ISSN: 2167-1435
v29 n1936

Software Engineering Frameworks: Life Cycle Changes

Kirby McMaster
kmcmaster@weber.edu
CSIS Dept, Fort Lewis College
Durango, CO 81301, USA

Samuel Sambasivam
ssambasivam@apu.edu
CS Dept, Azusa Pacific University
Azusa, CA 91702, USA

Stuart Wolthuis
stuart.wolthuis@byuh.edu
CIS Dept, Brigham Young University-Hawaii
Laie, HI 96762, USA

Abstract

This research examines frameworks developed by Computer Science and Information Systems students during a two-semester sequence in Software Engineering. The second semester course is project-based, where students work as teams to produce a software product. A questionnaire listing 60 Software Engineering concepts was given to students at three universities at the end of the second semester course. The concepts were chosen to span the software development life cycle. Students were asked to rate the importance of each concept on *two* Software Development scales--an *Early* scale (analysis and design) and a *Later* scale (implementation and beyond). From the responses, we calculated the average perceived importance for each concept, both Early and Later in the life cycle. We then examined how the relative importance of each concept changes and how student Software Engineering frameworks evolve during software development.

Keywords: Software Engineering, framework, life cycle, concept, rating.

1. INTRODUCTION

Learning is more effective if course topics are organized within an overall mental *framework*. A framework provides a way to fit concepts together into a meaningful whole. According to Donald (2002), a course needs a framework to improve understanding.

If we are to understand the relationships between concepts, we need to know in what order and how closely concepts are linked and the character of the linkage.

Bain (2004) argues that instructors should provide frameworks for their courses and not rely on students to construct their own.

When we encounter new material, we try to comprehend it in terms of something we think we already know. ... Even if [students] know nothing about our subjects, they still use an existing mental model of something to build their knowledge of what we tell them.

Frameworks are common in many Information Systems and Computer Science fields. Some frameworks involve a series of layers, in which services are provided to a layer by the layer below it. Operating systems often include layers such as: user-interface, API, OS processes, microkernel, and device drivers (Tanenbaum, 2008; Silberschatz, Galvin, & Gagne, 2003). Database systems provide views of data at the internal physical level, the logical schema level, and user view levels to simplify access to data (Connolly & Begg, 2009; Date, 2004). Network functions are divided into layers based on the OSI model and the Internet Protocol Suite (Peterson & Davie, 2011; Comer, 2009).

Not all computing frameworks are layered. Object-oriented programs consist of modular classes and methods that interface with each other (Lafore, 2001). Computer hardware systems synchronize various subsystems to provide input, computation, output, and storage (Patterson & Hennessy, 2008).

In Software Engineering (SE), the most common framework is based on the classical software development life cycle, where systems are constructed through a set of stages that are performed in sequence or iteratively (Pressman, 2009; Sommerville, 2010). Schach (2010) focuses primarily on object-oriented methods for software development. Recent SE approaches try to integrate sound management practices with agile methods (Cohn, 2009).

In this research, we sought to determine what mental frameworks *students* had formed by the end of their second Software Engineering course. We also attempted to measure how these frameworks had evolved over a two-semester SE course sequence. During this sequence, much of each student's work was performed as part of a team, in which a complete software product was developed.

The next section describes our methodology for collecting data on student ratings of Software Engineering concepts.

2. METHODOLOGY

A questionnaire listing 60 Software Engineering concepts was given to Computer Science and Information Systems students upon completion of their second SE course. The concepts were selected from a variety of sources. We chose SE concepts that we (the authors) felt were important, concepts that were emphasized in SE textbooks, and concepts that were recommended by other SE instructors.

We tried to include concepts that spanned the stages of the software development life cycle, such as *requirements*, *prototype*, *structured programming*, and *deployment*. We chose SE concepts that represented various categories and groupings, such as:

1. current approaches (e.g. *agile methods*, *iterative development*, *extreme programming*),
2. common development activities (e.g. *system integration*, *training*, *validation*),
3. specific deliverables (e.g. *class diagram*, *hierarchy chart*, *test plan*),
4. software product characteristics (e.g. *cost*, *quality*, *usability*),
5. work environment features (e.g. *CASE tools*, *project management*, *version control*).

Unlike an earlier study (McMaster, Hadfield, Wolthuis, & Sambasivam) in which all but 1 of 64 Software Engineering concepts were defined by a single word, in this study 36 of the 60 concepts were presented using 2+ words (or acronyms). We did not expect all students to perceive the concepts identically. However, the more complete description of concepts provided by multiple words, plus the tangible nature of activities (*version control*) and documents (*class diagram*), should make the student ratings of concept importance in SE more comparable.

Once the concept list was compiled, the items were randomized so that there would be no implied significance to the order in which the concepts were presented to students. The concepts were organized into two columns, with 30 items per column, so that the entire questionnaire would fit on one page.

To identify which SE concepts were valued most, students were asked to rate the importance of each concept on *two* Software Development scales--an *Early* scale and a *Later* scale. The Early scale represents the importance of

concepts during analysis and design. The Later scale represents the importance of concepts from implementation and beyond. The Early and Later scales were 4-point scales, with values:

3 = most important,
2 = important,
1 = less important,
0 (or blank) = unimportant.

A copy of the questionnaire is included in Appendix A.

The questionnaire was presented to second-semester Software Engineering students at three schools. The School1 sample consisted of 13 students at a small state university. The School2 sample included 11 students from a small private college. The School3 sample of 20 students was drawn from a larger private university. The combined sample size was 44. Almost all students were juniors or seniors. The course sections had different instructors and textbooks, but each group of students received a traditional, project-oriented SE course.

We first analyzed the response patterns displayed on the questionnaires to assess the validity of the data. We then calculated the average perceived importance (Early and Later) for each concept at each school and for the combined sample. Next, we looked at concept rating similarities and differences between schools. Finally, we examined changes in concept ratings when moving from the Early scale to the Later scale.

3. RESPONSE PATTERNS

An initial screening of the data was performed to check if student response patterns were representative of valid measures for the Early and Later ratings. We examined the distribution of ratings values for the two scales to see if they were consistent with what might be expected.

Table 1 summarizes the Early and Later distributions of values (0-3) across all 44 students and all 60 items. The number of scores included in the table for each scale is 44 x 60 = 2640.

Table 1. Early and Later Ratings Distributions.

Rating	Early	Later
0	566 (21.4)	720 (27.3)
1	433 (16.4)	322 (12.2)
2	845 (32.0)	821 (31.1)
3	796 (30.2)	777 (29.4)
Total	2640 (100%)	2640 (100%)

If students randomly picked a value between 0 and 3 for each concept (either scale), we would expect about 25% of the 2640 scores (=660) to appear for each scale value. According to Table 1, for the Early scale, 32% of the values are 2, and 30% are 3. The percentages are lower for Early values 0 and 1. A similar result holds for the Later scale, but with more zeros.

The larger number of 2's and 3's can be explained by the fact that concepts were placed on the questionnaire because they were considered relevant to Software Engineering. The larger number of 0's compared to 1's requires a different explanation.

Frequency of Zeros by Scale

There are several possible reasons for the abundance of zeros in the data. A zero represents "unimportant". To reduce the amount of writing required in rating 60 concepts on two scales (120 values), we allowed students to leave an item blank if it was considered unimportant, which equates to a 0 score. One explanation for the large number of zeros is that students might not have fully deliberated when rating items of little importance. Instead of choosing between a 1 and a 0, they just left the item blank.

Figure 1 summarizes the number of zeros recorded by students on the Early and Later scales. The categories on the horizontal axis are intervals for the number of zeros. The vertical bars depict the number of students whose zero count fell within the intervals. Results are presented separately for each scale.

ISSN: 2167-1435
v29 n1936

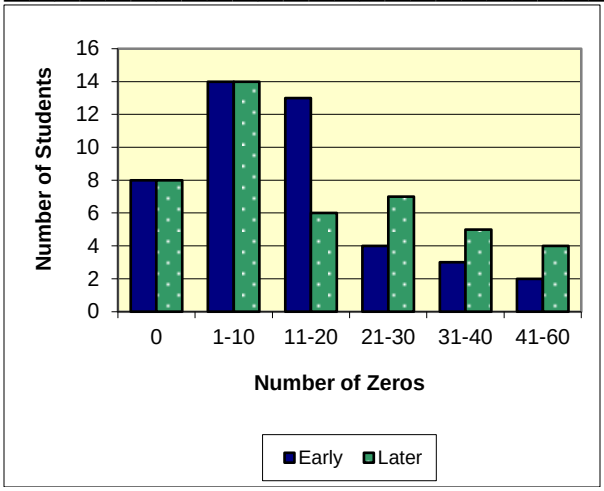

Figure 1. Total Zeros for Students.

If the expected number of zeros were approximately 25% (of 60 items), then the number of zeros recorded by a student on a scale should be about 15. With reasonable variability, the frequency should rarely exceed 30. This barrier of 30 was exceeded by only 5 students for the Early scale, but by 9 students on the Later scale. If the Later scores were recorded after the Early scores, then "fatigue" may have contributed to the larger number of Later zeros (blanks).

Frequency of Zeros by Column

To allow the entire questionnaire to fit on one page, the 60 concepts were arranged in two columns of 30. Assuming that the second column of items were rated by students after the first column, there is a possible column "fatigue" effect. This column effect may have led to additional zeros (blanks) being assigned to second column concepts.

Table 2 presents a breakdown of zero counts for the first column vs. the second column of items. Results are shown separately for each scale.

Table 2. Zero Summary by Early/Later Scales and Questionnaire Columns.

Column	Early	Later
1: Items 1-30	262 (46.3)	333 (46.2)
2: Items 31-60	304 (53.7)	387 (53.8)
Total	566 (100%)	720 (100%)

On each scale, 46% of the zeros apply to first column concepts. The remaining 54% occurred in the second column items. This pattern is identical for both the Early and Later scales.

We conclude that the sample data likely includes more zeros than might have occurred if the questionnaire and its administration were changed. One way to improve the questionnaire would be to require students to record a 0 (and not allow blanks) for "unimportant" items. Alternatively, we could change the scale values to 1 (Unimportant) through 4 (Most Important). Another methodology improvement would be to prepare a "script" that provides better instructions for administering and filling out the questionnaire.

The oversupply of zeros in the data should not invalidate the results of the study. From Figure 1, only 2 of the Early student sets of ratings and 4 of the Later sets had over 40 zeros. Some of the "fatigue" zeros probably should have been 1's, which would have made the average concept ratings slightly higher. These zeros shouldn't bias the relative importance of the concepts.

4. CONCEPT RATINGS

The primary objective of this study was to determine how Software Engineering frameworks evolve as students move through a two-semester course sequence. In particular, we wanted to measure how working on a team project affects a student's perception of Software Engineering.

A framework is more than a set of concepts. It provides a rationale for how the concepts fit together and how they interact. Nevertheless, our data analysis focused on determining which SE concepts are perceived as most important both early and later in the software development life cycle.

For this purpose, we calculated an average Early rating and an average Later rating for each concept at each school. We then combined these school averages into unweighted overall means for each item. We did not compute weighted averages because the School3 sample was almost twice the size of the other two schools. To make the values easier to read (no decimal points), we rescaled the averages by multiplying times 100. Thus, the original scale scores of 0 to 3 were transformed into averages between 0 and 300.

Early Ratings

The 20 Software Engineering concepts considered most important by students in the Early stages of development are listed in Table 3.

The concepts are presented in order of *decreasing* overall means. We include the averages for each school to provide an initial impression of the consistency of individual school ratings. Clearly, the concept considered most important in the early stages of development was *requirement*, with an overall mean rating of 271 (out of 300). Students invariably spend a substantial amount of time defining requirements at the beginning of their team projects.

The next five Early concepts--*teamwork*, *project management*, *cost*, *schedule*, and *documentation*--have combined averages above 230. Of these concepts, all except *documentation* have averages above 200 at each school.

Table 3. Top-20 Early Mean Ratings (x100).

SE Concept	Sch 1	Sch 2	Sch 3	All
requirement	285	264	265	271
teamwork	269	227	245	247
project mgmt	254	200	270	241
cost	246	218	240	235
schedule	262	209	230	234
documentation	238	273	185	232
class diagram	231	200	240	224
customer	246	209	200	218
E-R diagram	238	209	200	216
S/W architecture	231	218	195	215
problem definition	223	264	155	214
flowchart	215	218	205	213
data flow diagram	185	227	220	211
specification	215	218	190	208
objects & classes	200	173	235	203
quality	215	182	210	202
data structure	200	191	190	194
program language	215	182	175	191
use case	223	109	240	191
functions & proc	192	145	210	183

The above Early concepts emphasize general work environment structure and features. Items further down the list mention specific Early deliverables, such as *class diagram*, *entity-relationship diagram*, *flowchart*, *data flow diagram*, and *use case*. It is encouraging to see concepts such as *customer* and *quality* on the Early important list.

Later Ratings

The Top-20 Software Engineering concepts considered most important during the Later stages of development are listed in Table 4.

Again, the concepts are presented in order of *decreasing* overall means. We also include the averages for each school. The concept considered most important during the Later phases of development was *performance*, with an overall mean rating of 242. Performance was probably an after-thought in the early stages of development, but awareness of the need for performance increases as students work to complete their projects.

The next five Later concepts--*teamwork*, *quality*, *deployment*, *usability*, and *documentation*--have combined averages above 220. Of these concepts, all except *usability* have averages above 200 at each school. The top-6 Later concepts include two that were also on the Early top-6 list--*teamwork* and *documentation*.

Table 4. Top-20 Later Mean Ratings (x100).

SE Concept	Sch 1	Sch 2	Sch 3	All
performance	269	227	230	242
teamwork	246	218	250	238
quality	262	227	200	230
deployment	231	245	205	227
usability	262	227	185	225
documentation	231	200	235	222
GUI	231	209	205	215
customer	223	209	205	212
source code	208	255	165	209
test plan	238	182	190	203
schedule	215	200	190	202
cost	192	191	220	201
system integration	169	209	220	199
verification	177	218	195	197
training	177	182	230	196
end user	215	164	205	195
project mgmt	223	155	205	194
version control	208	255	120	194
database	185	191	195	190
objects & classes	192	218	150	187

The other four concepts moved up in importance, sometimes dramatically, in the Later stages of development. Quality should

ISSN: 2167-1435
v29 n1936

probably have been rated higher on the Early scale, as well.

The Later list includes several other concepts in common with the Early list, but at lower levels of importance--*customer*, *schedule*, *cost*, and *project management*. Activities that receive more emphasis while the actual software is being constructed include *source code*, *test plan*, *training*, *system integration*, and *version control*.

The Early list includes 16 concepts with an average rating above 200. The Later list includes only 12 such concepts. One reason for the lower averages on the Later scale is the larger number of 0 ratings, as described in the previous section.

Taken together, the concept ratings for the Early list and the Later list can be defended as reasonable in terms of face validity. Highly-rated Early concepts *requirement* and *problem definition* have much lower ratings on the Later scale. Highly-rated Later concepts *performance* and *deployment* have much lower ratings on the Early scale. Concepts such as *teamwork* and *documentation*, which are important throughout development, are rated high on both scales.

A complete list of the overall Early and Later ratings is included in Appendix B. In this list, the concepts are presented in the order they appear on the questionnaire.

5. COMPARISONS AMONG SCHOOLS

The Early and Later concept ratings shown in Table 3 and Table 4 are fairly consistent across the three schools, at least for the higher-rated concepts. The four Early and Later concepts having the highest overall average ratings also have individual school ratings above 200.

Between-School Correlations

One measure of school-to-school consistency is the correlation between average ratings for the 60 items on the questionnaire. The Early and Later correlations for each pair of schools are summarized in Table 5.

Table 5. Ratings Correlations Between Schools.

	Sch1 vs. Sch2	Sch1 vs. Sch3	Sch2 vs. Sch3
Early	0.663	0.637	0.540
Later	0.722	0.720	0.504

The correlations between Early ratings for each pair of schools varies from 0.540 to 0.663. For the Later scale, the two correlations involving School1 exceed 0.720, while the School2-School3 correlation is barely above 0.500 (the lowest in the table). Students at School3 apparently perceive Software Engineering somewhat differently than students at the other two schools.

In all cases, the correlation values suggest a moderately strong positive relationship between the average ratings for the three schools.

School Ratings Profiles

In Table 3, we listed the top-20 SE concepts having the largest combined Early ratings, along with the Early ratings for each school. We now provide a visual representation of school-to-school variation in Early ratings for all 60 concepts.

Figure 2 is a line graph of the Early concept ratings for each school. Concepts are ordered by decreasing Early combined rating. The first 20 concepts, starting at the left, are the Table 3 concepts, starting with *requirement*.

The chart has a separate "line" for each school, plus a bold line for the combined sample. This figure presents the ratings pattern for each school as a *profile*. In most cases, the successive differences between concept means for the schools has the appearance of random variation.

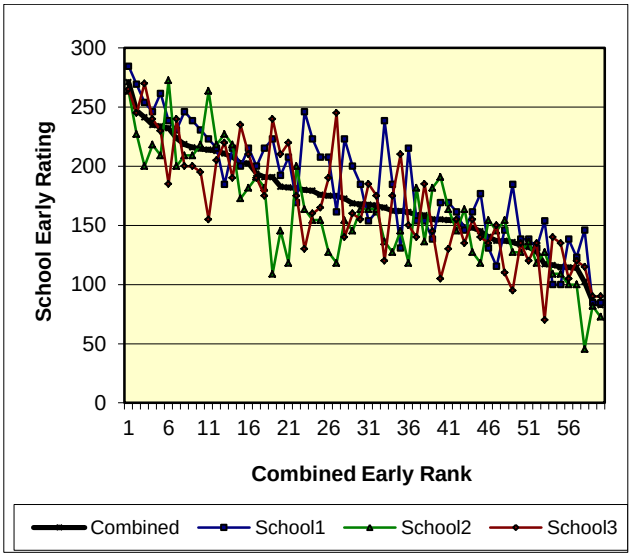

Figure 2: Early Concept Ratings Profiles.

ISSN: 2167-1435
v29 n1936

The concepts with the largest between-school ratings variation, as measured by the range, are: *use case* (rank 19, range=131), *UML* (rank 27. range=127), *training* (rank 33, range=118), *algorithm* (rank 23, range=116), and *problem definition* (rank 11, range=109). Note that the largest variation seems to occur for concepts with Early ratings in the middle of the scale.

Ratings variation between-schools for the Later scale is presented as a similar line graph profile in Figure 3.

Concepts are ordered by decreasing Later combined rating. The first 20 concepts, starting at the left, are the Table 4 concepts, starting with *performance*. For almost all of the Later ratings values, the successive differences between concept means for the schools appear to be random. The only concept with large between-school ratings variation is *programming language* (rank 18, range = 135). Overall, the between-school variation for Later ratings is noticeably smaller than for Early ratings.

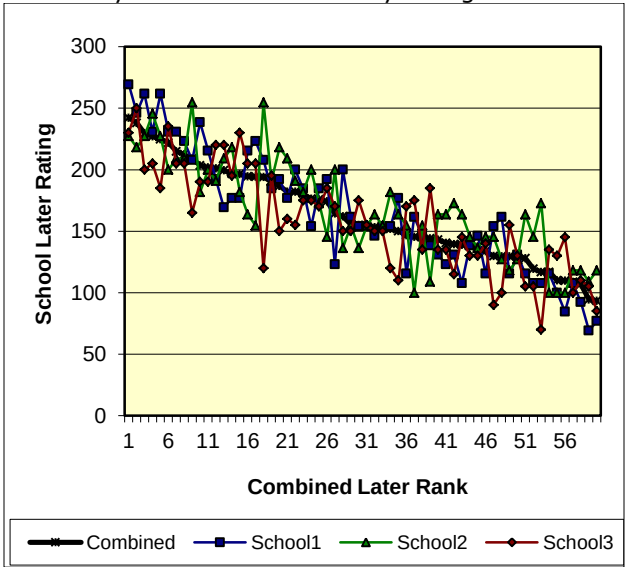

Figure 3: Later Concept Ratings Profiles.

We propose a model in which a school's Early and Later concept ratings are due primarily to a combined ratings component, plus a random component. The model includes a third component consisting of a small number of concepts that received special emphasis at a specific school. To support this model, we present in Table 6 the Early and Later correlations for each school with the combined sample.

Table 6. Ratings Correlations: Schools vs. Combined.

	Sch1 vs. Combined	Sch2 vs. Combined	Sch3 vs. Combined
Early	0.891	0.852	0.842
Later	0.939	0.841	0.844

The correlations for Early ratings between each school and the combined sample vary from 0.84 to 0.89. For the Later scale, the correlations fall between 0.84 and 0.94. These are very high values, even knowing that the combined ratings are formed from the individual school ratings. These high correlations result from the combined ratings component. The remaining variation includes the random component and, possibly, a few selected concepts.

Top Early Concepts at Each School

Another way to compare concept ratings between schools is to examine the highest-rated Early and Later concepts at each school. Table 7 presents the top-12 Early concepts for the three schools, listed in order of decreasing ratings (not shown).

Four concepts (marked in **bold**) are included among the top-rated Early concepts at all three schools. These concepts are *requirement*, *teamwork*, *schedule*, and *cost*. Requirement is primarily an Early activity, but the other three concepts are important in all stages of development. At least there is agreement among the schools that these basic concepts are relevant early in the life cycle.

Table 7. Top-12 Early Concepts by School.

	School1	School2	School3
1	**requirement**	documentation	project management
2	**teamwork**	problem definition	**requirement**
3	**schedule**	**requirement**	**teamwork**
4	project management	data flow diagram	UML
5	algorithm	**teamwork**	class diagram
6	**cost**	**cost**	**cost**
7	customer	flowchart	use case
8	documentation	software architecture	objects & classes
9	E-R diagram	specification	**schedule**
10	training	customer	activity diagram
11	class diagram	E-R diagram	data flow diagram

12	software architecture	**schedule**	quality

Seven Early concepts were top-rated at two of the schools. This list includes *project management*, *documentation*, *customer*, *software architecture*, *data flow diagram*, *class diagram*, and *entity-relationship diagram*. The first three of these concepts contribute throughout the life cycle. The diagrams in this list are usually prepared during the analysis and design stages.

The remaining ten concepts in Table 7 were highly-rated by just one of the schools. Note that the early importance of *quality* was recognized only by School3.

Top Later Concepts at Each School

In a similar way, we compared ratings of Later concepts at each school. Table 8 lists the top-12 Later concepts for the three schools, again in order of decreasing ratings (not shown).

Six concepts (marked in **bold**) are included among the top-rated Later concepts at all three schools. These concepts are *performance*, *teamwork*, *quality*, *deployment*, *customer*, and *graphical user interface*. Deployment is clearly a Later activity. The desire for performance and the need for a suitable graphical user interface increases as development nears completion. It is heartening to see that School1 and School2 eventually joined School3 in recognizing the importance of quality.

Table 8. Top-12 Later Concepts by School

	School1	School2	School3
1	**performance**	source code	**teamwork**
2	**quality**	version control	documen-tation
3	usability	**deployment**	**performance**
4	**teamwork**	**performance**	training
5	test plan	**quality**	cost
6	**deployment**	usability	system integration
7	documen-tation	objects & classes	**customer**
8	**graphical user interface**	**teamwork**	**deployment**
9	**customer**	verification	end user
10	project management	**customer**	**graphical user interface**
11	end user	**graphical user interface**	project management
12	schedule	structured programming	**quality**

Three Later concepts were top-rated by students at two of the schools. These concepts are *documentation*, *project management*, and *usability*. Like performance, usability often receives more emphasis closer to product completion. The other two concepts also received high Early ratings by two schools (but not the same two schools).

The remaining twelve concepts in Table 8 were highly-rated by just one of the schools. In real-world development, the importance of *system integration*, *test plan*, *version control*, and *training* would be more appreciated. It is unclear why the ratings of *cost* and *schedule* diminished from Early to Later. We noted previously that the Later scale received more zero (blank) responses than the Early scale, which would contribute to reduced ratings.

There was a satisfying level of consistency in the Early and Later ratings for the three schools. The Later ratings showed more between-school agreement than the Early ratings. Hopefully, the efforts of instructors, along with the experience of working on team projects, contributed to the similarity of concept ratings.

6. RATINGS CHANGES: EARLY TO LATER

In this section, we describe how Software Engineering concept ratings change as students move through the software development process. First, we examine the concepts for which the *decrease* from Early rating to Later

rating was largest. Then, we look at concepts having the greatest *increase* from Early to Later ratings. Finally, we show the overall pattern of changes for all Early rating levels.

Ratings Decreases

Eleven concepts that showed the largest overall ratings *decreases* (between -109 and -53) from Early to Later are presented in Table 9. The items are listed from largest change to smallest (in magnitude).

Table 9. Largest Early-to-Later Ratings Decreases (x100).

SE Concept	Early	Later	Change
requirement	271	162	-109
flowchart	213	107	-106
data flow diagram	211	119	-91
hierarchy chart	224	145	-78
class diagram	173	94	-78
activity diagram	182	109	-73
specification	208	143	-65
entity-relationship diagram	216	153	-63
problem definition	214	153	-61
UML	175	117	-58
use case	191	138	-53

The ratings for two concepts--*requirement* and *flowchart*--decreased by more than 100 from Early to Later. Requirement had the highest Early rating, but fell to a Later rating below 200 (Important). The Later rating for flowchart dropped to just above 100 (Less Important).

Two analysis phase activities--*specification* and *problem definition*--with Early ratings above 200, had Later ratings in the mid-100's range. The remaining concepts in Table 9 are documents and deliverables that are prepared (often using UML) during analysis and design.

Ratings Increases

Eleven concepts showing the largest overall ratings *increases* (between 128 and 45) from Early to Later are presented in Table 10. Again, the items are listed from largest change to smallest.

The ratings of two concepts--*performance* and *deployment*--increased by more than 100 from Early to Later. Neither performance nor deployment had a high Early rating. The relevance of these two concepts, along with *source code*, *test plan*, *version control*, *verification*, and *validation*, becomes more evident as the software nears completion.

Table 10. Largest Early-to-Later Ratings Increases (x100).

SE Concept	Early	Later	Change
performance	114	242	128
deployment	102	227	125
verification	115	197	82
source code	132	209	77
graphical user interface	148	215	67
test plan	140	203	63
version control	136	194	58
extreme programming	82	141	58
usability	176	225	49
validation	128	176	48
system integration	154	199	45

In addition, even though *usability* and *graphical user interface* are design phase considerations, their impact on the system is more visible when actual code is running.

Ratings Change Patterns

It seems logical that a concept having a high Early rating will often receive a lower Later rating, because most of the Later scale is below the high Early value. By the same reasoning, a concept with a low Early rating is likely to increase on the Later scale. This pattern is sometimes referred to as "regression to the mean" (Barnett, van der Pols, & Dobson, 2005).

An illustration of the effect of "regression to the mean" is shown in Table 11. The 60 questionnaire items were grouped into six equal-size intervals based on the ranks (1-60) of their Early ratings. The table summarizes for each Early rating interval the number of Later ratings that increased vs. the number that decreased.

Table 11. Early-to-Later Ratings Changes.

Early Rank	Later Increase	Later Decrease
1-10	0	10
11-20	1	9
21-30	2	8
31-40	4	6
41-50	6	4

51-60	9	1
Total	22	38

For the ten concepts having the highest Early ratings (ranks 1-10), all of their Later ratings dropped (below the Early value). For the ten concepts with the lowest Early ratings (ranks 51-60), nine had a higher Later rating.

For an explanation of why more Early ratings decreased (38) than increased (22), this is partially due to the excess of zeros (mentioned earlier) recorded for the Later scale items.

7. CONCLUSIONS

Learning is more effective if course topics are organized within an overall mental *framework*. A framework provides a way to fit concepts together into a meaningful whole. In this research, a questionnaire listing 60 Software Engineering concepts was given to Computer Science and Information Systems students at three schools upon completion of their second Software Engineering course. Students were asked to rate the importance of each concept on *two* Software Development scales--an *Early* scale (analysis and design) and a *Later* scale (implementation and beyond).

For the combined sample, the concept considered most important in the Early stages of development was *requirement*. Students spend a substantial amount of time defining requirements at the beginning of their team projects. The next five Early concepts were *teamwork*, *project management*, *cost*, *schedule*, and *documentation*. The above concepts emphasize general work environment structure and product features. Other items include specific Early deliverables, such as *class diagram*, entity-relationship *diagram*, *flowchart*, *data flow diagram*, and *use case*.

The concept considered most important during Later development was *performance*. The need for performance increases as students work to complete their projects. The next five Later concepts were *teamwork*, *quality*, *deployment*, *usability*, and *documentation*. The top-six Later concepts include two that were also on the Early top-six list--*teamwork* and *documentation*.

Comparing the three schools, the higher-rated Early and Later concept ratings are fairly consistent. Among the top 12 Early concepts for each school, four concepts appear on all three lists. These concepts are *requirement*, *teamwork*, *schedule*, and *cost*. Similarly, six concepts are included among the top 12 Later concepts at all three schools. These concepts are *performance*, *teamwork*, *quality*, *deployment*, *customer*, and *graphical user interface*. Note that *teamwork* is on both lists.

Ratings changes from Early to Later showed a familiar pattern, sometimes referred to as "regression to the mean". Higher Early ratings were more likely to decrease during the Later stages. The two concepts with the largest ratings decreases were *requirement* and *flowchart*. Conversely, large increases from Early to Later ratings were more common for concepts with low Early ratings. The two concepts with the largest ratings increases were *performance* and *deployment*.

Software Engineering teachers can benefit from comparing concept ratings as summarized in this paper with their students' perceptions of most important concepts. Where there are differences, identify how you emphasize your favored concepts. You are encouraged to use the questionnaire in Appendix A to obtain feedback from your students.

Future Research

Future research plans include a replication of this study with larger samples and an improved questionnaire to verify our preliminary findings. Software Engineering instructors should be surveyed in a similar manner to discover which concepts they feel are most important. We would then be able to compare how closely student ratings match instructor ratings.

The focus of this research has been on concepts that form frameworks for Software Engineering. We have measured how 60 concepts relate to the software development life cycle. Our chosen concepts represent different development environments, management styles, methodologies, and project deliverables. Future research should explore other possible ways to integrate SE concepts into a unified Software Engineering framework.

8. REFERENCES

Bain, K. (2004). What the Best College Teachers Do. Harvard University Press, pp 26-27.

Barnett, A., van der Pols, J., and Dobson, A. (2005). "Regression to the mean: what it is and how to deal with it." *International Journal of Epidemiology* 2005;34:215–220.

ISSN: 2167-1435
v29 n1936

Cohn, M. (2009). Succeeding with Agile: Software Development Using Scrum. Addison Wesley.

Comer, D. E. (2009). Computer Networks and Internets (5th ed). Prentice Hall.

Connolly, T., and Begg, C. (2009). Database Systems: A Practical Approach to Design, Implementation and Management (5th ed). Addison Wesley.

Date, C. J. (2004). An Introduction to Database Systems (8th ed). Addison Wesley.

Donald, J. (2002). Learning to Think. Jossey-Bass, p 15.

Lafore, R. (2001). Object-Oriented Programming in C++ (4th ed). Sams.

McMaster, K., Hadfield, S., Wolthuis, S., and Sambasivam, S. (2012). "Software Engineering Frameworks: Textbooks vs. Student Perceptions." *Information Systems Education Journal*, 10(5) pp 4-14.

Patterson, D., and Hennessy, J. (2008). Computer Organization and Design (4th ed). Morgan Kaufmann.

Peterson, L., and Davie, B. (2011). Computer Networks: A Systems Approach (5th ed). Morgan Kaufmann.

Pressman, R. (2009). Software Engineering: A Practitioner's Approach (7th ed). McGraw-Hill.

Schach, S. (2010). Object-Oriented and Classical Software Engineering (8th ed). McGraw-Hill.

Silberschatz, A., Galvin, P., and Gagne, G.(2003). Operating System Concepts (6th e). Wiley.

Sommerville, I. (2010). Software Engineering (9th ed). Addison Wesley.

Tanenbaum, A. S. (2008). Modern Operating Systems (3rd ed). Prentice Hall.

ISSN: 2167-1435
v29 n1936

APPENDIX A

Software Engineering Concept Ratings Name _____

For each concept listed below, please rate the importance of the concept during *each* of the following Software Development phases:

Early - Analysis & Design **Later** - Implementation & Beyond

Ratings values can be:

3 - most important, **2** - important, **1** - less important, **blank** - unimportant.

Concepts can vary in importance from phase to phase.

Concept	Early	Later
flowchart		
documentation		
requirement		
schema		
pseudocode		
class diagram		
validation		
database		
source code		
formal methods		
data dictionary		
quality		
usability		
functions & procedures		
component/control		
test plan		
programming language		
structured programming		
hierarchy chart		
prototype		
CASE tools		
verification		
design pattern		
objects & classes		
information		
entity-relationship diagram		
performance		
cost		
refactoring		
relational model		

Concept	Early	Later
change management		
activity diagram		
version control		
maturity model		
extreme programming		
specification		
schedule		
graphical user interface		
data flow diagram		
algorithm		
customer		
framework		
data structure		
iterative development		
end user		
state diagram		
IDE		
project management		
system integration		
data model		
agile methods		
training		
teamwork		
software architecture		
process model		
modularity		
UML		
deployment		
problem definition		
use case		

ISSN: 2167-1435
v29 n1936

APPENDIX B

Software Engineering Concept Ratings:
Early and Later Ratings - Overall Averages (x100)

Concept	Early	Later
flowchart	213	107
documentation	232	222
requirement	271	162
schema	158	128
pseudocode	168	138
class diagram	224	145
validation	128	176
database	167	190
source code	132	209
formal methods	168	155
data dictionary	154	147
quality	202	230
usability	176	225
functions & procedures	183	180
component/control	116	154
test plan	140	203
programming language	191	176
structured programming	155	182
hierarchy chart	173	94
prototype	155	144
CASE tools	162	130
verification	115	197
design pattern	162	129
objects & classes	203	187
information	179	155
entity-relationship diagram	216	153
performance	114	242
cost	235	201
refactoring	85	130
relational model	161	139

Concept	Early	Later
change management	134	164
activity diagram	182	109
version control	136	194
maturity model	114	93
extreme programming	82	141
specification	208	143
schedule	234	202
graphical user interface	148	215
data flow diagram	211	119
algorithm	180	182
customer	218	212
framework	175	146
data structure	194	150
iterative development	145	130
end user	167	195
state diagram	137	110
IDE	117	117
project management	241	194
system integration	154	199
data model	181	134
agile methods	148	139
training	165	196
teamwork	247	238
software architecture	215	174
process model	159	110
modularity	137	152
UML	175	117
deployment	102	227
problem definition	214	153
use case	191	138