SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2018

13 Module Decomposition (Ghezzi Ch. 4, H&S Ch. 7)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

February 2, 2018

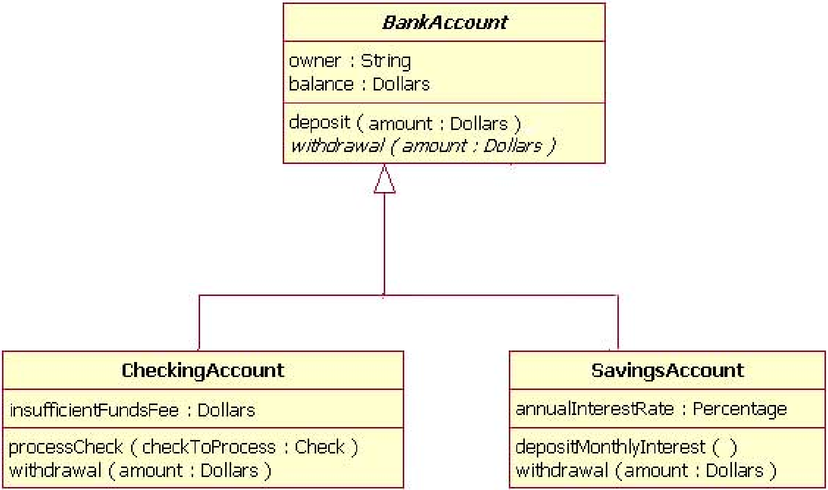
13 Module Decomposition (Ghezzi Ch. 4, H&S Ch. 7)

- Administrative details
- Finish OOD
- Exceptions and assumptions
- Quality criteria
- Module decomposition
- Software architecture
- Design for change
- Relationship between modules
- The USES relation
- Module decomposition by secrets
- The IS_COMPONENT_OF relation
- Techniques for design for change
- Module guide

Administrative Details

- Assignment 2 (Still in Draft Form)
  - Part 1: February 12, 2018
  - Partner Files: February 18, 2018
  - Part 2: March 2, 2018
- Midterm exam
  - Wednesday, February 28, 7:00 pm
  - 90 minute duration
  - Multiple choice - 30–40 questions

Bank Account Example



Class Diagram Versus MIS

- What information do the MIS and Class Diagram have in common?
- What information does the MIS add?
- What information does the Class Diagram add?

Class diagrams are closer to code since syntax of methods closer to actual syntax

Class Diagram Versus MIS

- What information do the MIS and Class Diagram have in common?
- What information does the MIS add?
- What information does the Class Diagram add?

Class diagrams are closer to code since syntax of methods closer to actual syntax

Class Diagram Versus MIS

- What information do the MIS and Class Diagram have in common?
- What information does the MIS add?
- What information does the Class Diagram add?

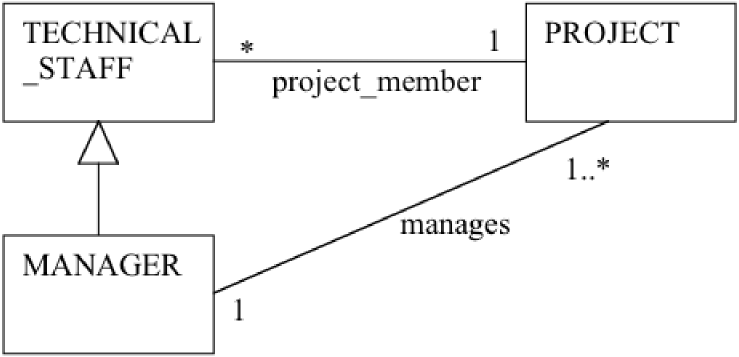
Class diagrams are closer to code since syntax of methods closer to actual syntax

Showing Exceptions in UML Class Diagrams

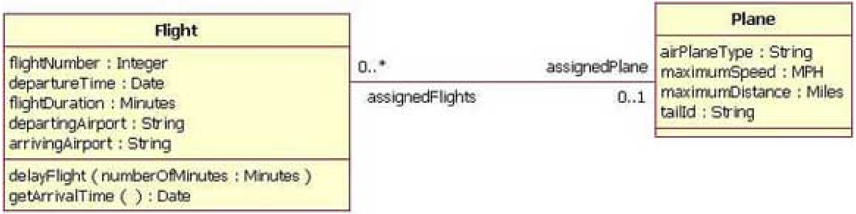
- Usually exceptions are not shown
- If they are, it is in brackets after the method name
- + findAllInstances(): Vector
  {exceptions=NetworkFailure, DatabaseError}

UML Associations

- Associations are relations that the implementation is required to support
- Can have multiplicity constraints



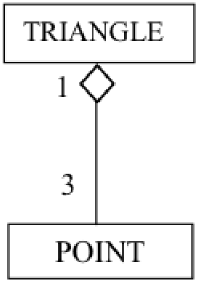
Flight Example



From IBM

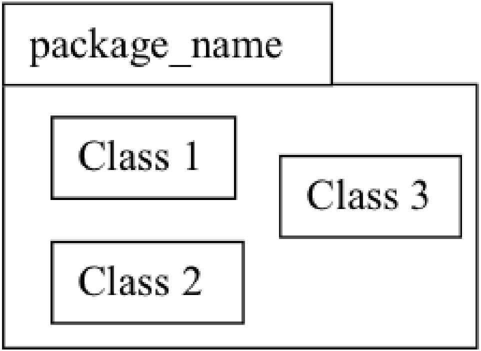
UML Aggregation

- Defines a PART_OF relation
- Differs from IS_COMPONENT_OF
- TRIANGLE has its own methods
- TRIANGLE implicitly uses POINT to define its data attributes



UML Packages

IS_COMPONENT_OF is represented via the package notation



Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Point ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointT	real, real	PointT	
xcoord		real	
ycoord		real	
dist	PointT	real	

Semantics

State Variables

xc: real
yc: real

Point Mass ADT Module

Template Module

PointMassT **inherits** PointT

Uses

PointT

Syntax

Exported Types

PointMassT = ?

Point Mass ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointMassT	real, real, real	PointMassT	NegMassExcept
mval		real	
force	PointMassT	real	
fx	PointMassT	real	

Semantics

State Variables

ms: real

Point Mass ADT Module Semantics

new PointMassT(x, y, m):

- transition: $xc, yc, ms := x, y, m$
- output: $out := self$
- exception: $exc := (m < 0 \Rightarrow \text{NegMassExcept})$

force(p):

- output:

$$out := \text{UNIVERAL\_G} \frac{self.ms \times p.ms}{self.\text{dist}(p)^2}$$

- exception: none

Assumptions versus Exceptions

- The assumptions section lists assumptions the module developer is permitted to make about the programmer's behaviour
- Assumptions are expressed in prose
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation
- Interface design should provide the programmer with a means to check so that they can avoid exceptions
- When an exceptions occurs no state transitions should take place, any output is *don't care*

Exception Signaling

- Useful to think about exceptions in the design process
- Will need to decide how exception signalling will be done
  - A special return value, a special status parameter, a global variable
  - Invoking an exception procedure
  - Using built-in language constructs
- Caused by errors made by programmers, not by users
- Write code so that it avoids exceptions
- Exceptions will be particularly useful during testing

Example Subclass Exception in Python

```
class Full(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return str(self.value)
```

Example of raising the exception

```
    if size == Seq.MAX_SIZE:
      raise Full("Maximum size exceeded")
```

Quality Criteria (H&S Section 7.3.2)

- Consistent
  - Name conventions
  - Ordering of parameters in argument lists
  - Exception handling, etc.
- Essential - omit unnecessary features
- General - cannot always predict how the module will be used
- As implementation independent as possible
- Minimal - avoid access routines with two potentially independent services
- High cohesion - components are closely related
- Low coupling - not strongly dependent on other modules
- Opaque - information hiding
- Checks available so programmer can avoid exceptions

Queue Module Syntax (Abstract Object)

What could we remove to make this essential?

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
q_init		queueT	
add	T		NOT_INIT, FULL
pop			NOT_INIT, EMPTY
front		T	NOT_INIT, EMPTY
size		integer	NOT_INIT
isempty		boolean	NOT_INIT
isfull		boolean	NOT_INIT

Queue Module Syntax (Abstract Object)

What could we remove to make this essential?

MAX_SIZE = 100

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
q_init		queueT	
add	T		NOT_INIT, FULL
pop			NOT_INIT, EMPTY
front		T	NOT_INIT, EMPTY
size		integer	NOT_INIT
isempty		boolean	NOT_INIT
isfull		boolean	NOT_INIT

Can replace isempty and isfull by by tests using size and MAX_SIZE

Queue Module Syntax (Abstract Object)

Is this interface minimal?

Exported Access Programs

Routine name	In	Out	Exceptions
q_init		queueT	
add	T		NOT_INIT, FULL
pop		T	NOT_INIT, EMPTY
size		integer	NOT_INIT
isinit		boolean	

- front has been merged with pop
- size replaces isempty and isfull
- isinit is added (why?)

Modular Decomposition

- Until now our focus has been on individual modules, but how do we decompose a large software system into modules?
- We need to decompose the system into modules, assign responsibilities to those modules and ensure that they fit together to achieve our global goals
- We need to produce a software architecture
- The architecture (modular decomposition) is summarized in a Software Design Document

Software Architecture

- Shows gross structure and organization of the system to be defined
- Its description includes the description of
  - Main components of the system
  - Relationship among those components
  - Rationale for decomposition into its components
  - Constraints that must be respected by any design of the components
- Guides the development of the design

Specific Techniques for Design for Change

What technique/tool would you use if you wanted to select at build time between two implementations of a module, each distinguished by a different decision for their shared secret?

Specific Techniques for Design for Change

- Anticipate definition of all family members
- Identify what is common to all family members, delay decisions that differentiate among different members
- Configuration constants
  - Factor constant values into symbolic constants
  - Compile time binding
  - MAXSPEED = 5600
- Conditional compilation
  - Compile time binding
  - Works well when there is a preprocessor, like for C
  - If performance is not a concern, can often "fake it" at run time
- Make
- Software generation
  - Compiler generator, like `yacc`
  - Domain Specific Language

Questions

- What relationships have we discussed between modules?
- Are there desirable properties for these relations?

Relations Between Modules

- Uses
- Inheritance
- Association
- Aggregation
- IS_COMPONENT_OF
- etc.

Relationships Between Modules

- Let S be a set of modules

$$S = \{M_1, M_2, ..., M_n\}$$

- A binary relation r on S is a subset of $S \times S$
- If M_i and M_j are in S, $< M_i, M_j > \in r$ can be written as $M_i r M_j$

Relations

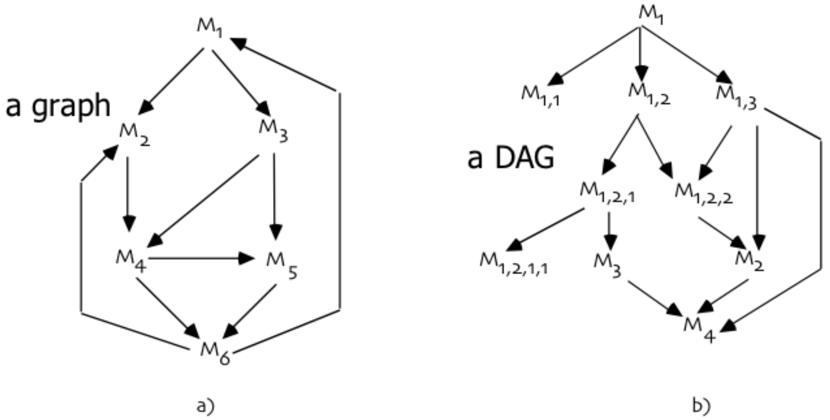
- Transitive closure r^+ of r

  $M_i r^+ M_j$ iff $M_i r M_j$ or $\exists M_k$ in S such that $M_i r M_k$ and $M_k r^+ M_j$

- r is a hierarchy iff there are no two elements M_i, M_j such that $M_i r^+ M_j \wedge M_j r^+ M_i$

Relations Continued

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)



Why do we prefer the uses relation to be a DAG?

References

- Parnas, David L, Software Fundamentals: collected papers by David L. Parnas, edited by Daniel M. Hoffmann and David M. Weiss, Lucent Technologies and Daniel M. Hoffmann, 2001, ISBN 0-201-70369-6
- Parnas, D. L., "On a 'Buzzword': Hierarchical Structure", IFIP Congress 74, North Holland Publishing Company, 1974, pp. 336–339
- Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, 15, 12, December 1972, pp. 1053–1058.

References Continued

- Parnas, D. L., "Designing Software for Ease of Extension and Contraction", Copyright 1979, IEEE Transaction on Software Engineering, March 1979, pp. 128–138,
- Parnas, D. L., Clements, P. C., Weiss, D. M., "The Modular Structure of Complex Systems", IEEE Transaction on Software Engineering, March 1985, Vol SE-11, No. 3, pp. 259-266 (special issue on the 7th International Conference on Software Engineering)

References Continued

- Parnas, D. L., Clements, P. C., “A Rational Design Process: How and Why to Fake it”, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 251-257.
- Parnas, On the design and development of program families, IEEE Transactions on Software Engineering, SE-2(1), March 1976.
- Hoffmann, Daniel, M., and Paul A. Strooper, Software Design, Automated Testing, and Maintenance A Practical Approach, International Thomson Computer Press, 1995, http://citeseer.ist.psu.edu/428727.html

References Continued

- Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, 1972 (modular decomposition)
- ElSheikh, Ahmed, W. Spencer Smith, and Samir E. Chidiac. (2004) Semi-formal design of reliable mesh generation systems. Advances in Engineering Software, Vol 35, Issue 12, pp 827-841.
- Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli, Fundamentals of Software Engineering, 2nd Ed., Prentice Hall, 2003

References Continued

- Dijkstra, The structure of THE multiprogramming system. Communications of the ACM, 11(5): 341-346, May 1968.
- Linger, Mills and Witt. Structured Programming: Theory and Practice, Addison-Wesley, 1979 (step-wise refinement)
- Wirth, Program development by stepwise refinement, Communications of the ACM, 14(4):221-227, April 1971.