Software Engineering

- Exercises 2023 -

References

[LL05] T. Lethbridge, R. Laganiere. *Object Oriented Software Engineering Practical Software Development using UML and Java* (1st, 2nd editions). MgGraw Hill, (2001, 2005).

[FP20] A. Fox, D. Patterson. *Engineering Software as a Service: An Agile Approach using Cloud Computing*. (1st, 2nd editions) Strawberry Canyon LLC, (2016, 2020). http://www.saasbook.info/

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[BW90] J.C. Baeten, W.P. Wijland. *Process algebra*. Cambridge Univ. Press, 1990.

[BB02] W. Boggs, M. Boggs. *Mastering UML with Rational Rose* (2nd edition). Sybex, 2002.

[BK08] C. Baier, J.-P. Katoen, *Principles of Model Checking*, MIT Press, 2008.

[CHVB18] E.M. Clarke, T.A. Henzinger, H. Veith and R. Bloem, Editors, *Handbook of Model Checking*, Springer, 2018.

[GoF94] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Nik16] I. Nikolov, *Scala Design Patterns - Write efficient, clean, and reusable code with Scala*, Packt Publishing, 2016.

[Pie23] B. Pierce et al, *Software Foundations*, 2023. https://softwarefoundations.cis.upenn.edu/

[RS13] D. Rosenberg, M. Stephens. *Use Case Driven Object Modeling with UML: Theory and Practice*, 2nd edition, Apress, 2013.

[Som16] I. Sommerville, *Software Engineering,* (10th edition). Addison-Wesley, 2016.

[IBM07] http://www-306.ibm.com/software/rational/

[Haskell] Haskell web page, www.haskell.org

[PRISM] PRISM model checker, http://www.prismmodelchecker.org

[PRISM-tut] PRISM tutorial, available from: http://www.prismmodelchecker.org/tutorial/ http://www.prismmodelchecker.org/courses/pmc1112/

[PRISM-sem] The PRISM language –semantics, available from: http://www.prismmodelchecker.org/doc/semantics.pdf

Developing requirements - Exercise 1

- (Source: [LL05], ch. 4) Draw an UML use case diagram to specify the requirements for the library application described in [LL05] Example 4.9

PRISM case study - Exercise 2

- Source: Dynamic Power Management (DPM) case study available from PRISM tutorial
  http://www.prismmodelchecker.org/tutorial/
  http://www.prismmodelchecker.org/courses/pmc1112/
- Study the (P, S and R) operators of the PRISM property specification language (described in the course lecture slides and the PRISM manual http://www.prismmodelchecker.org/manual/).
  - Analyze the behavior of DPM system by following the steps in the tutorial (see part 3 of the PRISM tutorial). This involves creating PRISM properties and may involve developing appropriate reward structures.

PRISM semantics - Exercise 3

The following PRISM code describes a DTMC:

```
dtmc
module M1
  v1 : [0..1] init 0;
  []  v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1);
  [a] v1=0 & v2=1 -> 1:(v1'=1);
  [b] v1=1 -> 1:true;
endmodule
module M2
  v2 : [0..1] init 0;
  []  v1=0 & v2=0 -> 0.7:(v2'=0) + 0.3:(v2'=1);
  [a] v1=0 & v2=1 -> 1:true;
  [b] v1=1 -> 1:true;
endmodule
```

[1] The PRISM language semantics, available from: http://www.prismmodelchecker.org/doc/semantics.pdf

Define the semantics of this PRISM model following [1]

a) Construct the system module from the component modules
b) Give the semantics of the system module as a transition probability matrix $P: S \times S \rightarrow [0,1]$

- We recall that a (discrete) probability distribution over a countable set S is a function $\mu: S \rightarrow [0,1]$ satisfying
$$\sum_{s \in S} \mu(s)=1$$
- We use $[s_0 \rightarrow p_0, \ldots, s_n \rightarrow p_n]$ to denote the distribution that chooses s_i with probability p_i (for $1 \leq i \leq n$) and Dist(S) for the set of distributions over S

PRISM semantics - Exercise 4

The following PRISM code describes a CTMC:

```
ctmc
module M1
v1 : [0..1] init 0;
[] v1=0 & v2=0 -> 4.5:(v1'=0) + 0.5:(v1'=1);
[a] v1=0 & v2=1 -> 1:(v1'=1);
[b] v1=1 -> 1:true;
endmodule
module M2
v2 : [0..1] init 0;
[] v1=0 & v2=0 -> 3.5:(v2'=0) + 1.5:(v2'=1);
[a] v1=0 & v2=1 -> 1:true;
[b] v1=1 -> 2:true;
endmodule
```

[1] The PRISM language semantics, available from: http://www.prismmodelchecker.org/doc/semantics.pdf

Define the semantics of this PRISM model following [1]

a) Construct the system module from the component modules

b) Give the semantics of the system module as a transition rate matrix $R: S \times S \rightarrow \mathbb{R}_{\geq 0}$

PRISM semantics - Exercise 5

The following PRISM code describes a MDP:

```
mdp
module M1
  v1 : [0..1] init 0;
  []  v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1);
  [a] v1=0 & v2=1 -> 1:(v1'=1);
  [b] v1=1 -> 1:true;
endmodule
module M2
  v2 : [0..1] init 0;
  []  v1=0 & v2=0 -> 0.7:(v2'=0) + 0.3:(v2'=1);
  [a] v1=0 & v2=1 -> 1:true;
  [b] v1=1 -> 1:true;
endmodule
```

[1] The PRISM language semantics, available from: http://www.prismmodelchecker.org/doc/semantics.pdf

Define the semantics of this PRISM model following [1]

a) Construct the system module from the component modules

b) Give the semantics of the system module as the function (given in [1]):

$$\text{Steps} : S \rightarrow 2^{\text{Dist}(S)}$$

- S is the set of states, $2^{\text{Dist}(S)}$ =power set of Dist(S), i.e., the set of all subsets of Dist(S)

c) Give the semantics of the system module when Steps: $S \rightarrow 2^{(\text{Act} \cup \{\wedge\}) \times \text{Dist}(S)}$ is (re)defined as follows:

$$\text{Steps}(s) = \{(\alpha , \mu_{c,s}) \mid c \in C,\ s \in S\}$$

- ($\alpha \in$) Act $\cup$ {^}; Act = set of action labels, ^$\not\in$Act (^ is not an element of Act)
- α=a, if c = [a] $g \rightarrow \lambda_1:u_1+\ldots+\lambda_n:u_n$
- α=^, if c = [] $g \rightarrow \lambda_1:u_1+\ldots+\lambda_n:u_n$
- $\mu_{c,s}$: Dist(S) - $\mu_{c,s}$ are given in [1]

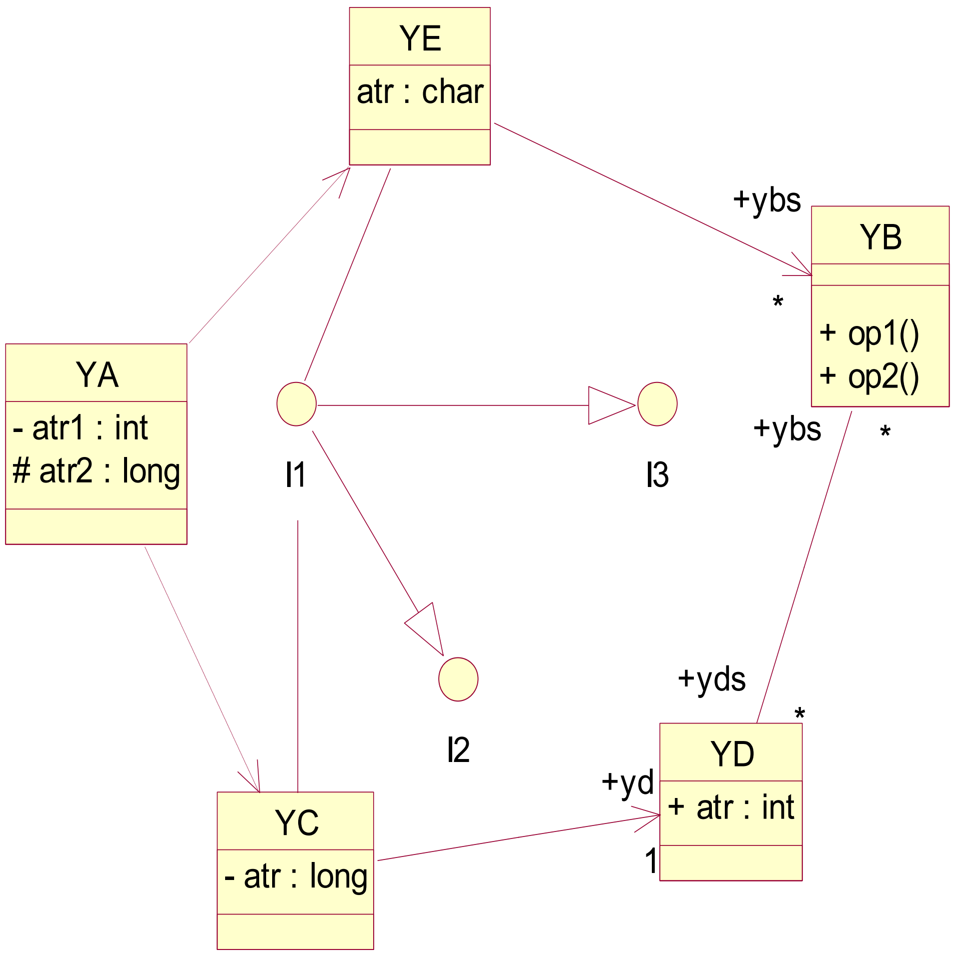
Modeling with classes - Exercise 6

- (Source: [LL05], E91) Draw a class diagram corresponding to the object diagram shown in [LL05] Fig. 5.18

Modeling with classes - Exercise 7

- (Source: [LL05], ch.5) Consider the description of the Airline system described in [LL05] ch.5 (and the course lecture slides)
- Analyze and design the system using UML class diagrams (apply the methodology given in [LL05] ch.5 ”Modeling with classes”)

Forward and reverse engineering
Forward Engineering (UML -> Java) – Exercise 8



Forward and reverse engineering
Reverse Engineering (Java -> UML) – Exercise 9

```
class F extends G implements I3 {
  private K[] ks;
  int i;
  public K k;
  protected H h;
  F() {}
}

class G {
  public G() {
  }
  public void f(H h) {}
}

class H {
  public H() {
  }
}
```

```
class K {
  private F f;
  public K() {
  }
  void g() { H h = new H(); }
}

class L {
  public static void main(String args[]) {
    H h = new H();
    G g = new G();
    F f = new F();
    System.out.println("Test");
  }
}

interface I1 {
}

interface I2 extends I1,I3 {
}

interface I3 {
}
```

Design patterns – Exercise 10

- Exercise: Use the General Hierarchy (or Composite) design pattern to model the Lisp data type, which can be specified using Haskell algebraic data types as follows:

```
data Lisp = Nil | Number Int | Symbol String | Cons Lisp Lisp
```

Design patterns – Exercise 11

- [Source: [LL05], ch.6, Exercise E113] By using the Abstraction-Occurrence design pattern develop a class diagram describing:
  - The issues of a periodical
  - The copies of the issues of a periodical

- [Source: [LL05], ch. 6] Consider the list of design patterns presented in [LL05] Chapter 6 ("Using Design Patterns") : Abstraction-Occurrence, Player-Role, General Hierarchy, Singleton, Observer, Delegation, Façade, Immutable, Read-Only Interface and Proxy. Solve Exercise E128 from [LL05] by determining the design patterns (from the above list) that would apply in specific circumstances.

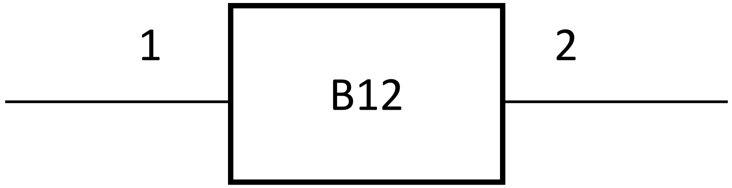
Design patterns – Exercise 12

- Using the General Hierarchy (or the Composite) pattern create a class diagram to represent PCTL (Probabilistic Computation Tree Logic) expressions (PCTL is a probabilistic extension of the temporal logic CTL). The syntax of PCTL is given below in BNF (p = probability, a = atomic proposition):

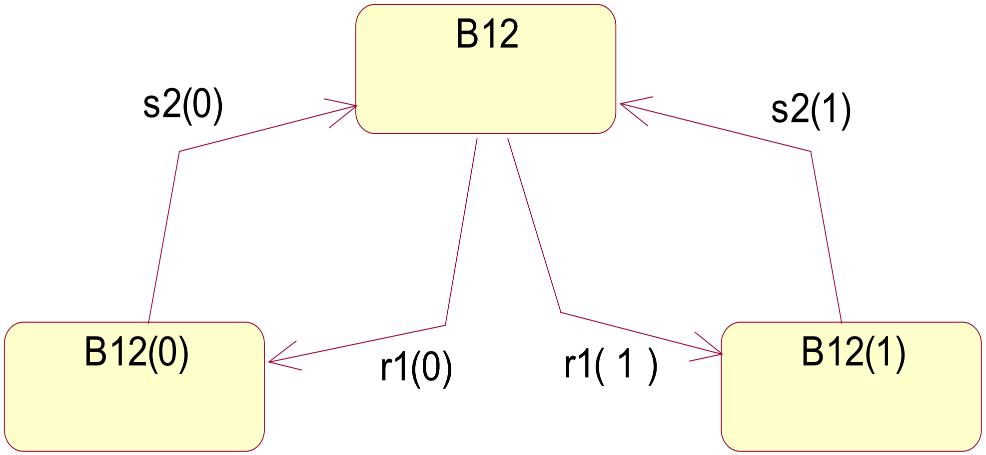
Φ ::= true | a | $\neg$ Φ | $\Phi \wedge \Phi$ | $P{\sim}p[\Psi]$
Ψ ::= X Φ | Φ $U{\leq}k$ Φ

Modeling interaction and behavior – Exercise 13

- Consider a **bit buffer** B12 of **capacity one** with ports 1 (input) and 2 (output):

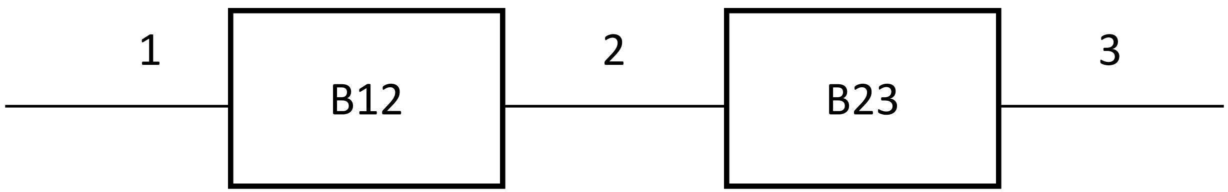


- The behavior of B12 is described by the following STD, where:
  - rj(b) = receive bit b at port j
  - sj(b) = send bit b at port j
  - B12 = buffer empty
  - B12(b) = buffer full (b = 0,1)



Modeling interaction and behavior – Exercise 13

- Draw a STD to model the behavior of two buffers B12 and B23 connected at port 2: B12 can send data at port 2, this data is received by B23.
  - By convention, the simultaneous execution of a pair of actions sj(b) (send bit b at port j) and rj(b) (receive bit b at port j) results in a transmission of b by a communication at port j: cj(b)



- Use the resulted STD to show that two one bit buffers connected in this way behave as a buffer of capacity two [BW90].

Modeling interaction and behavior – Exercise 14

(Source: [LL05], ch. 8) Implement the class CourseSection in Java based on the class diagram given in [LL05] Fig. 8.2 and the state transition diagram given in [LL05] Fig. 8.14

Architecting & Designing Software – Exercise 15

- (Source: [LL05], ch. 9) A software design should lead to modules that exhibit high cohesion and low coupling. Solve exercises E172 and E176 (from book [LL05]), which require you to categorize certain design aspects according to the types of cohesion or coupling of the modules involved.

Testing & Inspecting to Ensure High Quality – Exercise 16

- (Source: [LL05], ch. 10) Solve exercise E193 (from book [LL05]) which refers to the use of terms *failure*, *defect* and *error* (according to the definitions given in the book [LL05] and the course lecture slides)

- (Source: [LL05], ch. 10, excerpt from Exercise E195) Describe a good set of equivalence class test cases for an information form that asks for the current *day* using the format *dd* (assuming that the current year and the current month are known)

Use case driven development – Exercise 17

Employ the use-case driven methodology in the Unified Process [JBR99] in developing a software solution for an automated teller machine system providing the following services: Withdraw Money, Deposit Money, Transfer between Accounts, View Balance. Develop complete UML models (specification, analysis, design, implementation and test) for one of the following services:

1. Withdraw Money
2. Deposit Money
3. View Balance

Declarative prototyping – Exercise 18

Consider the following Haskell prototype (quicksort with difference lists):

```
qsd :: [Int] -> [Int] -> [Int]
qsd []       ys = ys
qsd (x : xs) ys =
   let (ls,gs) = partit x xs
   in qsd ls (x : qsd gs ys)

partit :: Int -> [Int] -> ([Int],[Int])
partit p []     = ([],[])
partit p (x:xs) =
   let (ls,gs) = partit p xs
   in  if (x < p) then (x : ls,gs) else (ls,x:gs)

Main> qsd [1,7,3,12,4,2,5] [0,0,0]
[1,2,3,4,5,7,12,0,0,0]
```

1. Prove the correctness of the prototype by mathematical induction
2. Implement the prototype by using an imperative language (e.g. C)

Declarative prototyping – Exercise 19

Consider the following Haskell prototype (binary search tree merging using difference lists):

```
data Tree = NIL | T Tree Int Tree
flatt :: Tree -> [Int] -> [Int]
flatt NIL       ys = ys
flatt (T l n r) ys = flatt l (n : flatt r ys)

merge :: Tree -> Tree -> [Int] -> [Int]
merge NIL t ys = flatt t ys
merge t NIL ys = flatt t ys
merge (T NIL n1 r1) (T NIL n2 r2) ys =
   if (n1 < n2) then n1:merge r1 (T NIL n2 r2) ys
                else n2:merge (T NIL n1 r1) r2 ys
merge (T (T ll1 ln1 lr1) n1 r1) t2 ys =
   merge (T ll1 ln1 (T lr1 n1 r1)) t2 ys
merge t1 (T (T ll2 ln2 lr2) n2 r2) ys =
   merge t1 (T ll2 ln2 (T lr2 n2 r2)) ys

t1 = T (T NIL 1 (T NIL 3 NIL)) 5 (T NIL 7 NIL)
t2 = T (T NIL 2 NIL) 4 (T (T NIL 6 NIL) 8 NIL)
Main> merge t1 t2 [0,0]
[1,2,3,4,5,6,7,8,0,0]
```

1. Prove the correctness of the prototype by mathematical induction
2. Implement the prototype by using an imperative language (e.g. C)