



Universiteit Leiden

Opleiding Informatica

Learning software design: Is abstraction ability key?

Name: Claire E. Stevenson
Date: 20/03/2015

1st supervisor: Prof. dr. Michel R.V. Chaudron
2nd supervisor: Dave Stikkelorum, MSc.

BACHELOR THESIS
Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This study examined the role of domain knowledge and abstract reasoning ability on undergraduate computer science students' software design learning and ability using the LIACS software design test. The study comprised a pretest-posttest design. Participants (N=151) were administered pretests that measured (1) software design skills on topics *separation of concerns, cohesion & coupling, maintenance, reuse* and *dependency*; (2) domain knowledge (UML); and (3) abstract reasoning ability respectively. During posttest the LIACS software design test was administered again; this was conducted after the students followed an undergraduate software engineering course. We found that initial software design ability was related to both abstract reasoning ability and domain knowledge; however, improvement in software design ability was somewhat related to domain knowledge, but not to abstract reasoning skills. More specifically with regard to the topics measured we found that *maintenance* and *reuse* were the most difficult pretest topics, but also showed the greatest improvement from pretest to posttest. In the future administering the LIACS software design test may be helpful for educators so that content can be tailored to the knowledge and skills the students already possess as well as evaluate how effective teaching of particular topics has been. Future research should investigate whether increasing students' knowledge of UML improves their chances of learning from a software design course.

1. Introduction

Software plays an essential role in our daily lives; from cell phones to kitchen appliances, and from furniture production to writing a paper. In order to create excellent software it must be well designed and good software designers are needed to avoid the high costs of software maintenance and error fixing. Software design "aims at the description of the basic features of the future computer system and prescribes the functions the system should perform" (p. 373)[1]. The question this paper explores is which factors influence ability and learning of software design in novices.

The two cognitive factors we investigated were fluid intelligence (i.e., abstract reasoning) and domain knowledge, both of which are related to problem solving skills and the development of expertise in a plethora of domains such as chess, computer programming or knowledge of baseball [2]–[5]. As Siau & Tan (2005) indicate in their review of human cognition factors that play a role in conceptual modeling for software design, domain knowledge and abstract reasoning ability are considered important aspects that play a role at level of the individual software designer[6]. However, this has yet to be investigated in the domain of software design ability. Evidence is provided by Leung & Bollejou (2006) based on their empirical study of design errors of novice analysts; they defined a set of frequent (syntactic and semantic) errors that were thought to be due deficits in one or more cognitive factors[7]. For example, syntactic errors could be caused by insufficient domain knowledge of UML notation whereas semantic errors may be due to inaccuracies in abstract reasoning and/or working memory constraints.

Kramer (2007) poses the question why some software engineers are able to design clear and eloquent systems while others, with similar education and conceptual modeling tools, are not; he argues that abstraction ability is key[8]. Abstraction is the ability to solve novel problems often measured by intelligence tests such as the Raven Progressive Matrices (RPM) that require one to complete visual patterns by inducing abstract rules. In research on the development of expertise abstract reasoning ability is said to play an important role with regard to a person's initial skill level and is also required for the first cognitive phase of skill acquisition[9].

Working memory, defined as the ability to process, store and retrieve information, also appears to be a factor predicting the acquisition of software development skills[6], [10]–[12]. However, working memory is strongly linked to abstract reasoning ability and some researchers argue that these two psychological constructs may be one and the same [13]–[15]. Given the similarities between working memory and abstract reasoning as well as the advice of Bergersen and colleagues (2011) to include a more pure measure of *g* (i.e., general intelligence often indicated a abstract reasoning ability) such as the Raven's Advanced Progressive Matrices (RAPM) to examine person factors influencing software development skills, we chose to use only RAPM in this study.

Domain knowledge is considered a main driving force behind skill acquisition - often referred to as the 'knowledge is power' hypothesis[2]. A few studies have found a relationship between software programming skills and prior experience[4], [16]. Bergersen et al. (2011) determined that domain knowledge was the underlying (i.e. mediating) source in this relationship - where people with more experience inherently have greater domain knowledge and thus perform better on software programming measures[12]. Indeed, domain knowledge showed a strong relationship with programming skills.

In the present study we examined how domain knowledge and abstract reasoning ability relate to students software design ability before and after a software engineering course. UML (Unified Modeling Language) is the current standard for visual abstract software design. For this reason we specifically focus on UML design ability and use the LIACS UML design skills test developed by Stikkelorum and colleagues (2013)[17]. By analyzing students performance and change on five item categories using statistical modeling techniques from item response theory (IRT)[18] we aimed to answer questions about the role of domain knowledge and abstract reasoning ability on: (1) initial performance on a software design test and (2) learning from a course on software design as measured by improvement on the LIACS software design test. In short, is abstract reasoning ability really the key predictor of which students can learn to design software or is "knowledge is power" and thus domain knowledge the driving force? Based on prior research in expertise development we expect

abstract reasoning ability to play the largest role in novice software designer's initial ability, whereas domain knowledge is expected to predict both initial ability and performance change.

2. Method

2.1. Participants

This study included 274 first year software engineering students (93% male) recruited from two university courses; one in Gothenburg, Sweden and the other in Utrecht, The Netherlands. Participants agreed to participation through an online informed consent form prior to completing the pretest. 248 students completed all pretest assessments and 151 students completed the posttest assessment.

2.2. Design & Procedure

The study comprised a pretest-posttest design. Each participant was administered three pretest assessments that measured (1) design skills, (2) domain knowledge and (3) abstract reasoning ability respectively. The posttest was conducted after the students followed a freshman undergraduate software engineering course at his/her university. The posttest consisted of the same software design skills task (assessment 1) administered at pretest.

2.3. Materials

2.3.1. LIACS Software Design Skills test

The LIACS Software Design test utilized in this study comprised 20 multiple-choice questions assessing the understanding of software design concepts[19]. The main concepts measured were design principles (separation of concerns, cohesion & coupling), quality (maintainability, reuse) and architecture (dependency); see Table 1 for an overview of which concepts were assessed per item. The items generally consisted of an example design and a question about consequences of changes in the design and/or a comparison of the quality two or more designs; see Figure 1 for an example. A complete list of the items including solutions and main measured concepts can be found in Appendix A. The designs

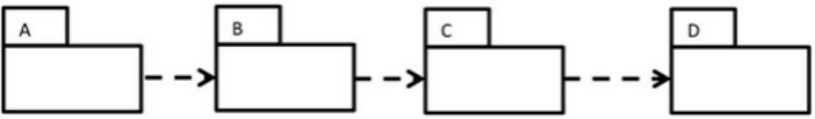
were presented in Unified Modeling Language (UML¹); this popular modeling language was chosen as it is used in the students' courses on software design.

Table 1.

Concepts measured per item from LIACS Design Skills test.

Item	Separation of concerns	Cohesion & coupling	Maintenance	Reuse	Dependency
1				X	X
2					X
3					X
4					X
5					X
6			X		
7					X
8	X				
9		X			
10	X				
11		X	X		
12		X			X
13			X		
14				X	
15	X	X			
16			X	X	
17	X		X		
18				X	
19			X		
20	X			X	

Consider the design below:



If C changes :

Choose one of the following answers

☐ Then A, B and D may have to change also.
☐ Only B may have to change.
☐ Both A and B may have to change.
☐ D may need to change.

Please enter your comment here:

Figure 1. Example item from LIACS Design Skills test.

The initial version of this test comprised 10 items that were evaluated by experts according to criteria of clarity, complexity and difficulty level and piloted in a

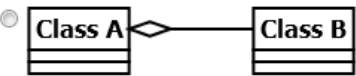
¹ <http://www.uml.org>

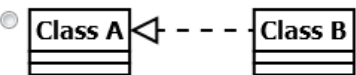
population of 60 first year university computer science students; a quantitative validation of the test items can be found in Stikkelorum et al. (2013)[17]. These items formed the basis for the items used in the present study. Additional items were created and rated by experts. A quantitative evaluation of the psychometric quality of the version of the items used in the present study can be found in section 3.1 of this paper.

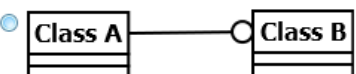
2.3.2. UML knowledge quiz

A UML knowledge quiz was created specifically for this experiment in order to assess domain knowledge that formed a prerequisite for understanding and correctly answering the diagrams used in the LIACS Software Design Skills test. The quiz consisted of 22 items focusing on the syntax and semantics of the Unified Modeling Language (UML). See Figure 2 for an example; the actual items can be found in Appendix B.

*12 Which of the following diagrams represents a relationship between Class A and Class B where A depends on B at a certain point during software execution?
Choose one of the following answers

☐ 

☐ 

☒ 

☐ other

Figure 2. Example item from the UML quiz used to measure domain knowledge.

2.3.3. Abstract Reasoning

A computerized short-form of the original Raven's Advanced Progressive Matrices[20] was administered to measure non-verbal abstract reasoning. This particular short form comprised the even items extracted from the original 36-item RAPM. The RAPM is a visuospatial reasoning task often used to measure fluid intelligence and requires participants to induce abstract rules and relations between geometric figures presented in a 3x3 matrix to complete a visual pattern by choosing among nine multiple-choice alternatives. Short forms of the

RAPM are used to reduce administration time without losing psychometric properties or predictive validity[21].

3. Results

3.1. Psychometric properties of software design and domain knowledge measures

Prior to examine the role of cognitive factors on the students' performance on the LIACS software design test and UML domain knowledge quiz the psychometric properties of these two new instruments were evaluated as measurement scales using both classical test theory (CTT) and item response (IRT) theory[22]. Both methods are used to evaluate test scores and item responses. CTT is more commonly utilized as the metrics are relatively easy to compute; however, IRT has distinct advantages such as being population independent so that item difficulty and discrimination can be evaluated without bias - important in the present situation as the population is rather specific with limited differences in software design ability.

According to CTT item difficulty is determined by the proportion correct responses per item. Item discrimination represents how sensitive an item is to actual differences between test-takers and is thus a means of examining an item's impact on the internal consistency of a test. In CTT item discrimination is represented by the point-biserial correlation between performance on a specific item and the test as a whole. In IRT item difficulty and discrimination can be computed using the two-parameter logistic (2PL) IRT model. In this IRT model the chance that an item is solved correctly (i.e., $y = 1$) depends on the difference between the latent ability (θ) of the test-taker p and the difficulty (β) and discrimination (α) of item i :

$$P(Y_{pi} = 1 | \theta_p, \beta_i, \alpha_i) = \frac{e^{(\alpha_i(\theta_p - \beta_i))}}{1 + e^{(\alpha_i(\theta_p - \beta_i))}}$$

$$\text{where } \theta_p \sim N(0, \sigma_\theta^2), \beta_i \sim N(0, \sigma_\beta^2), \text{ and } \alpha_i \sim N(0, \sigma_\alpha^2) \quad (1)$$

In this section we evaluate item difficulty and discrimination with both CTT and IRT; furthermore, reliability and construct validity are addressed from the CTT perspective.

3.2.1. LIACS Software Design Skills

Item difficulty (CTT, proportion correct) ranged from .19 to .80 (M=.58, SD=.17) for the pretest and from .25 to .87 (M=.67, SD=.16) for the posttest; this indicates that there was likely some improvement from pretest to posttest and that the items were generally solved correctly at above chance level (> .25 on items with 4 multiple-choice options). Ideally the 2PL IRT item difficulty parameters represent a range of difficulties so that the estimation of ability is accurate for persons with a range of latent aptitude (see Table 2).

Table 2.
Psychometric item properties of LIACS software design test.

item	Pretest				Posttest			
	CTT		IRT		CTT		IRT	
	p-value	Item-total correlation	β	α	p-value	Item-total correlation	β	α
1	.38	.21	0.81	0.73	.49	.26	0.08	0.63
2	.72	.27	-1.25	0.83	.74	.39	-1.02	1.29
3	.63	.23	-0.66	0.85	.70	.07	-4.39	0.19
4	.69	.42	-0.27	4.16	.57	.35	0.55	0.62
5	.71	.19	-0.56	10.75	.74	.31	-1.31	0.90
6	.57	.31	-0.67	0.41	.86	.36	-1.77	1.32
7	.63	.18	-2.81	0.28	.42	.25	-0.31	1.07
8	.80	.20	-6.48	0.18	.62	.16	-1.41	0.89
9	.61	.25	0.53	0.56	.85	.20	-3.85	0.51
10	.74	.32	3.00	0.15	.54	.16	-1.00	1.41
11	.68	.24	-2.61	0.29	.70	.25	5.30	0.21
12	.65	.23	1.18	0.30	.78	.37	-1.19	1.36
13	.44	.17	-6.23	-0.23	.87	.17	-1.17	1.39
14	.77	.15	-2.97	0.48	.75	.32	-1.40	0.37
15	.68	.07	-0.47	0.30	.25	.10	-0.72	1.61
16	.39	.18	-1.74	0.64	.74	.36	-0.33	0.46
17	.42	-.10	-1.03	0.43	.77	.37	-3.07	0.62
18	.19	.16	-1.29	0.62	.77	.34	-1.53	0.60
19	.54	.05	-1.18	0.52	.70	.41	-1.36	1.16
20	.30	.21	-8.60	-0.10	.50	.14	-0.03	0.37

* p<.05, ** p<.01

Item discrimination for CTT is represented by the item-total correlation. A rule-of-thumb in psychometrics is to discard items with an item-total correlation of less than .20. As can be seen in Table 2 nine items in the pretest and six items in the posttest meet this criterion. In the 2PL IRT model the item discrimination index is related to the range of test scores for which the item discriminates best between test-takers; higher values indicate greater discrimination and those

near zero ($\alpha < .20$) yield little to no information on the test-taker's ability and should be discarded (see Table 2). Based on these values it is advisable to revise items 8, 10 and 20 from the pretest and item 3 from the posttest. It is important to note for analyses presented in section 3.2.2 that items 8, 10 and 20 each measure understanding of the concept *separation of concerns*.

Reliability, i.e. "...the extent to which differences in respondents' observed scores are consistent with differences in their true scores" (Furr & Bacharach, 2014, p. 103), was determined using Cronbach's alpha coefficient of internal consistency; for the 20 item scale at pretest this was $\alpha = .60$ and at posttest $\alpha = .69$. These reliabilities are considered acceptable[23].

Construct validity, i.e. "...the degree to which test scores can be interpreted as reflecting a particular psychological construct" (Furr & Bacharach, 2014, p. 201), was determined by computing the correlation between posttest IRT scale scores and the students' scores on the software design course final exams. The resulting Pearson's ($r = .26, p = .06$) for the students from Utrecht who completed both the posttest and the final exam ($N = 55$) is in the expected positive direction and considered weak; this provides some evidence that similar constructs were measured by the LIACS software design test and the respective course exams.

3.2.2. UML Domain Knowledge Quiz

Item difficulty ranged from .00 to .77 ($M = .37, SD = .21$). Reliability based on Cronbach's alpha coefficient of internal consistency was $\alpha = .69$ and is considered good[23]. Table 3 provides an overview of the CTT and IRT item properties. Based on these results it is advisable to remove or revise item 13 as no one solved this correctly and also to revise items 1, 5-7, 11 and 20 given the low item-total correlations.

Table 3.
Psychometric item properties of UML domain knowledge quiz.

item	CTT	IRT		
	<i>p</i> -value	Item-total correlation	β	α^*
1	.08	.02	2.34	1.25
2	.64	.56	-0.54	1.25
3	.68	.58	-0.73	1.25
4	.47	.40	0.18	1.25
5	.18	.15	1.52	1.25
6	.18	.08	1.54	1.25
7	.23	.15	1.23	1.25
8	.17	.22	1.60	1.25
9	.38	.40	0.58	1.25
10	.29	.30	0.93	1.25
11	.15	.19	1.76	1.25
12	.17	.20	1.62	1.25
13	.00	.00	10.07	1.25
14	.77	.57	-1.19	1.25
15	.40	.41	0.48	1.25
16	.52	.46	-0.03	1.25
17	.39	.40	0.51	1.25
18	.59	.59	-0.32	1.25
19	.50	.49	0.07	1.25
20	.13	.05	1.88	1.25
21	.60	.58	-0.38	1.25
22	.29	.22	0.95	1.25

* The best IRT model for the UML quiz is a special case of the 2PL model, i.e. the Rasch (1PL) model with the same α value for all items.

3.2. Performance on the LIACS software design test

This was investigated using explanatory IRT[18] analyses where item responses (Y_{pi}) for person p on item i on the LIACS Software design ability test was the dependent variable and j variables were evaluated as predictors (X) using the following regression model:

$$P(Y_{pi} = 1 | X_{pij}, \beta_i) = \frac{e^{(\sum_{j=1}^J B_j X_{pij} + \theta_p + \beta_i)}}{1 + e^{(\sum_{j=1}^J B_j X_{pij} + \theta_p + \beta_i)}}$$

$$\text{where } \theta_p \sim N(0, \sigma_\theta^2) \text{ and } \beta_i \sim N(0, \sigma_\beta^2) \quad (2)$$

The resulting regression weights (B) for the person predictors (abstract reasoning, domain knowledge) and item predictors (measured concepts: separation of concerns, cohesion & coupling, maintenance, reuse, dependency)

are shown in Table 4. The results of this model are discussed per predictor type (persons, items) in the following two sections.

Table 4.

Results of the explanatory item response model evaluating item predictors (measured concepts: separation of concerns, cohesion & coupling, maintenance, reuse, dependency) and person predictors (abstract reasoning, domain knowledge, pretest-to-posttest change). Main item effects were evaluated with the pretest as reference time point.

Predictor	B	SE	p	lower bound	upper bound
Intercept (overall mean)	-0.02	0.41	.97	-0.42	0.39
<i>Main effects item predictors</i>					
separation of concerns	-0.23	0.30	.44	-0.53	0.07
cohesion & coupling	-0.04	0.28	.88	-0.32	0.23
maintenance	-0.71	0.30	.02	-1.02	-0.41
reuse	-1.23	0.28	<.001	-1.50	-0.95
dependency	-0.17	0.33	.60	-0.50	0.16
<i>Main effects person predictors</i>					
post-pre	0.23	0.36	.53	-0.13	0.59
abstract reasoning	0.08	0.02	<.001	0.06	0.10
domain knowledge	0.23	0.06	<.001	0.17	0.29
<i>Interaction effects item predictors with pretest-to-posttest change</i>					
separation of concerns X post-pre	-1.02	0.19	<.001	-1.20	-0.83
cohesion & coupling X post-pre	-0.29	0.18	.11	-0.46	-0.11
maintenance X post-pre	0.95	0.19	<.001	0.76	1.15
reuse X post-pre	0.88	0.17	<.001	0.71	1.06
dependency X post-pre	-0.54	0.21	.01	-0.75	-0.33
<i>Interaction effects person predictors with pretest-to-posttest change</i>					
abstract reasoning X post-pre	0.02	0.02	.29	0.00	0.05
domain knowledge X post-pre	-0.15	0.08	.06	-0.23	-0.07

3.2.1. Role of abstract reasoning and domain knowledge in learning software design

The main research question was whether abstract reasoning and domain knowledge were able to predict initial ability and learning of software design. The results of the statistical model (see Table 4) showed that the chance of solving an item correctly generally improved from pretest to posttest for a student with average abstract reasoning skills and domain knowledge. Domain knowledge was a significant predictor of pretest performance ($B=0.23$, $SE=0.06$, $p<.001$) and a marginal predictor of improvement from pretest to posttest ($B=-0.15$, $SE=0.08$, $p=.06$); see Figure 3. Abstract reasoning was a significant predictor of pretest performance ($B=0.08$, $SE=0.02$, $p<.001$), but did not predict performance change over time ($B=0.02$, $SE=0.02$, $p=.06$); see Figure 4.

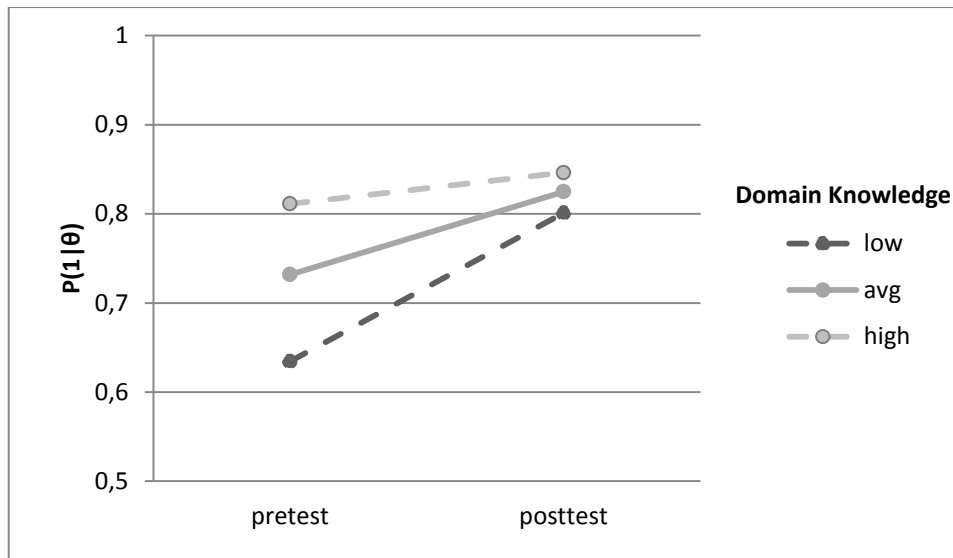


Figure 3. Domain knowledge (low = -2 SD, avg = Mean and high = +2 SD) significantly influenced the chance of solving LIACS software design test items correctly (y-axis) on the pretest and showed a marginal effect in predicting improvement from pretest to posttest (x-axis). Students with low domain knowledge generally performed less well on the software design test, whereas those with higher domain knowledge performed better. The gap between students with lower and higher domain knowledge decreased from pretest to posttest after following a software engineering course.

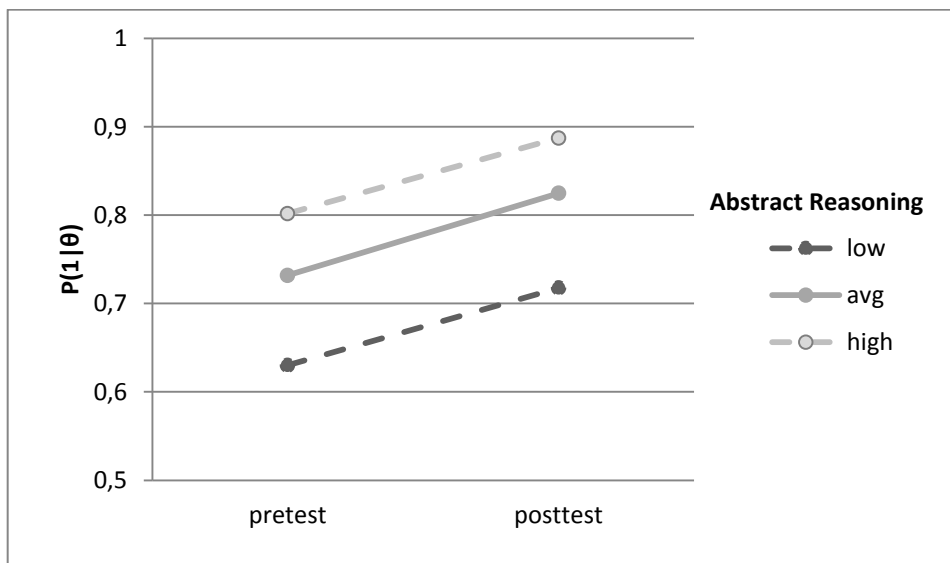


Figure 4. Abstract reasoning ability (low = -2 SD, avg = Mean and high = +2 SD) significantly influenced the chance of solving LIACS software design test items correctly (y-axis) on the pretest, but did not affect improvement from pretest to posttest (x-axis). Students with low abstract reasoning ability generally performed less well on the software design test, whereas those with higher abstract reasoning ability performed better. The differences between students of differing abstract reasoning ability did not change after following a software engineering course, i.e. abstract reasoning ability did not predict learning from before and after the course.

3.2.2. Performance per measured concept

The LIACS software design test contained items measuring five concepts: separation of concerns, cohesion & coupling, maintainability, reuse and dependency. The results are depicted in Figure 5. Items measuring understanding of the concepts *maintainability* and *reuse* were significantly more difficult than the average item (maintenance: $B=-0.71$, $SE=0.30$, $p=.02$; reuse: $B=-1.23$, $SE=0.28$, $p<.001$), whereas *separation of concerns*, *cohesion & coupling* and *dependency* were of average difficulty. Performance improvement from pretest to posttest was significant ($p<.05$) for items measuring understanding of the concepts *maintainability* and *reuse* (maintenance: $B=0.95$, $SE=0.19$, $p<.001$; reuse: $B=0.88$, $SE=0.17$, $p<.001$); however, for *separation of concerns* and *dependency* there was a significant decline in performance from pretest to posttest (separation of concerns: $B=-1.02$, $SE=0.19$, $p<.001$; dependency: $B=-0.54$, $SE=0.21$, $p<.01$). There were no significant changes in performance on the items measuring understanding of *cohesion & coupling* ($B=-0.29$, $SE=0.18$, $p=.11$).

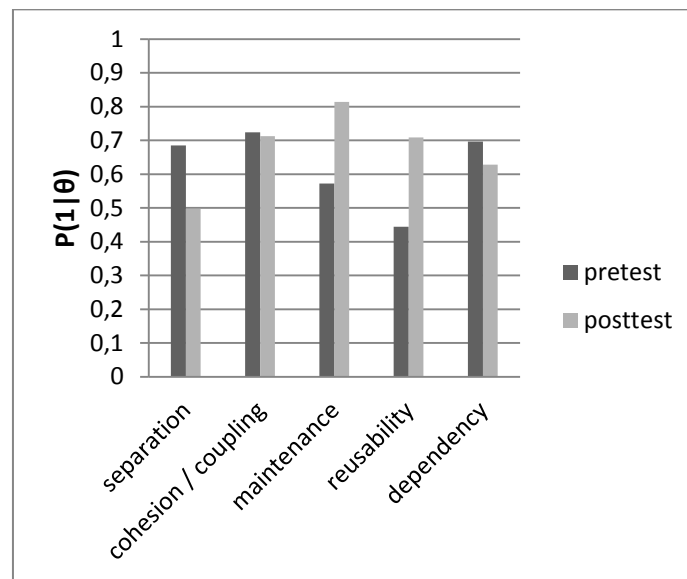


Figure 5. The chance of solving an item correctly (y-axis) that measured a particular concept (x-axis) on the LIACS software design test administered at pretest (light gray columns) and posttest (dark gray columns). Results show that items concerning the topics *maintenance* and *reuse* were more difficult on the pretest than items measuring understanding of the concepts *separation of concerns*, *cohesion & coupling* and *dependency*. Performance on items measuring understanding of *maintainability* and *reuse* improved, but performance on items measuring understanding of *separation of concerns* and *dependency* declined. There was no change in performance on items measuring *cohesion & coupling*. The presented model was computed with domain knowledge and abstract reasoning fixed at average skill levels.

4. Discussion

The aim of the current study was to examine the role of domain knowledge and abstract reasoning ability on undergraduate computer science students' software design learning and ability. To this end, participants completed assessments of (1) software design ability in which comparisons between and/or judgements of changes to designs were assessed five concepts: separation of concerns, cohesion & coupling, maintainability, reuse and dependency; (2) domain knowledge of the syntax and semantics of the Unified Modeling Language (UML) commonly utilized in software design; and (3) abstract reasoning ability. The software design task was administered to first year computer science students before and after a software engineering course. There were four main findings: (1) initial software design ability was related to both abstract reasoning ability and domain knowledge; (2) improvement in software design ability was somewhat related to domain knowledge, but not to abstract reasoning skills; (3) the topics *maintenance* and *reuse* were more difficult on the pretest than *separation of concerns*, *cohesion & coupling* and *dependency*; and (4) understanding of the concepts *maintainability* and *reuse* improved from pretest to posttest, but performance on items measuring understanding of *separation of concerns* and *dependency* declined. The discussion is organized along these findings.

4.1. Abstract reasoning and domain knowledge as predictors software design ability and learning

The students' knowledge of the correct syntax and semantics of the Unified Modeling Language (UML) was used as a proxy to measure their domain knowledge in the field of software design. Based upon prior research on the development of expertise[24] and the "knowledge is power" hypothesis[2] we expected domain knowledge to play a role in both initial software design ability as well as how much students learned about software design after following a course in software engineering. Indeed UML knowledge was a predictor of both initial software design skills as well as improvement from pretest to posttest. This finding is in accordance with the findings of Bergensen et al. (2011) in the domain of software programming where greater knowledge indicated better programming skills in expert software developers.

Abstract reasoning ability, a main component of fluid intelligence, has been postulated to be involved in all learning, especially novel tasks[25], [26]. Therefore, we expected the initial ability and learning of software design of the undergraduate computer science students in our study also to be related to their performance on the Raven's Advanced Progressive matrices - a measure of fluid intelligence. Indeed our results showed that initial understanding of software design principles were significantly predicted by the students' abstract reasoning ability. However, the students' learning from a course in software design as measured by their change in performance on the LIACS software design test administered before and after the course, was not related to their abstract reasoning ability. Perhaps the effect of abstract reasoning, which is a construct similar to working memory, is also mediated by domain knowledge as was the case in Bergensen et al.'s (2011) found that the predictive effect of working memory on software programming skills in experts was mediated by domain knowledge. Abstract reasoning is very similar if not the same construct as working memory[13]–[15]; therefore perhaps domain knowledge also functions as a mediator in the prediction of software design skills learning - i.e., those with better abstract reasoning likely have greater domain knowledge and show the most growth in software design performance. In our statistical model we included abstract reasoning and domain knowledge as predictors at the same level, if domain knowledge is a mediator this could remove evidence the influence of abstract reasoning; future investigations should focus on whether domain knowledge mediates the relationship between abstract reasoning and the learning of software design principles.

4.2. Understanding of measured software design concepts

On the whole the students performed quite well on the LIACS software design test, with approximately 60% chance of solving pretest items correctly and a 70% chance on the posttest - thus on the whole showing significant improvement from pretest to posttest. Students appeared to know more about the design concepts *separation of concerns*, *cohesion & coupling* and *dependency* during the pretest than the topics *maintenance* and *reuse*. However, the most improvement from pretest to posttest was on these two more difficult topics;

this is most likely because of the relatively high chance of solving items on the other three topics correctly - there may have been less room to improve on these particular topics. Strangely, performance on items measuring understanding of *separation of concerns* and *dependency* declined somewhat; this may be due to problems with the items measuring these topics; the psychometric analyses of the test indicate that nearly all of the items measuring separation of concerns should be revised. On the whole our results provide insight into which topics students mastered both before and after the course. In the future administering the LIACS software design test may be helpful for educators so that content can be tailored to the knowledge and skills the students already possess as well as evaluate how effective teaching of particular topics has been.

4.3. Limitations

Some limitations of this study deserve mention and can be informative for future research. First, domain knowledge in the form of a UML quiz was only assessed at pretest. In future research this task should also be administered at posttest in order to (1) assess gains in domain knowledge as a consequence of the course and (2) examine whether the mediating role of domain knowledge on the students' design ability changes during the learning process - a specific prediction of Ackerman's (1998) theory of how skills develop[9].

Second, the instruments we used to assess software design ability and domain knowledge (UML) were created specifically for this study and require further development and evaluation before assuming their reliability and validity. The reliability of these measures were addressed using psychometric classical test theory and item response theory; however, certain revisions to the items are warranted given their low correlation with other related items and/or lack of fit in the item response measurement model. The construct validity was examined by computing correlations with course grades; the moderate correlation indicated that the software design skills test indeed measures a similar construct to the software engineering course exams at both universities.

A third limitation was that we did not account for the role of experience in the students' ability to learn software design. Experience is a good predictor of expertise in novices[1] and likely strongly related to both domain knowledge and

initial software design ability as appears to be the case with software programming [12]. A fourth and final limitation to overcome in future research is the inclusion of a control group. By including a control group we can control for the effect of retesting when measuring learning in a pretest-intervention-posttest design.

4.4. Conclusions & Future Directions

Based on this study we can conclude that both domain knowledge and abstract reasoning are key predictors of students' software design ability. Domain knowledge in particular predicts how much students learn from a software design course. Perhaps increasing students' knowledge of UML may improve their chances of learning from a software design course. In the future the effectiveness of improving domain knowledge on course or training outcomes should be investigated.

The present study focused on undergraduate students performance before and after a course on software design; it would be of particular interest to use the LIACS software design test to examine the performance of complete novices (e.g., psychology students) and undergraduate computer science students before and after following training in software design[27].

5. Acknowledgements

This project was conducted under the supervision of prof. dr. Michel Chaudron and PhD candidate Dave Stikkelorum. Dave conducted nearly all of the data collection for this project and has published a paper on this topic[19]. I am very grateful for the opportunity to collaborate with these researchers on such an interesting project that in addition to completing my computer science study also spans my interests and expertise developed as a researcher in developmental and educational psychology.


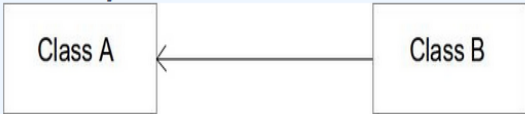
6. References

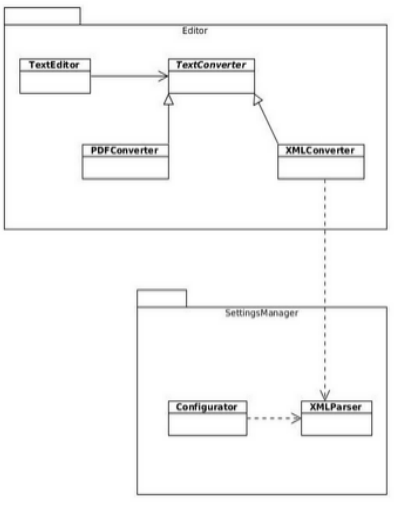
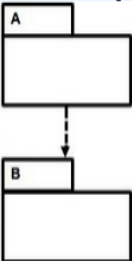
- [1] S. Sonnentag, C. Niessen, and J. Volmer, "Expertise in software design," in *The Cambridge handbook of expertise and expert performance*, no. 1981, 2006, pp. 373–387.

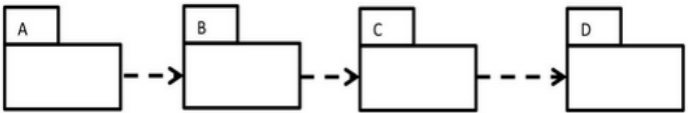
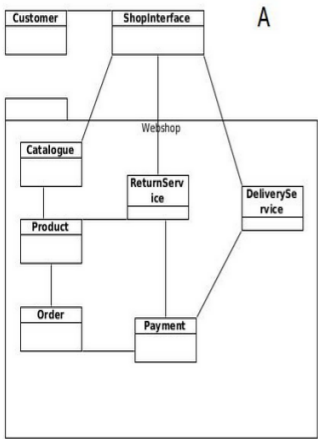
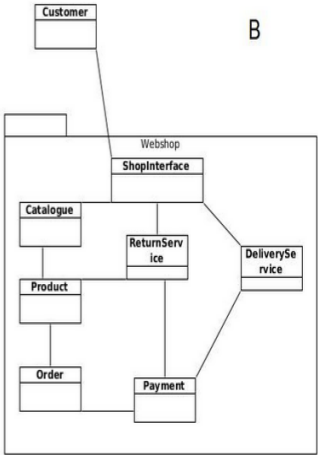
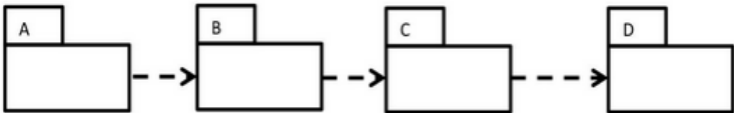
- [2] D. Z. Hambrick and R. W. Engle, "Effects of domain knowledge, working memory capacity, and age on cognitive performance: an investigation of the knowledge-is-power hypothesis.," *Cogn. Psychol.*, vol. 44, no. 4, pp. 339–87, Jun. 2002.
- [3] R. B. Cattell, *Intelligence: its Structure, Growth and Action: its Structure, Growth and Action*. New York, New York, USA: Elsevier, 1987.
- [4] J. R. Anderson, "Skill acquisition: Compilation of weak-method problem situations.," *Psychol. Rev.*, vol. 94, no. 2, pp. 192–210, 1987.
- [5] P. L. Ackerman, "New Developments in Understanding Skilled Performance," *Curr. Dir. Psychol. Sci.*, vol. 16, no. 5, pp. 235–239, Oct. 2007.
- [6] K. Siau and X. Tan, "Improving the quality of conceptual modeling using cognitive mapping techniques," *Data Knowl. Eng.*, vol. 55, no. 3, pp. 343–365, Dec. 2005.
- [7] N. Bolloju and F. S. K. Leung, "Assisting novice analysts in developing quality conceptual models with UML," *Commun. ACM*, vol. 49, no. 7, pp. 108–112, 2006.
- [8] J. Kramer, "Is abstraction the key to computing?," *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007.
- [9] P. L. Ackerman, "Determinants of individual differences during skill acquisition: Cognitive abilities and information processing.," *J. Exp. Psychol. Gen.*, vol. 117, no. 3, pp. 288–318, 1988.
- [10] V. J. Shute, "Who is Likely to Acquire Programming Skills?," *Journal of Educational Computing Research*, vol. 7, pp. 1–24, 1995.
- [11] W. W. Wittmann and H.-M. Süß, "Investigating the paths between working memory, intelligence, knowledge, and complex problem-solving performances via Brunswik symmetry.," in *Learning and individual differences: Process, trait, and content determinants.*, P. L. Ackerman, P. C. Kyllonen, and R. D. Roberts, Eds. Washington, D.C.: APA, 1999, pp. 77–108.
- [12] G. R. Bergersen and J.-E. Gustafsson, "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective," *J. Individ. Differ.*, vol. 32, no. 1987, pp. 201–209, 2011.
- [13] P. L. Ackerman, M. E. Beier, and M. O. Boyle, "Working Memory and Intelligence : The Same or Different Constructs ?," vol. 131, no. 1, pp. 30–60, 2005.
- [14] M. Buehner, S. Krumm, and M. Pick, "Reasoning = working memory + attention," vol. 33, pp. 251–272, 2005.

- [15] R. Colom, I. Rebollo, A. Palacios, M. Juan-espinoza, and P. C. Kyllonen, "Working memory is (almost) perfectly predicted by g," vol. 32, pp. 277–296, 2004.
- [16] E. Arisholm and D. I. K. Sjöberg, "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 30, no. 8, pp. 521–534, Aug. 2004.
- [17] D. R. Stikkolorum, C. E. Stevenson, and M. R. V Chaudron, "Technical report 2013-02," Leiden, The Netherlands, 2013.
- [18] P. A. L. De Boeck and M. Wilson, *Explanatory item response models: A generalized linear and nonlinear approach*. New York, New York, USA: Springer, 2004.
- [19] D. R. Stikkolorum, C. E. Stevenson, and M. R. V Chaudron, "Assessing Software Design Skills and Their Relation With Reasoning Skills," in *EduSymp@ MoDELS*, 2013, pp. 1–8.
- [20] J. Raven, J. C. Raven, and J. H. Court, *Manual for Raven's Progressive Matrices and Vocabulary Scales. Section 4: The Advanced Progressive Matrices*. San Antonion, TX: Harcourt Assessment, 1998.
- [21] D. A. Bors and T. L. Stokes, "Raven's Advanced Progressive Matrices: Norms for First-Year University Students and the Development of a Short Form," *Educ. Psychol. Meas.*, vol. 58, no. 3, pp. 382–398, Jun. 1998.
- [22] S. E. Embretson and S. Reise, *Item response theory for psychologists*. Mahwah, NJ: Erlbaum Publishers., 2000.
- [23] R. M. Furr and V. R. Bacharach, *Psychometrics An Introduction*, 2nd ed. London: SAGE, 2014, p. 442.
- [24] K. A. Ericsson, N. Charness, P. J. Feltovich, and R. R. Hoffman, *Handbook of Expertise*. London: Cambridge University Press, 2006.
- [25] M. C. Voelkle, W. W. Wittmann, and P. L. Ackerman, "Abilities and skill acquisition: A latent growth curve approach," *Learn. Individ. Differ.*, vol. 16, no. 4, pp. 303–319, Jan. 2006.
- [26] R. Primi, M. E. Ferrão, and L. S. Almeida, "Fluid intelligence as a predictor of learning: A longitudinal multilevel approach applied to math," *Learn. Individ. Differ.*, vol. 20, no. 5, pp. 446–451, Oct. 2010.
- [27] D. R. Stikkolorum, M. R. V. Chaudron, and O. de Bruin, "The Art of Software Design, a Video Game for Learning Software Design Principles," in *Models 2012*, 2012.

Appendix A: Software Design Skills Questions

Item: solution	Content	Concepts
1: B	<p>*15 Consider the design below:</p>  <pre> classDiagram ClassB --> ClassA </pre> <p>If we want to reuse Class B we:</p> <p>Choose one of the following answers</p> <p> <input type="radio"/> have to change Class A <input type="radio"/> also need to use Class A <input type="radio"/> do not need to use Class A <input type="radio"/> need to inherit from A </p> <p>Please enter your comment here:</p> <input type="text"/>	Reuse, Dependency
2: C	<p>*16 Consider the design below:</p>  <pre> classDiagram ClassB --> ClassA </pre> <p>If we change Class B, we:</p> <p>Choose one of the following answers</p> <p> <input type="radio"/> need to change Class A <input type="radio"/> need to inherit from A <input type="radio"/> do not need to change Class A <input type="radio"/> can not do that, because B uses A </p> <p>Please enter your comment here:</p> <input type="text"/>	Dependency

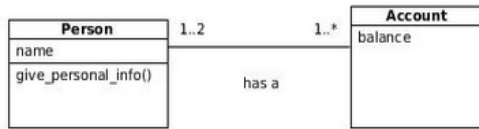
<p>3:</p> <p>C</p>	<p>*17 Consider the design below:</p>  <p>This question concerns the impact of changes in Package 'Settingsmanager' on the package 'Editor'.</p> <p>Suppose the developer wants to add an attribute to the class 'Configurator'.</p> <p>Then ...</p> <p>Choose one of the following answers</p> <ul style="list-style-type: none"> <input type="radio"/> it requires a change in 'Editor', because it uses a class from 'Settingsmanager'. The use of this class propagates changes in 'Settingsmanager' to all classes in 'Editor'. <input type="radio"/> it requires a change in Editor because it uses XMLParser. And XMLParser is used by 'Configurator' <input checked="" type="radio"/> it does not require a change, because 'Configurator' has no impact on 'XMLParser'. <input type="radio"/> it requires a change in 'XMLConverter', because XMLParser would be affected. 	<p>Dependency</p>
<p>4:</p> <p>D</p>	<p>*18 Consider the design below:</p>  <p>If package A depends on package B, and a change is performed in a class of package B, then:</p> <p>Choose one of the following answers</p> <ul style="list-style-type: none"> <input type="radio"/> changes in one package never affect other packages (separation of concerns) <input type="radio"/> it is always necessary to check all classes in package A whether they need to be changed accordingly to the change in B. <input type="radio"/> it is always necessary to change at least one class in package A <input type="radio"/> only if the change in B affects the interface it offers to package A, do we need to check whether A also needs to be changed. 	<p>Dependency</p>

<p>5: B</p>	<p>•19 Consider the design below:</p>  <p>If B changes :</p> <p>Choose one of the following answers</p> <p> <input type="radio"/> Then A, C and D must also change. <input type="radio"/> Only A may change. <input type="radio"/> Both A and C must change. <input type="radio"/> C and D must change. </p> <p>Please enter your comment here:</p>	<p>Dependency</p>
<p>6: C</p>	<p>•20 Consider the 2 designs of the same system below:</p> <div style="display: flex; justify-content: space-around;"> <div data-bbox="408 712 727 1149"> <p>A</p>  </div> <div data-bbox="823 696 1142 1149"> <p>B</p>  </div> </div> <p>If we change Class B, we:</p> <p>Choose one of the following answers</p> <p> <input type="radio"/> A is a better design, because interfacing classes should only exists outside packages <input type="radio"/> Both Design A and B are equally good considering maintainece <input type="radio"/> B is a better design, because the number of points of entries to the package is lowered by making the interfacing classes part of the package. <input type="radio"/> Both are bad designs because they introduce an extra indirection . </p>	<p>Maintenance</p>
<p>7: C</p>	<p>•21 Consider the design below:</p>  <p>If C changes :</p> <p>Choose one of the following answers</p> <p> <input type="radio"/> Then A, B and D may have to change also. <input type="radio"/> Only B may have to change. <input type="radio"/> Both A and B may have to change. <input type="radio"/> D may need to change. </p> <p>Please enter your comment here:</p>	<p>Dependency</p>

8:
D

*22

Consider the design below:



In a banking system a person's account balance is stored. The operation 'procesPayment()', that lowers the balance has to be added. Which class does this operation belong to?

Choose one of the following answers

- ☐ In class Person
- ☐ In both classes
- ☐ Neither, Person can approach balance
- ☐ In class Account

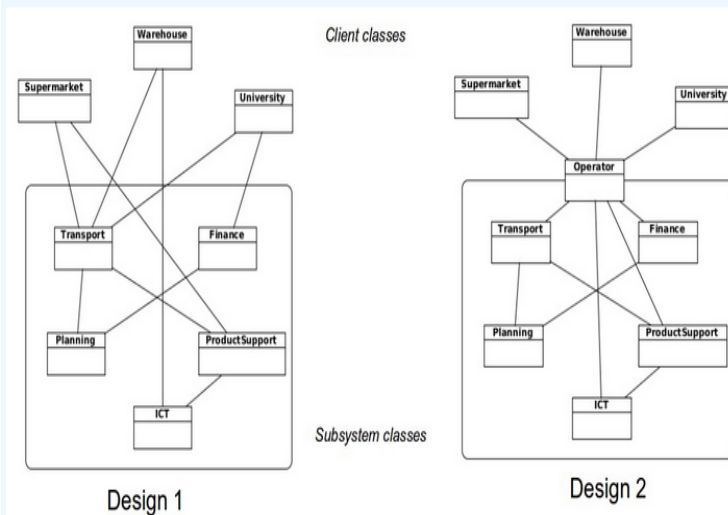
Please enter your comment here:

Separation of
concerns

9:
C

*23

Below are two designs for the same system. The Operator handles method-requests for the clients classes instead of directly (as in Design 1).



What is true about using this 'Operator' in Design 2?

Choose one of the following answers

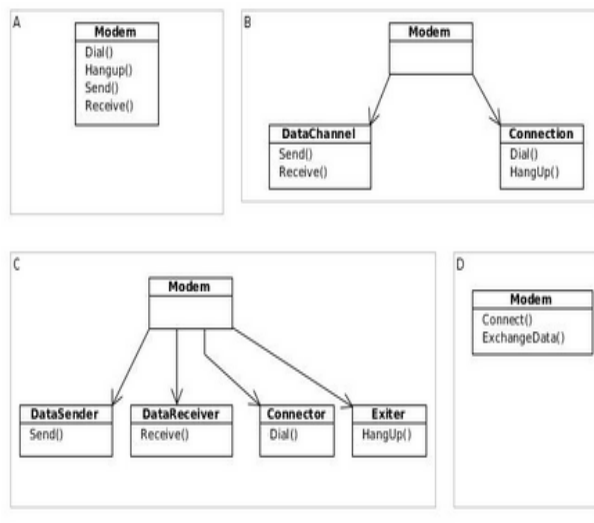
- ☐ It is a bad idea, considering separation of concerns.
- ☐ It is a good idea, but it increases coupling.
- ☒ It is a good idea, it hides information from the clients.
- ☐ It is a bad idea, because it costs an extra class.

Coupling /
Cohesion,
Separation of
concerns

10:
B

*24

Consider the designs of the same system below:



Which one is a better design, considering assignment of responsibility?

Choose one of the following answers

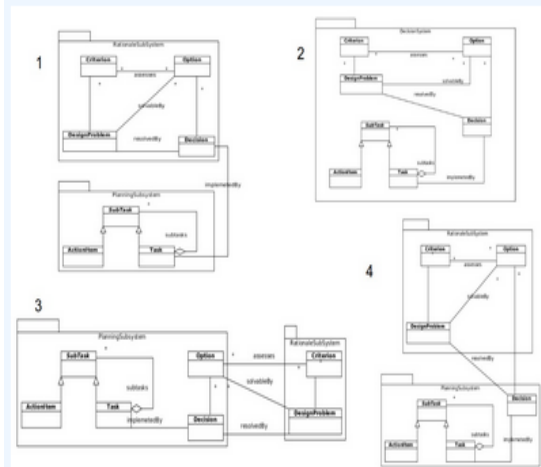
- ☐ Design A, because the system is too small to split up in different classes with different responsibilities.
- ☒ Design B, because operations that are part of the same task are combined to a responsibility.
- ☐ Design C, because every operation is a responsibility.
- ☐ Design D, because it is necessary to reduce the amount of operations in a class, not the responsibility.

Separation of
concerns

11:
A

*25

Consider the designs for the same system below:



Which design is the easiest to maintain?

Choose one of the following answers

- ☒ Design 1, because there is low coupling between the packages and high cohesion within the packages
- ☐ Design 2, because there is no coupling of packages and average cohesion.
- ☐ Design 3, because there is high coupling between the packages and average cohesion within the packages.
- ☐ Design 4, because there is high cohesion in the packages and average coupling between the packages.

Coupling /
Cohesion,
Maintenance

<p>12:</p> <p>C</p>	<p><small>*26</small></p> <p>Design 1 and 2 represent possible designs for the same system. In design 2 Class A1 en A2 together have the same function as Class A in design 1. Which one is the preferable design?</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Design 1</p> </div> <div style="text-align: center;"> <p>Design 2</p> </div> </div> <p>Choose one of the following answers</p> <ul style="list-style-type: none"> <input type="radio"/> Design 1, considering loose coupling. <input type="radio"/> Design 1, because it has less associations. <input type="radio"/> Design 2, because it brakes circular dependency <input type="radio"/> Design 2, because it is more tight coupled. <input checked="" type="radio"/> Both bad design , because of to many dependencies 	<p>Coupling / cohesion, Dependency</p>
<p>13:</p> <p>C</p>	<p><small>*27</small></p> <p>Design 1 and 2 represent possible designs for the same system. Suppose the price for a movie is determined by its length. Which one is the preferable design?</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Design 1</p> </div> <div style="text-align: center;"> <p>Design 2</p> </div> </div> <p>Choose one of the following answers</p> <ul style="list-style-type: none"> <input type="radio"/> Design 1 <input type="radio"/> Design 2 <input type="radio"/> One is not better than the other. 	<p>Maintenance</p>
<p>14:</p> <p>A</p>	<p><small>*28</small></p> <p>Consider the design below:</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>A</p> </div> <div style="text-align: center;"> <p>B</p> </div> </div> <p>Design B has interfaces, that design A does not have. This ...</p> <p>Choose one of the following answers</p> <ul style="list-style-type: none"> <input checked="" type="radio"/> makes translator easier to reuse <input type="radio"/> makes translator more difficult to reuse <input type="radio"/> makes no difference with a concerning reuse <input type="radio"/> makes translator more dependent of Dutch and English 	<p>Reuse</p>

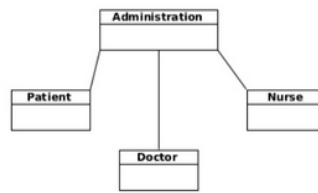
15:
C

*29
A patient comes to the administration of a hospital. The administration calls the doctor for an examination. The doctor diagnoses and calls the nurse to make because he has to stay over.

Design 1



Design 2



Which diagram is a better design?

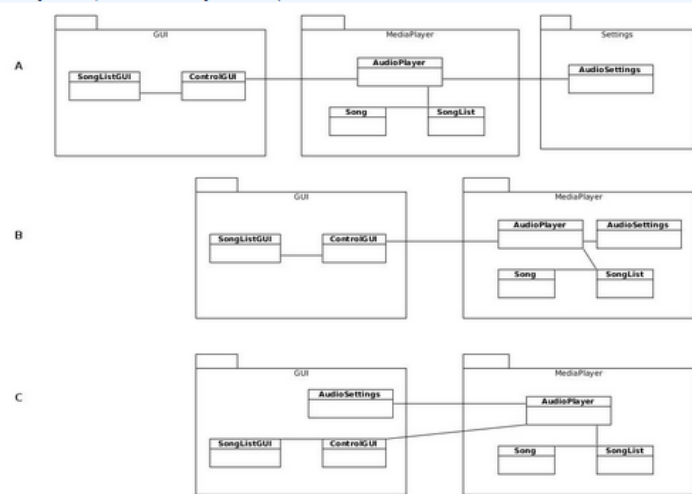
Choose one of the following answers

- ☐ Design 1 is a better design, because it has low coupling. In design 2 the coupling is high.
- ☐ Design 1 is a better design, because the classes correspond to a process order, thus easier to understand.
- ☒ Design 2 is a better design, because the responsibility of controlling the process can be implemented in a single place: Administration.
- ☐ Design 2 is a better design, because of low coupling.

Separation of
concerns,
coupling /
cohesion

16:
A

*30
The designs below represent alternative design of the same system.



In this question we use coupling and cohesion as criteria for quality of design.

Which statement is true?

Choose one of the following answers

- ☐ A is the best design.
- ☐ B is the best design.
- ☐ C is the best design.
- ☒ A and B are equally good

Please enter your comment here:

Reuse,
maintenance

17:
C

22 Consider the design below:

A

```

classDiagram
    class PushButton {
        push()
    }
    class Lamp {
        turnOn()
        turnOff()
    }
    PushButton --> Lamp
  
```

B

```

classDiagram
    class PushButton {
        push()
    }
    class SwitchableDevice {
        <<interface>>
        turnOn()
        turnOff()
    }
    class Lamp {
    }
    PushButton --> SwitchableDevice
    Lamp --|> SwitchableDevice
  
```

C

```

classDiagram
    class AbstractSwitch {
        <<interface>>
        activate()
    }
    class SwitchableDevice {
        <<interface>>
        turnOn()
        turnOff()
    }
    class PushButton {
        push()
    }
    class Lamp {
    }
    AbstractSwitch --|> PushButton
    SwitchableDevice --|> Lamp
    AbstractSwitch --> SwitchableDevice
  
```

Suppose one wants to add the fragment below.

```

classDiagram
    class SwitchButton {
        switch()
    }
  
```

In which of the designs can this fragment be added without changing the classes of the original design and preserve maintainability of the design?

Choose one of the following answers

☐ Only A

☐ Only B

☐ Only C

☒ B and C

Please enter your comment here:

Maintenance,
separation of
concerns

18:
A

22 Consider the design below:

A

```

classDiagram
    class Display {
        showTime()
        showSong()
    }
    class Radio {
        chooseFrequency()
    }
    class USBDevice {
        readFile()
    }
    class Controller {
        playRadio()
        playUSB()
        enableAlarm()
        stopAlarm()
    }
    class Alarm {
        setAlarm()
    }
    class Timer {
        tick()
    }
    class Speaker {
        produceSound()
    }
    Display --> Controller
    Radio --> Controller
    USBDevice --> Controller
    Controller --> Alarm
    Controller --> Timer
    Controller --> Speaker
  
```

B

```

classDiagram
    class Display {
        showTime()
        showSong()
    }
    class MusicPlayer {
        playRadio()
        readFile()
        chooseFrequency()
        playUSB()
        operation()
    }
    class AlarmControl {
        enableAlarm()
        stopAlarm()
    }
    class Alarm {
        setAlarm()
    }
    class Timer {
        tick()
    }
    class Speaker {
        produceSound()
    }
    Display --> MusicPlayer
    MusicPlayer --> AlarmControl
    AlarmControl --> Alarm
    AlarmControl --> Timer
    AlarmControl --> Speaker
  
```

C

```

classDiagram
    class Display {
        showTime()
        showSong()
    }
    class Radio {
        chooseFrequency()
    }
    class USBDevice {
        readFile()
    }
    class AlarmClock {
        playRadio()
        playUSB()
        enableAlarm()
        setAlarm()
        stopAlarm()
        tick()
    }
    class Speaker {
        produceSound()
    }
    Display --> AlarmClock
    Radio --> AlarmClock
    USBDevice --> AlarmClock
    AlarmClock --> Speaker
  
```

D

```

classDiagram
    class Display {
        showTime()
        showSong()
    }
    class RadioAlarmClock {
        playRadio()
        chooseFrequency()
        playUSB()
        readFile()
        enableAlarm()
        setAlarm()
        stopAlarm()
        tick()
    }
    class Speaker {
        produceSound()
    }
    Display --> RadioAlarmClock
    RadioAlarmClock --> Speaker
  
```

Suppose one wants to reuse classes of one of these designs that provide functions like time and timing (alarm) for a microwave oven. Which one would suit the best?

Choose one of the following answers

☒ Design A

☐ Design B

☐ Design C

☐ Design D

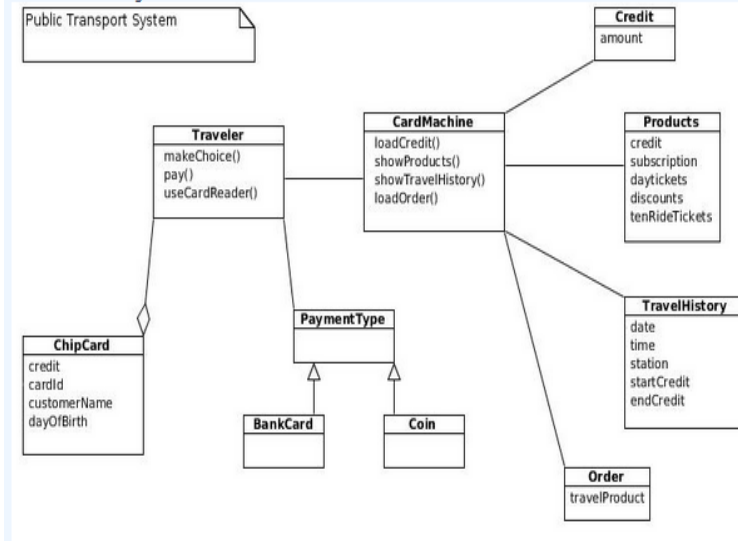
Please enter your comment here:

Reuse

19:
A

*33

Consider the design below:



Suppose this is a final design:

Errors that have been made are:

Choose one of the following answers

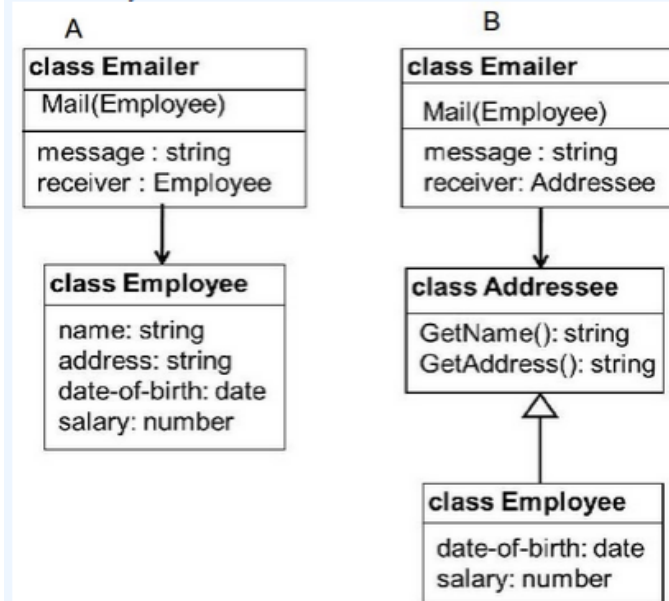
- ☒ wrong classes are associated
- ☐ wrong use of inheritance
- ☐ poor choice of classnames
- ☐ coupling of class 'Traveler' is too high

Maintenance

20:
D

*34

Consider the designs below:



These are designs for sending mails to large groups of employees.

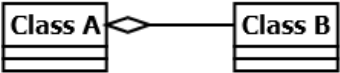
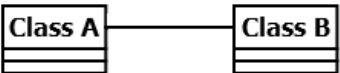
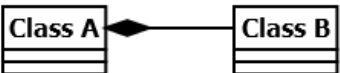

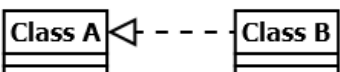

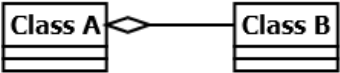
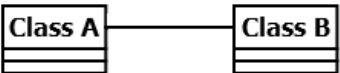
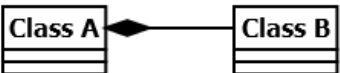

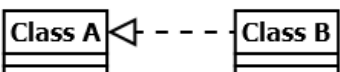

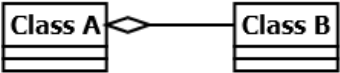
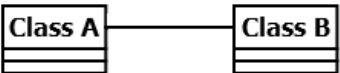
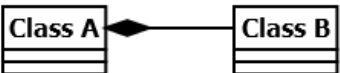

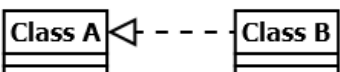

Q: which design is better (left : A , right: B)?

Choose one of the following answers

- ☒ A because it has better cohesion of attributes in Employee
- ☐ B because the use of inheritance make the design easy to extend
- ☐ A because it is more reusable
- ☐ B because class Addressee prevents direct access of Emailer to class Employee

Reuse,
separation of
concerns

Appendix B UML knowledge quiz

1	<p>*8 A class diagram can be used to model following things except Choose one of the following answers</p> <p><input type="radio"/> interaction between entities</p> <p><input checked="" type="radio"/> implementation of entities</p> <p><input type="radio"/> states and behavior of entities</p> <p><input type="radio"/> relationships between entities</p>																																																															
2 to 7	<p>*9 Which symbol specifies the visiability of a ... class member?</p> <table border="1"> <thead> <tr> <th></th><th>Public</th><th>Private</th><th>Protected</th><th>Package</th><th>Derived</th><th>Static</th></tr> </thead> <tbody> <tr> <td>#</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>-</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>!</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>~</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>/</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>^</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>+</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>not shown</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> </tbody> </table>		Public	Private	Protected	Package	Derived	Static	#	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	-	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	!	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	~	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	/	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	^	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	+	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	not shown	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
	Public	Private	Protected	Package	Derived	Static																																																										
#	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
-	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
!	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
~	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
/	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
^	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
+	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
not shown	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																										
8 to 11	<p>*10 Which instance level relationship is represented in each of the following diagrams?</p> <table border="1"> <thead> <tr> <th></th><th>Unidirectional Association</th><th>Aggregation</th><th>Composition</th><th>Reflexive Association</th></tr> </thead> <tbody> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> <tr> <td>not shown</td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td><td><input type="radio"/></td></tr> </tbody> </table>		Unidirectional Association	Aggregation	Composition	Reflexive Association		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>		<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	not shown	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																							
	Unidirectional Association	Aggregation	Composition	Reflexive Association																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												
not shown	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>																																																												

