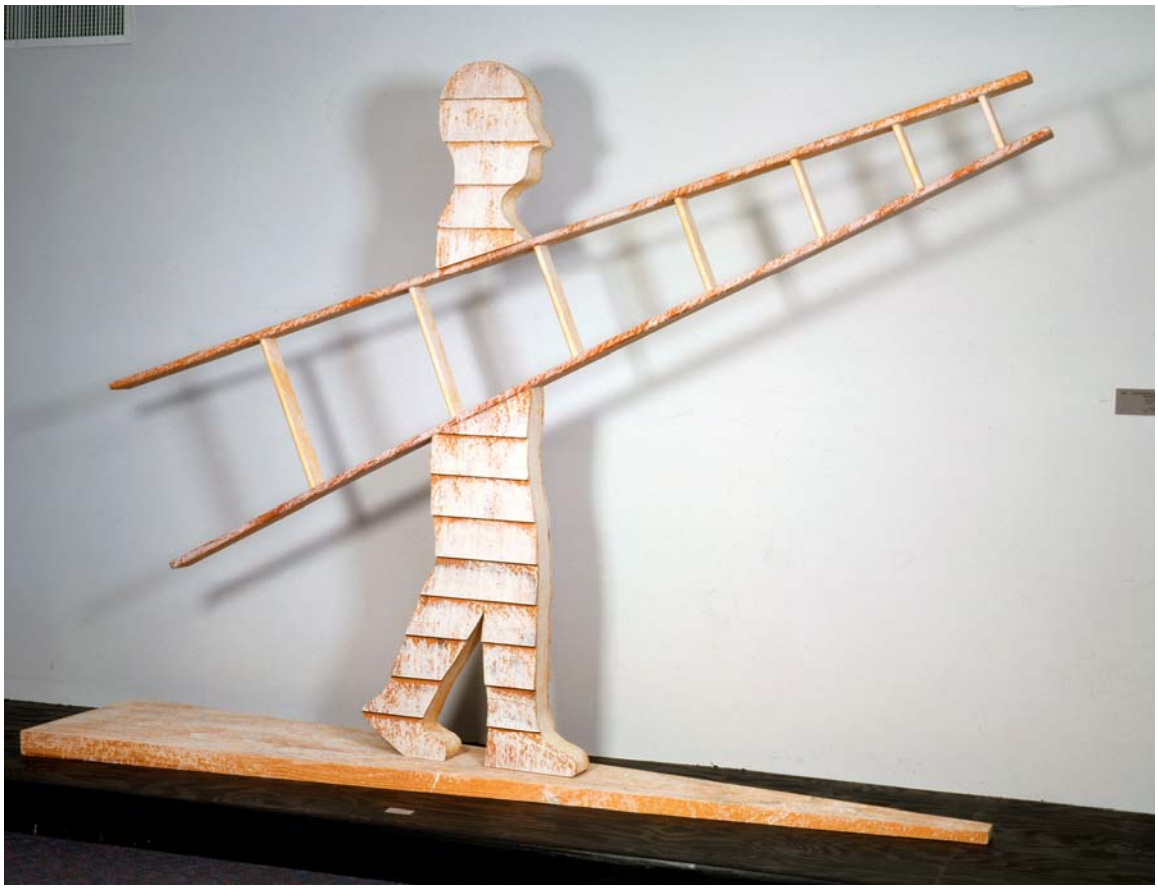
SCAFFOLDING RESEARCH
in Computer Science Education

The Experiment Kit



Experiment Kit: TOC

1. Question Formulation

General research question

This kit addresses understanding of software *design* and software design *criteria* used by undergraduate students at two levels: those we term 'first competency' programmers (using McCracken et al.'s formulation, ITiCSE, 2001), and those completing their Bachelor degrees (that is: those within the last eighth of a Bachelor degree program, or equivalent e.g. the last 6 months of a four-year, full-time program). The study aims to examine students' ability to recognise and generate structures underlying software designs, to elicit students' understanding and valuation of key design activities, and to examine whether students' understanding changes during the course of their undergraduate education.

Study's focal questions

The study is in two parts:

i) Can students decompose a proposed solution and describe the components and their relationships? Can students demonstrate through their decomposition and descriptions an understanding of fundamental concepts?

ii) Do students recognise different criteria within the design process? If so, what value do they place on those criteria, and what do they think their roles are within software design?

The study has an enveloping framework that compares data from first-competency and completing students and from educators. This supports comparative questions:

i) Does students' ability to decompose a proposed solution and describe it change or improve during the course of their CS education? Are students' decompositions and descriptions significantly different from educators'?

ii) Does students' relative prioritisation of design criteria alter over time, and how closely does their prioritisation approximate educators' prioritisation at each stage?

Subsidiary queries:

Are there differences between?

- high- and low-performing students?
- male and female students?
- students exposed to different first programming languages or paradigms?
- students exposed to different instructional approaches?
- educators presenting different programming languages or paradigms?
- educators taking different instructional approaches?

Is it possible to identify one or more typical progressions in design understanding, as characterised by performance of these tasks?
Are there groups whose responses more closely approximate educators' responses? If so, what distinguishes those groups?
Are students' responses more similar to their own educators' responses than to other educators' responses?

How able are students to incorporate domain knowledge in their solutions?
How attentive are students to design context (as suggested by the scenarios)?

Are there obvious breakdowns in students' design reasoning?
Are there any patterns of omission or breakdown in the performance of the decomposition task?
Are there indications of misconceptions or specific areas of ignorance?

Links to relevant theory

- Why care about software design? Software design is difficult: dealing with ill-defined and ill-structured problems; subject to complex and often conflicting constraints; producing large, complex, dynamic, intangible artefacts; and deeply embedded in a domain (cf. Goel and Pirolli's characteristics of the design task, 1992). Marshalling resources, applying knowledge, prioritising sub-tasks, managing constraints, evaluating proposed solutions, managing the design process itself are constituent and interacting skills—and potential sources of breakdown even in professional design behaviour (Guindon, Krasner and Curtis, 1987). These qualities make software design elusive to characterise and difficult to teach; hence, research is needed on how software design is understood and how design understanding develops.

- A key component of creativity is divergent thinking (Guilford, 1967), consideration of problems in terms of multiple solutions. It is complemented by convergent thinking, which narrows options to a single solution. Design reasoning requires a combination of divergent and convergent thinking: divergence to generate alternatives, and convergence to evaluate and prune alternatives. Divergent thinking is poorly understood and difficult to assess.

- Software design requires a variety of skills and knowledge: domain knowledge, software and computing knowledge and skill, design knowledge and skill (Soloway and Ehrlich, 1984). Importantly, it requires the ability to map between problems and solutions, between domain and software/computation. One consistent result in the expert-novice literature is that experts form abstractions based on deep (semantic, functional) characteristics rather than on surface (syntactic) characteristics (summarised in Kaplan et al., 1986; Allwood, 1986). Reasoning about deep structures allows experts to relate problems and solutions as reflections of a single schema or abstract structure.

- Various studies contribute to the notion that developing expertise is reflected in knowledge consolidation, and that consolidation of knowledge into meaningful conceptual structures is a reflection of 'deep learning' characterised by abstracted meta-knowledge. (For an overview, see, e.g., Eysenck and Keane, 1995; for "deep learning" see, Marton and Saljo, 1976).

- Models of software design, whether breadth-first decomposition or opportunistic, tend to involve decomposition into sub-problems, hence requiring management of the design process, including keeping track of the status of and relationships among sub-problems, and requiring integration of sub-problems into a coherent structure. Novices have been observed to differ from experts in their ability to decompose a problem effectively, to solve sub-problems, and to integrate solutions (e.g., Jeffries et al. 1981). One consistent result in the expert-novice literature is that experts organise information differently from novices, producing different and larger chunks (summarised in Kaplan et al., 1986; Allwood, 1986). Decomposition has the potential to provide insight into chunking and hence into how software designers organise and structure their proposed solutions.

Study's approach

This study uses two tasks to explore students' understanding of the software design process:

i. *a decomposition task*, to examine students' ability to analyse a problem and design an appropriate solution structure, and to assess students' understanding-in-action of fundamental concepts,
ii. *a design criteria prioritisation task*, to elicit which criteria students consider most and least important in the design process, in the task undertaken and for different design scenarios.

Assumptions:

- By decomposing a solution, students are indicating the main structure of the solution.
- The description of components reveals something about subjects' understanding of software design concepts, such as cohesion and coupling. Saying *anything* about such underlying concepts reveals some knowledge of them, and identifies them as important to the student's understanding.
- Prioritisation of design criteria reveals something not only about what is valued, but also about what is not valued.
- Different prioritisations for different scenarios, reveals an understanding that design is a contextual activity and that criteria change with context.
- Differences in prioritisation between different groups (e.g., naïve and expert groups) indicates differences in their values, beliefs, and understanding.

Justification of approach:

- **Paradigm independence**
  The use of a generic problem and generic stimuli makes the tasks paradigm independent, so that comparisons can be made across paradigms, languages, and pedagogic styles.
- **No imposition of existing overview**
  The general aim of this study is to attempt to elicit knowledge which is not constrained by a pre-defined scheme or by syntax or concepts specific to any one programming language. Therefore, the tasks are elicitative, to see if students recognise underlying structures. We make no imposition of existing overview – we don't assume any particular model of design.
- **Triangulation**
  The study combines different approaches and collects both qualitative and quantitative data, in order to provide opportunities to contradict or corroborate within the study, by comparing the different data.
- **Basis in survey of educators**
  The tasks and stimuli draw on a survey of software design educators, in order to focus the study on design knowledge considered relevant in CS education. The survey included educators from a variety of stances and so identified design issues and criteria underpinning all software design, regardless of paradigm or language.

2a Data collection specification

Collection of background data:

You need access to students' academic records.
If you need to recruit subjects from another institution, you will need a collaborating academic within that institution. Work with the academic ***early*** to complete Human Subjects Approval at that institution.

For all students, from their academic records:

- Identify all of their CS courses. (For students who have transferred from another institution, use only the grades from your own institution.)
- Average the grades for those courses.
- Record both the average and standard deviation.
- Map the average into a "performance bucket" (online, from December 2003).

Data collection from subjects:

Minimum data collection:

10 first-competency programmers (using McCracken et al.'s formulation, ITiCSE, 2001)
10 students completing their Bachelor degrees (that is: those within the last eighth of a Bachelor degree program)
2 educators (full-status faculty members, preferably those who teach software design)
If you don't have suitable subjects at your own institution, then visit a colleague at another institution and interview their students.

Try to complete all subjects within each constituency (e.g. first competency) within one month.

Time allowance:

- Allow at least ten minutes for the completion of the preliminary forms (human subjects and background questionnaire).
- Allow at least an hour per subject for the tasks. All of our pilot subjects completed the tasks within an hour (some well within), but it's safer to leave a margin.
- You will also need to take the time to label all your materials (tapes, notes, subjects' responses) carefully, and to write down any extra observations that strike you at the time.

2b Details/Phrases that may be useful with regard to Human Subjects Approval forms

This research is part of an international, multi-site project to investigate "first competency" programmers' and graduating students' conceptions and constructs of software design. Each subject will be asked to undertake a decomposition task followed by a design criteria elicitation exercise. These will, together, take *circa* one hour. During the sessions, data arising from the decomposition task and subsequent design criteria elicitations, will be captured by written notes and audio recording.

Subjects will be drawn from:

- students enrolled in computer science courses
- staff who teach them

Their age range will be 18-65. Students will be selected to represent different levels of achievement in computer programming courses.

Personal data – age, gender, institution and academic grades – will be associated with the elicited material. The name (or other identifiable data, such as student number) of participants will be known to internal investigators, but will not be stored or made available to researchers outside of this institution.

RESEARCH SUBJECTS' INFORMATION SHEET

You have been asked to participate as a subject in a study that is part of a multi-site international research project investigating design in software. This research is designed to investigate "first competency" and graduating students' approach to, and understanding of, software design. You will be asked to undertake short design task, followed by a card sort exercise. These will, together, take *circa* one hour. During the exercises, we will take notes, and ask that we may tape-record the session.

Some personal data – your age, gender, institution and academic achievement – will be associated with the design task and card-sort material. However, neither your name (nor any other identifiable data, such as student number) will be stored, nor made available to researchers outside of this institution. All data gathered will be used solely for the purposes of this research project.

You may obtain answers to any pertinent questions about this research by telephoning <insert name> on <insert telephone number>_during the following times: <insert availability>

If you decide not to participate, your refusal will involve **no** penalty and **no** loss of benefits to which you are otherwise entitled.

Participation in this study is voluntary, and you may withdraw your consent to participate at any time without penalty.

You have the right to receive a copy of any consent form that you sign and of any written consent documentation information that is used in obtaining your consent.

In order not to bias subsequent interviews, please do not discuss details of the tasks with other students.

Human Subjects Research Consent Form

Letter of Informed Consent

I, (print name in full) _____ am a student registered at <insert name of institution>. In signing this consent form, I agree to volunteer in the research project being conducted by <insert your name here> between <enter dates here>. I understand that the research being conducted relates to the approach to, and understanding of, software design. I understand that data from the design task and associated design criteria elicitation will be used in aggregate, and that excerpts from tape-recorded verbal communications with the researcher will be studied and may be quoted in papers, journal articles and books that may be written by the researchers.

I grant authorization for the use of the above information with the full understanding that my anonymity and confidentiality will be preserved at all times. I understand that my name or other identifying information will never be disclosed or referenced in any way in any written or verbal context.

I understand that my participation is entirely voluntary and that I may withdraw my permission to participate in this study without explanation at any point up to and including, the last day of April 2004.

Signature
Date

2c Background questionnaire

For each institution:

- Characterise your context. This should include:
    - type of institution (e.g., university, liberal arts college, polytechnic, community college, etc.)
    - characterisation of intake (e.g., entrance requirements, students' focus, age range, etc.)
    - instructional structure of the course(s) (e.g., do they have labs) and
    - pedagogic structure of the course(s) (e.g. breadth first, uses an environment that assists "design thinking" – e.g. BlueJ etc.).
- Describe the point of "first competency" intervention (that is, "end of first year"; "half way through second year" etc.)
- Characterise the paradigm you teach in for "first competency" and graduating students, in 2003 (perhaps the "first competency" will only have had Scheme, but the graduators will have had additionally Java, C etc.)

For each subject:

- Note their:
    - Age (student subjects only. Do not record age for educators).
    - Gender.
    - Program enrolled in (with major, if known).
- Ask them to self-rate their programming experience on the form provided
- Assign each subject a unique identifier of the form: F01 (First-competency Student 01) or G01 (Graduating student) or E01 (Educator 01), appended to your institution code (thus the first "first-competency" subject for Poppleton University would be HF01):

If you use subjects from another institution, request an additional code.

Programming Experience

On a scale of 1 (never used) to 5 (have used a lot) please rate your familiarity with the following programming languages. For "other", please indicate specific additional languages.

Please indicate if you have had formal instruction any of these, and for how long (1 semester, a year etc.).

	1	2	3	4	5	Formal instruction?
Java						
C++						
C						
Ada						
Scheme						
Pascal						
Visual Basic (VB)						
Other languages. Please specify each on a separate row						

2d Discriminator Question

You should choose "first competency" subjects from your undergraduate cohort at the point you believe them to be capable of undertaking at least one question within the following McCracken task set (as used in *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students* ITiCSE 2001). You do not have to give these questions to the students, just believe them able to undertake them. You should time your intervention within the course, or term, in which they would reach "first competency".

Assessment of Programming Skills of First Year CS Students: Problem Set

Enclosed in this file are the three problems. To maintain consistency, the problems (no matter which you choose) should be given in the following manner.

- This is individual work.
- The work is to be done in a closed lab (meaning proctored with the students doing the work in the allotted time).
- The student's job is to produce a working, tested, etc., program in the time allotted.
- This is a programming exercise, the expectations are that the students will produce a program. Any design documentation, though important to solving the problem, is not important to this assessment.
- What follows can be cut and pasted as the assignment. Note the introduction is applicable to all three problems.

Introduction (10 minutes to read and understand)

There are two main notations for entering information into hand-held calculators; Hewlett Packard calculators for the most part implement a Reverse Polish Notation (RPN), or "post-fix" notation, whereas Texas Instruments calculators implement the traditional "infix" notation. The major difference in the two styles is the order in which the mathematical operations are entered and executed.

RPN is a stack-based method of processing input. Numbers are read in and processed in the reverse order in which they are entered, while the operations themselves are read in and processed sequentially. "Infix" notation is the traditional method that most modern textbooks have adopted. It is mainly a symbol-based method of processing mathematical expressions, making use of a set of precedence rules defining the order in which values are processed by encapsulating operators and symbols.

RPN has a clear advantage over "infix" notation because RPN offers a simple implementation, naturally avoids ambiguity, and does not require parentheses. However,

unless the user is well practiced in the art of RPN, it is not automatically intuitive in its use. For example, the "infix" notation for a particular formula is:

-(4 + (6 * (-2)) - 15) (1)

However, in RPN, the same expression yields the following formula:

15 4 -2 6 * + - ~ (2)

In RPN, the numbers on the bottom of the stack are processed first, and then "work their way up" as the values are reduced toward the top. Clearly, the more intuitive format from an academic standpoint is "infix" notation (Equ. 1). However, you must deal with issues of precedence; in general, this is not as easy to program as the RPN-style of input (Equ. 2).

Other examples of "infix" and RPN style expression pairs are as follows:

4 * -(4 + 10 / 2.5 – 8) (3)
4 8 4 2.5 10 / + - ~ * (4)

17.2 + 21 * 3 / (-7) + 21 (5)
21 17.2 -7 3 21 * / + + (6)

((-(2 + 3 * 8)) ^ 4) * 7 (7)
7 4 2 8 3 * + ~ ^ * (8)

Problem Set #1: Programming an RPN Calculator

DIFFICULTY LEVEL: 1

Problem Statement:

You must write a program that reads, parses, and solves RPN-styled equations. You may assume that input will come directly from a terminal's standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your RPN expressions should be able to read the following operators. These operators' descriptions are provided for clarity.

- `+` Addition Operator. Adds two operands together.
- `-` Subtraction Operator. Subtracts two operands together.
- `*` Multiplication Operator. Multiplies two operands.
- `/` Division Operator. Divides two operands, in the order they are placed in the stack.
- `~` Inverse Operator (Negative Operator). Takes the inverse of the current value in the buffer.
- `^` Power Operator. Multiplies value in buffer by itself by a number of times specified by the operand that follows.

Each line of input will be entered in the relative form of:

{ numbers } { operators }

Each number and operator may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program can NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter 'q'. If there is an error with the input, the program should state such and begin accepting the next expression.

```
{unix: user: !} ./runprog
> 2 2 3 * +
ans = 8
> 4 5 2 ^ - ~ +
ERROR: Expression Invalid.
> 4 3 21 / - ~
ans = 3
> q
{unix: user: !}
```

Problem Set #2: Programming an "infix" Calculator w/o precedence

DIFFICULTY LEVEL: 2

Problem Statement:

You must write a program that reads, parses, and solves "infix"-styled equations. You may assume that input will come directly from a terminal's standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your "infix" expressions should be able to read the following operators. These operators' descriptions are provided for clarity.

+	Addition Operator. Adds two operands together.
-	Subtraction Operator. Subtracts two operands together.
*	Multiplication Operator. Multiplies two operands.
/	Division Operator. Divides two operands, in the order they appear.
^	Power Operator. Multiplies value in by itself by a number of times specified by the operand that follows.

Each line of input will be entered in the relative form of:

NUM OP NUM OP NUM …

Where NUM and OP represent Numbers and Operators, respectively.

Each number and operator may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program may NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter 'q'. If there is an error with the input, the program should state such and begin accepting the next expression.

This program does NOT have to consider precedence, nor contain any parenthesis. This will lead to some mathematically incorrect answers, but to keep things simple we will not be enforcing precedence. A sample session is below:

```
{unix: user: !} ./prog
> 2 + 3 * 4 – 5
ans = 15
> 2 + 3 ^ 2 * 4 * -1
ans = -100
>q
{unix: user: !}
```

Problem Set #3: "Infix" Calculator with simple precedence

DIFFICULTY LEVEL: 3

Problem Statement:

You must write a program that reads, parses, and solves "infix"-styled equations. You may assume that input will come directly from a terminal's standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your "infix" expressions should be able to read the following operators. These operators' descriptions are provided for clarity:

- ( ) Parenthesis. Used to group numbers and operators to give a simple order of precedence. Expressions contained within parentheses must be evaluated before expressions outside of parenthesis.
- ^ Power Operator. Multiplies value in by itself by a number of times specified by the operand that follows.
- * Multiplication Operator. Multiplies two operands.
- / Division Operator. Divides two operands, in the order they appear.
- + Addition Operator. Adds two operands together.
- - Subtraction Operator. Subtracts two operands together.

Each line of input will be entered in the relative form of the following examples:

```
> ( 2 + 3 ) * ( 5 * 2 * -1 )
ans = -50
> ( 3 + 1 * 2 ) ^ ( 2 + 1 )
ans = 512
> ( ( 3 + 1 ) * 2 ) ^ ( 9 / 3 )
ans = 512
```

Each number, operator, and parentheses may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program can NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter 'q'. If there is an error with the input, the program should state such and begin accepting the next expression.

This program does NOT have to consider operator precedence, but must consider parenthesis. This may still lead to some mathematically incorrect answers, but to keep things simple we will only be enforcing parenthesis precedence.

2e Specification of setup

You should make sure you conduct the experiment in a quiet room, where you will not be disturbed. Make sure you have:

- Human Subjects' Information Sheet and 2 copies of the Consent Form (one for them and one for you)
- The design brief
- The self-evaluation form
- The design criteria stimuli
- A working tape recorder
- Enough tapes & batteries
- Pens and pencils (in several colours)
- A supply of plain, blank paper
- If there isn't a clock in the room, make sure you take a watch

Sit so that you can see the subject (and the subject can see you!). Side by side, or across the corner of a desk is probably better than face to face across a table. It's fine – sometimes reassuring – for subjects to see the investigator's notes.

You should allow an hour for them to complete both tasks.

2f Experimenters' script

i) Consent Form and Background Questionnaire

ii) Decomposition Task

- Offer the subject the *design brief* and blank paper and pens/pencils. Note the time.
- Direct the subject "Please read all of the instructions. Please be sure to read the instructions in the box before starting."
- Invite them to ask any questions they want, and to take as long as they wish to complete the task. They are welcome to talk aloud, or to work silently, as they wish.
- When they feel they've finished, if they haven't already done so, ask them to put their solution on paper (words, diagrams, whatever form).
- When they've indicated the solution design on paper, note the time. Then ask:
  "Would you talk me through your design?" If they need another prompt: "Can you tell me what the parts of your solution are and what they do?"
  If they want to know what "parts" are, you can substitute other words: chunks, main pieces, components etc. But do not offer other prompts—we want to see what comes freely.
- If they offer new names, ask them to annotate their notes. Their spoken descriptions should be recorded verbatim by the experimenter —ask them to pause if necessary in order to allow you time to record their words. Their descriptions should be taken "as they are"— you should not lead the subject by asking for clarifications or elaborations.
- If there's any uncertainty, they should be asked to confirm how many parts they've identified.
- If they are unable to identify parts, you may ask "Have you read all of the instructions?" Capture their first answer, and the prompt, and any new information they offer.
- If they have not already identified relationships, interactions or flows between components, then you may ask elicitative questions that reflect their descriptions and representations e.g. point at an arrow and ask "What does this mean?"
- Record their subject ID and the date on the back of every sheet of their design. If there is more than one sheet, annotate each with its number in the sequence, e.g., "1 of 3", "2 of 3" etc.
- Thank the subject, with something encouraging, e.g., "Thank you, that's really helpful."

Therefore, the elicitation captures:

- number of components
- verbatim name of each component
- verbatim description of each component
- time from handover to completion of the decomposition (that is, up to "Would you talk me through your design?")

Data collection for the decomposition task includes:

- audio recording of the session (for backup)
- the experimenter's notes, including verbatim records of names and descriptions, and a note of the time taken for the decomposition
- the subject's marks-on-paper
- the experimenter's responses to the characterising questions (see sub-section iv below)

Reiterations:

- You may *not* ask leading questions, e.g. "What are the inter-relationships?"

- You may reflect back anything they have said, or refer to anything they have drawn e.g. "What's this?" or "Why does this arrow have two heads?"
- Use their terms verbatim. Do not try to paraphrase or clarify. Add no interpretations.
- If there are other things you want to note down, do so in a way that clearly distinguishes them as your own notes, and not verbatim quotations.
- Don't try to help the subject by suggesting answers or giving them pithier terms.
- Take as long as it takes. If necessary, ask the subject to slow down or repeat what they've said.
- Tape the session.

iii) Design Criteria elicitation

- Offer the subjects the cards.
- Ask them: "Thinking about the design you've just completed, what were the five most important, and five least important design criteria?" Record the numbers of the cards on the sheet. There is no need for the subject to rank order the cards (we're not interested in first, second, third etc.).
- Ask them: "If you were to undertake the same task, but in a team, what would be the five most important and the five least important design criteria?" Record the numbers on the sheet.
- Ask them: "Imagining that you had to do the same original task—on your own—but that you had to deliver a fully-functional result at this time tomorrow, what would be the five most important and the five least important design criteria?" Record the numbers on the sheet.
- Ask them: "Finally, if you were designing this system as the basis of a product line that would have a 5-year lifespan, what would be the five most important and the five least important design criteria?" Record the numbers on the sheet.
- Make sure that you record verbatim any comments they make along the way (e.g., "This one always stays in most important: It's my style, I can't design any other way." "Ah. With this scenario, everything moves." "These in the middle are only unimportant because this problem is so limited." etc.)
- Thank the subject.

Reiterations:

- Take as long as it takes. If necessary, ask the subject to slow down or repeat what they've said.
- Verify the accuracy of what you're recording; say it back to the subject. Repeat the numbers of the cards back to the subject.

iv) Decomposition task profile

The experimenter should answer the following "characterisation questions" immediately *after* the subject has left (and hence after the card sort task). We use the term "description" to mean their verbatim responses with regard to the *name* and *description* of the parts, and the term "representation" to mean the marks they made on paper. We use the term "notes" to mean other recorded responses, or your own written notes.

0. Number of components identified by the subject in the description, representation or other recorded responses.

1. Did the subject name and describe the solution in terms of component parts? (based on the description).
number named and described

2. Did the subject attempt to address the requirements of the specification; that is, does the solution (as expressed in the description and/or representation) map onto the specification evidently or explicitly?
yes (addresses all requirements)
partially (addresses at least 50% of the requirements)
hardly (addressed at least one but not 50% of the requirements)
no (addressed none)

3a. Are the component parts evident in the representation?
number

3b. Are the component parts evident in the description?
number

4.Does each component have a stated, intelligible, specific API?
number for which an interface is articulated
number for which interface (or 'what information is passed where') is fully articulated
number for which interface is incompletely articulated (An example of an "incomplete" articulation is "These things talk to each other")

5.Does each component have a stated, intelligible role? (based on description).
number

6. Did the subject identify the back-end as a component? (based on description and/or representation)
yes
no

7. Did the subject indicate the user explicitly as interacting with the system (e.g., as a stick figure)?
yes
no

8a. Did the subject ask questions about ambiguities and omissions in the specification (as distinct from questions about word meanings)?
yes
no

8b. Did the subject make explicit assumptions (in the description, representation or other recorded responses) about ambiguities and omissions in the specification.
yes (specify where: description, representation, other)
no

9. Categorise the type of solution (i.e., the organising principle underlying the solution structure) e.g.

event-driven
user-interface-based
functions

Design Brief

Getting People to Sleep

In some circles sleep deprivation has become a status symbol. Statements like “I pulled another all-nighter” and “I’ve slept only three hours in the last two days” are shared with pride, as listeners nod in admiration. Although celebrating self-deprivation has historical roots and is not likely to go away soon, it’s troubling when an educated culture rewards people for hurting themselves, and that includes missing sleep.

As Stanford sleep experts have stated, sleep deprivation is one of the leading health problems in the modern world. People with high levels of sleep debt get sick more often, have more difficulties in personal relationships, and are less productive and creative. The negative effects of sleep debt go on and on. In short, when you have too much sleep debt, you simply can’t enjoy life fully.

Your brief is **to design a "super alarm clock" for University students** to help them to manage their own sleep patterns, and also to provide data to support a research project into the extent of the problem in this community. You may assume that, for the prototype, each student will have a Pocket PC (or similar device) which is permanently connected to a network.

Your system will need to:

- Allow a student to set an alarm to wake themselves up.
- Allow a student to set an alarm to remind themselves to go to sleep.
- Record when a student tells the system that they are about to go to sleep.
- Record when a student tells the system that they have woken up, and whether it is due to an alarm or not (within 2 minutes of an alarm going off).
- Make recommendations as to when a student needs to go to sleep. This should include "yellow alerts" when the student will need sleep soon, and "red alerts" when they need to sleep now.
- Store the collected data in a server or database for later analysis by researchers. The server/database system (which will also trigger the yellow/red alerts) will be designed and implemented by another team. You should, however, indicate in your design the behaviour you expect from the back-end system.
- Report students who are becoming dangerously sleep-deprived to someone who cares about them (their mother?). This is indicated by a student being given three “red alerts" in a row.
- Provide reports to a student showing their sleep patterns over time, allowing them to see how often they have ignored alarms, and to identify clusters of dangerous, or beneficial, sleep behaviour.

In doing this you should (1) produce an initial solution that someone (not necessarily you) could work from (2) divide your solution into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does – in short, why it is a part. If important to your design, you may indicate an order to the parts, or add some additional detail as to how the parts fit together.

1

Hiding the internal workings of each part of the solution from the user, presenting them with a simple interface to its functionality.

2

Knowing how each part of the solution could be implemented.

3

Making sure related things appear together

4

Making sure that un-related things are linked via a narrow (internal) interface

5

Making sure the design is made up of appropriately-sized “chunks”

6

Being able to explain what each part of the solution is, and what it does, to yourself

7

Being able to explain what each part of the solution is, and what it does, to others

8

Constructing a solution using the simplest thing that gets the job done

9

Working to achieve a solution of maximum generality

10

Ensuring that the parts which make up the solution map onto the structure of the problem

11

Designing so that someone else can implement the solution with little (or no) additional information or domain expertise.

12

“Sanity-checking” the solution, by checking back to the specification

13

Designing a system that can be easily maintained.

14

Considering the technological implementation (target platform or device) and designing for efficient use of that resource

15

Using ideas that I know work

16

Expressing the functionality clearly

Design criteria elicitation record sheet

Subject id:	Most important	Least important
Current task "Thinking about the design you've just completed, what were the five most important, and five least important design criteria?"		
Task in team "If you were to undertake the same task, but in a team, what would be the five most important and the five least important design criteria?"		
Extreme time pressure "Imagining that you had to do the same original task—on your own—but that you had to deliver a fully-functional result at this time tomorrow, what would be the five most important and the five least important design criteria?"		
Longevity "Finally, if you were designing this system as the basis of a product line that would have a 5-year lifespan, what would be the five most important and the five least important design criteria?"		

Decomposition task profile sheet

Subject ID:		***Number***			
0	Number of components identified by the subject in the description, representation or other recorded responses				
1	Did the subject name and describe the solution in terms of component parts? (based on the description).				
2	Did the subject attempt to address the requirements of the specification; that is, does the solution (as expressed in the description and/or representation) map onto the specification evidently or explicitly?	***yes***	***partially***	***hardly***	***no***
		Number			
3a.	Are the component parts evident in the representation?				
3b	Are the component parts evident in the description?				
4	Does each component have a stated, intelligible, specific API?				
	▪ ***number*** for which an interface is articulated				
	▪ ***number*** for which interface (or 'what information is passed where') is fully articulated				
	▪ ***number*** for which interface is incompletely articulated (An example of an "incomplete" articulation is "These things talk to each other")				
5	Does each component have a stated, intelligible role? (based on description).				
		Yes		***No***	
6	Did the subject identify the back-end as a component? (based on description and/or representation)				
7	Did the subject indicate the user explicitly as interacting with the system (e.g., as a stick figure)?				
8a	Did the subject ask questions about ambiguities and omissions in the specification (as distinct from questions about word meanings)?				
8b	Did the subject make explicit assumptions (in the description, representation or other recorded responses) about ambiguities and omissions in the specification.				
9	Categorise the type of solution (i.e., the organising principle underlying the solution structure)				

3. Analysis Protocol

For each subject, collect & record

- The time taken to complete the decomposition task
- All components with names and descriptions
- Note the number of components per subject
- Prioritisation data

Across your subjects

- Identify verbatim commonality between component names ("verbatim" means that they use exactly the same terms to name the parts.)
- Identify gist commonality between component names and descriptions. ("gist" means that they are saying the same thing, but expressing it in different ways, using different words.)
- Identify decomposition solutions that have similar profiles (based on the characterising questions)
- Identify co-occurrence in "most important" and "least important" categories

By December …

- We will make available a web upload page, so that you can enter your data and start to compare it with others'.

4. Background

Design of study materials & pilot studies

The design of the tasks had a number of developmental inputs.

1. Expert elicitation

At SIGCSE 2003, we interviewed a number of eminent CS educators. We asked them what they would like to know about:

- what their students know, understand, or experience about software design
- what criteria they would apply to determine these things
- what criteria they would like their students to apply in assessing alternative software designs;
- on what dimensions they would vary alternatives if they were presenting alternative software designs for comparison and critique.

During one of those interviews, Clayton Lewis suggested the use of a decomposition task to reveal students' ability to identify and articulate underlying software design structure.

2. First design: a categorisation task

We discussed a first task design. This was a categorisation task which presented students with several descriptions of alternative software designs—varied on dimensions identified by the educators. Their task was to categorise them on a "goodness scale", given different context scenarios.

3. Pilot study of categorisation task

Josh Tenenberg ran a pilot study with his students, presenting them with descriptions of six different software designs and asking them to sort the designs using their own criteria. This task was found to take too long and to be too demanding, especially on "comprehension time".

4. Pilot study of first decomposition task

We returned to Clayton Lewis's suggestion of a decomposition task. Josh Tenenberg and Tammy VandeGrift constructed a first decomposition task, using a sales and information system for a video rental store as the design problem. We conducted a pilot study of the task with 6 academics and post-graduate students. The choice of the video rental store scenario proved too simplistic, as it admitted a straightforward, standard database solution which revealed little or no design reasoning.

5. Pilot study of second decomposition task

We constructed a second decomposition task, using a 'smart alarm clock' as the design problem, building on a scenario composed by B.J. Fogg, with elaboration by Ian Utting. We conducted a pilot study of this task with 9 academics, post-graduate students and undergraduate students from two different universities. The subjects found the scenario to be comprehensible and feasible. They were able to complete it in between 10 and 30 minutes.

6. Development of characterising questions

On the basis of a qualitative analysis of the decompositions produced by the first 4 subjects in the pilot study of the alarm clock task, in discussion with expert educators with different pedagogic approaches, we composed a set of 'characterising questions' which identified important distinctions among the decompositions. We then tried the 'characterising questions' with the subsequent subjects. We found them

to be adequate (that is, usable and sufficient to signal some distinctions among the decompositions) but not yet complete.

7. Background to the prioritisation task

Newstetter and McCracken published a study on 'novice conceptions of design' which involved two tasks:

i) identifying the 5 most and 5 least important design processes from a set of 16, and
ii) giving marks out of 10 for two design scenarios, which embodied different design processes.

Sally Fincher had been replicating that study with her own students for several years, and had collected substantial data. However, although the tasks were tractable, the focus of the intervention was generic design activities and students' ability to recognise a design processes, regardless of context. These aims were too far from our domain of software design.

8. Identifying new design criteria

We decided to use the structure of the first task of the Newstetter and McCracken study, but adapted to our own purposes. Drawing on a qualitative analysis of the interviews at SIGCSE, we identified a collection of software design criteria identified by educators as being of interest or importance. We identified the 16 most prominent of those criteria. Most were expressed by the educators using single-word, often technical terms, such as *coupling*, *encapsulation*, and *intelligibility*. Our experience with previous studies was that students were often unable to 'unpack' such professional terms, and so we expressed each of the criteria as a descriptive phrase. We then checked our descriptors with three CS educators, presenting them with both the descriptors and the original terms and asking them to match the two—which they were able to do accurately. We also presented one educator with just the phrases and asked them to express them as single-word terms. He was able to do this, and his terms matched ours.

9. Developing context scenarios

Because "design" is a closely contextualized activity, we needed to extend data collection beyond the simple, single decomposition task. Consequently, we devised four scenarios for 'typical' software design contexts which exposed different criteria. The selected scenarios were vetted by two expert CS educators.

10. Pilot study of the prioritisation task

We ran a pilot study of the prioritisation task using seven post-graduate and undergraduate students from two universities. (Five of these performed the prioritisation task after the decomposition task, as part of the complete protocol.) The stimuli were usable by all subjects; all were able to complete the task. The completed it in between 10 and 30 minutes. Preliminary analysis suggests possible novice/expert differences.

5. Literature

References (included in the Kit)

Literature that contributes to the question

Cynthia J. Atman, Justin R. Chimka, Karen M. Bursic and Heather M. Nachtmann (1999) *A comparison of freshman and senior engineering design processes.* Design Studies, 20, 131-152.

Simon P. Davies, Adrian M. Castell (1992) *Doing design and describing it: Accounting for divergent perspectives in software design.* Proceedings of the 4th Annual Workshop of the Psychology of Programming Interest Group.

Vikki Fix , Susan Wiedenbeck , Jean Scholtz (1993) *Mental representations of programs by novices and experts.* Proceedings of the SIGCHI conference on Human factors in computing systems.

W. Michael McCracken (in press) *Research on Learning to Design Software*. [Do not quote from this draft.]

Literature that contributes to the methodology

M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y.B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. (2001) *A multinational, multi-institutional study of assessment of programming skills of first-year CS students.* Proceedings of ITiCSE.

Wendy C. Newstetter, W. Michael McCracken (2001) *Novice conceptions of design: Implications for the design of learning environments.* In Charles M. Eastman, W. Michael McCracken, Wendy C. Newstetter (eds.) *Design Knowing and Learning: Cognition in Design Education*. Elsevier.

Background (optional reading)

… on Software Design & prior knowledge structures

Beth Adelson and Elliot Soloway (1988) *A model of software design.* In Michelene Chi, Robert Glaser, M.J. Farr (eds.) *The Nature of Expertise.* Lawrence Erlbaum. 185-208.

Françoise Détienne (2001) *Software Design-Cognitive Aspects.* Springer-Verlag (Practitioner Series). [Health Warning – great content: terrible translation]

Vinod Goel and Peter Pirolli (1992) *The structure of design problem spaces*. Cognitive Science, 16, 395-429

Raymonde Guindon (1990) *Designing the design process: Exploiting opportunistic thoughts*. Human-Computer Interaction, 5, 305-344.

R. Guindon, H. Krasner, and B. Curtis (1987) *Breakdowns and processes during the early activities of software design by professionals.* In G.M. Olson, S. Sheppard, and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*. Ablex. 65-82.

M.C. Linn & M. Clancy (1992) *Can Experts' Explanations Help Students Develop Program Design Skills?* Int'l. J. Man-Machine Studies, 36, 4, 511-551.

M.C. Linn & M. Clancy (1992) *The Case for Case Studies of Programming Problems.* Communications of the ACM, 35, 3, 121-132.

E. Soloway, and K. Ehrlich (1984) *Empirical studies of programming knowledge.* IEEE Transactions on Software Engineering, 10 (5), 595-609.

Visser, W. (1990). More or less following a plan during design: opportunistic deviations in specification. *Int. Journal of Man-Machine Studies*, **33**, 247-278.

… on expert/novice distinctions & knowledge consolidation

Simon P. Davies, David J. Gilmore, Thomas R. G. Green (1995) *Are objects that important? The effects of expertise and familiarity on the classification of object-oriented code.* Human-Computer Interaction, 10 (2&3), 227-248 and http://www.psychology.nottingham.ac.uk/staff/dg/HCI_paper/HCI_paper.html

M. W. Eysenck and M. T. Keane (1995) *A Handbook of Cognitive Psychology,* 3rd edition. Psychology Press.

J.P. Guilford (1967) *The Nature of Human Intelligence.* McGraw-Hill.

R. Jeffries, A.A. Turner, P.G. Polson, and M.E. Atwood (1981) *The processes involved in designing software.* In J. Anderson (ed) *Cognitive Skills and Their Acquisition*. Lawrence Erlbaum Associates.

S. Kaplan, L. Gruppen, L. M. Levanthal and F. Board (1986) *The Components of Expertise: a Cross-Disciplinary Review* (University of Michigan, Ann Arbor).

… on "deep learning"

Marton, F. & Saljo, R. (1976) *On qualitative differences in learning: Outcome and process*. British Journal of Educational Psychology, 46, 4-11.

Marton, F. & Saljo, R. (1997) *Approaches to learning*. In F. Marton, D. Hounsell & Entwistle, N. (Eds.) *The experience of learning. Implications for teaching and studying in higher education*. Edinburgh: Scottish Academic Press.

Richardson, J.T.E., Eysenck, M.W. & Warren Piper, D. (eds) (1987) *Student learning: Research into education and cognitive psychology*. Milton Keynes: Open University Press.