;
  }
}

void X::term()
{
  delete y;
  delete z;
  y = 0;
  z = 0;
}
```

7.8 Canonieke functies van afgeleide class

Let op:

- 'Decoratie' van **`operator=`** van base-class.

```
class X : public class Y
{
  // bla bla
private:
  int z;
};


X::X()
  : z( 0 )
{}
```

```
X::X( const X& org )
 : Y( X )
 , z( org.z )
{}

X::~X()
{}

X& X::operator=( const X& rhs )
{
  if (this!=rhs)                    // note self-protection
  {
    Y::operator=( rhs );     // note use of base-class operator
    z = rhs.z;
  }
  return *this;
}
```